IM227119 - Making the Leap from iLogic to Add-Ins - or Not?

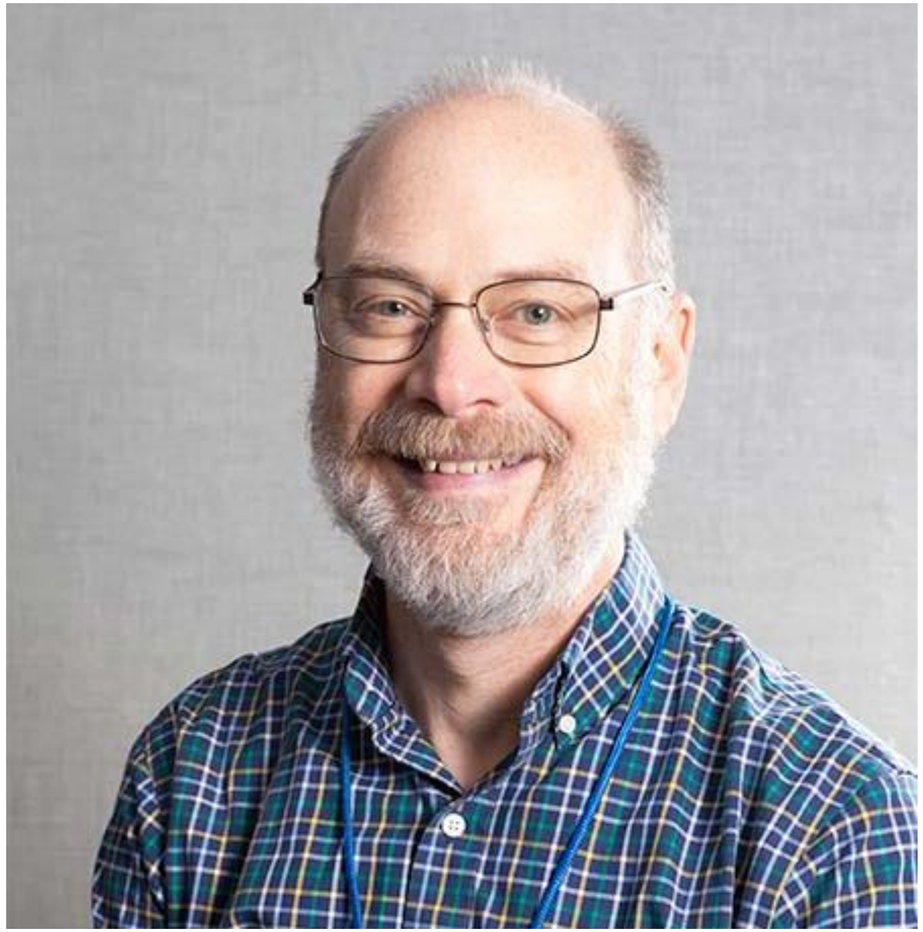
Jon Balgley

Software Architect, Autodesk

AUTODESK UNIVERSITY

Rev: 3-Nov-2018

Welcome



About the speaker

Jon Balgley

Jon has worked at Autodesk since 2005, and has designed, developed and taught CAD automation technologies since the 1980’s. Recent work includes Design Automation API for Inventor, Configurator 360, and iLogic.

Learning Objectives

- Learn how to identify key indications for when to use rules versus add-ins versus class libraries
- Understand the pros and cons of each technology versus the others
- Understand the basic components and software “plumbing” required for an add-in
- Discover how easy it is to use a custom class library from iLogic rules

Agenda

- Quick overview of iLogic strengths
- “Class libraries” – in conjunction with iLogic
- Addins
- Comparison of technologies
- Q&A

Not on the Agenda

- Details about iLogic
- How to program
- How to use the Inventor API, in general
- Exact step-by-step’s … just highlights

iLogic Rules Overview

iLogic Rules Overview

iLogic rules are VB.Net code…

- …with some aspects hidden, for ease-of-use (e.g., “Sub Main”) …
- … provided with a built-in editor with “snippets”, etc., …
- … provided with a built-in/seamless/invisible compiler …
- … invoked/executed/triggered automatically …
- … with easy-to-use “wrappers” for some Inventor API and other .Net functions (e.g., ThisDoc, Feature.IsActive, GoExcel, etc.) …
- … while allowing seamless use of Inventor API and .Net functions directly …
- …and some minor extensions (e.g., parameters and units) …
- Two kinds of iLogic rules:
  - Internal; stored in the IPT/IAM/IDW file.
  - External; stored in separate file

Easy to get started!

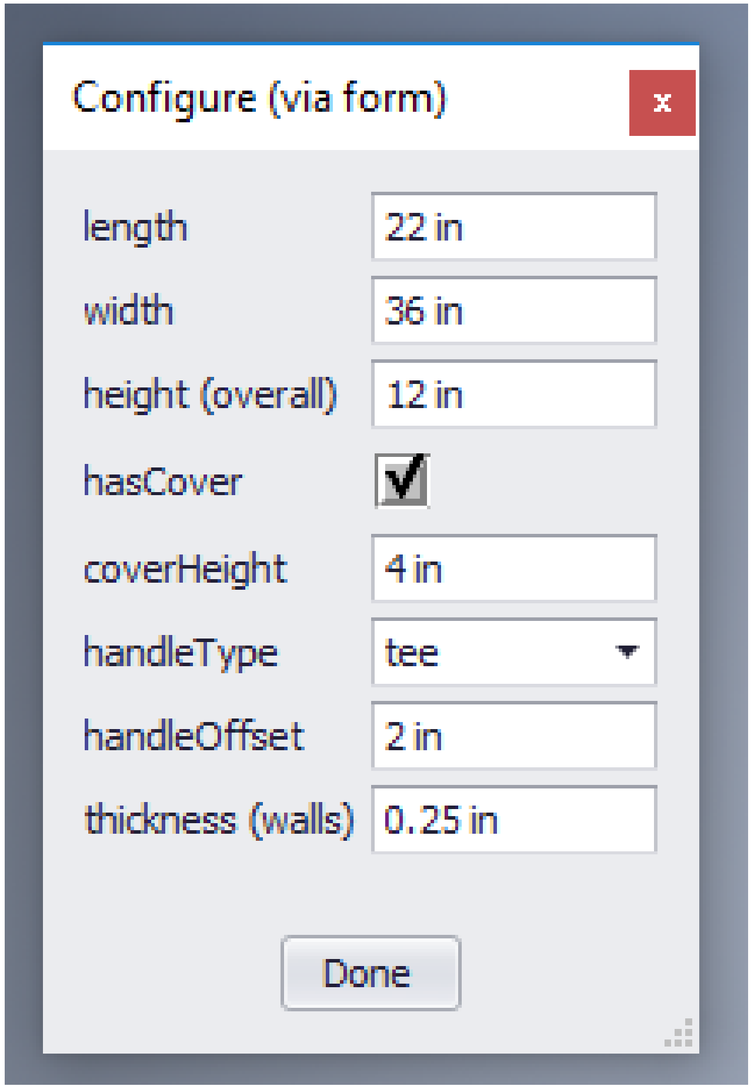
Tightly integrated with Inventor!

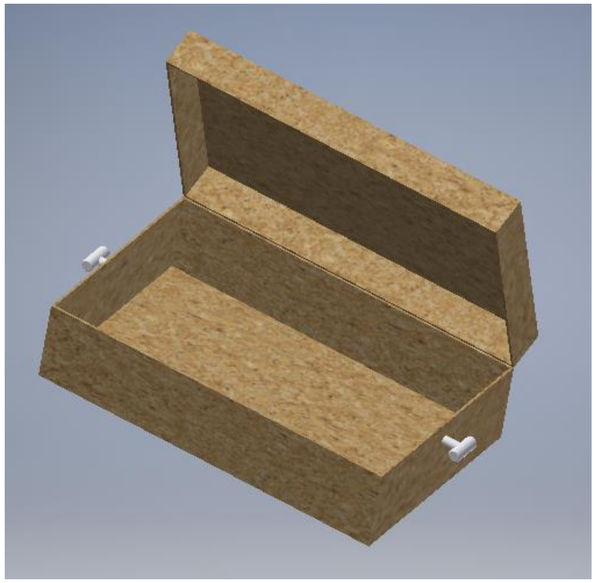
iLogic Forms Overview

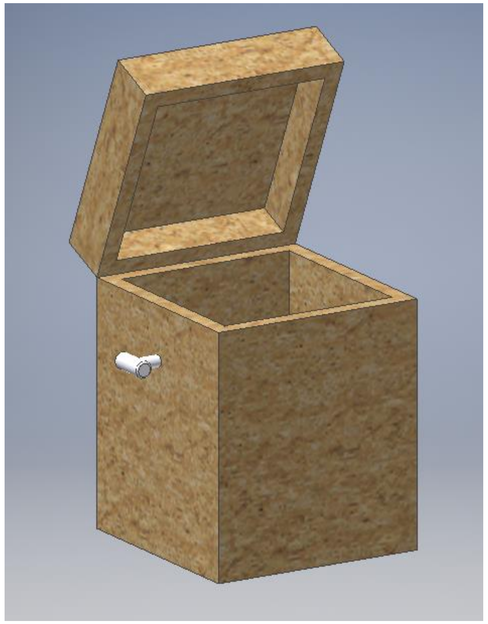
iLogic forms …

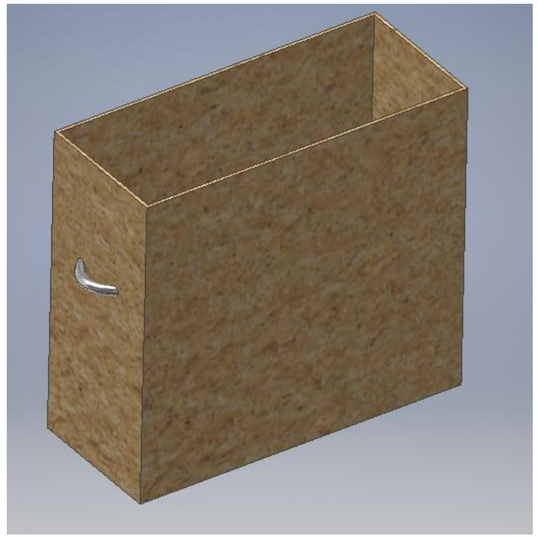
- … are a simple pop-up “dialog box” (window)…
- … have easy drag-and-drop construction
- …support parameters, iProperties, and rules-as-buttons
- SIMPLE = LIMITED

Pure iLogic Example









Summary

EASY TO USE / INTEGRATED WITH INVENTOR

Everything built-in. Runs automatically.

INTERNAL RULES IMPLY COPYING IPT/IAM/IDW'S

Copying has pro's and con's.

EXTERNAL RULES

Choose storage location. Still "built-in" Somewhat easier to maintain & update.

EVENTS

Another easy way to trigger rules. Different events for external rules.

iLogic + Class Library

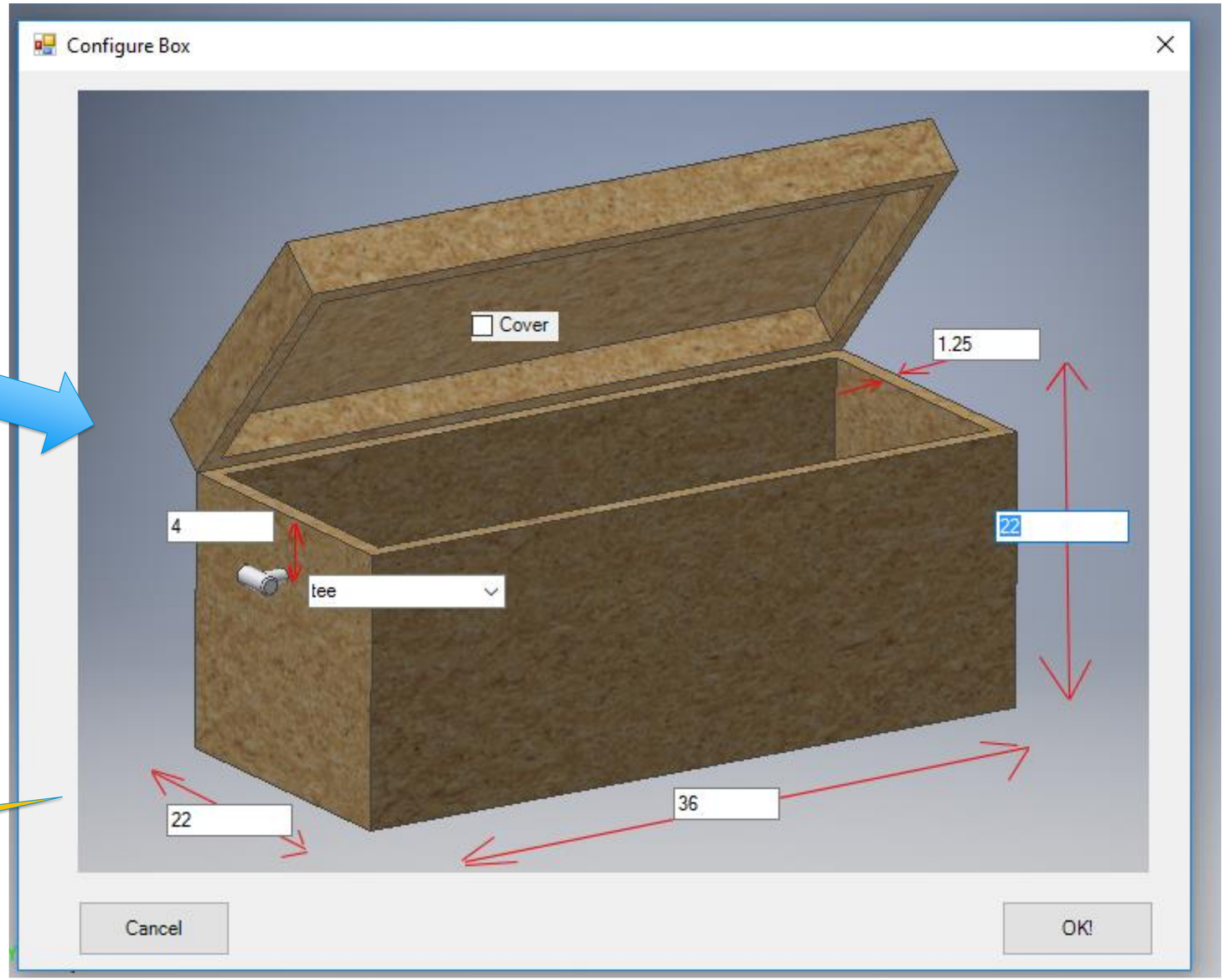
iLogic + Class Library

Two major uses:

1. Add simple dialogs/UI … beyond Forms, but less than Addins

2. Write more complex code involving Inventor API (picture later)

- Easy “next step” beyond iLogic, in conjunction with iLogic



Please laugh at the terrible graphic design!

What can a Class Library Do?

SHOW A CUSTOM “FORM” (AKA DIALOG)

It’s easy to make custom forms in Visual Studio. There are a variety of techniques.

USE THE INVENTOR API

The API was designed/intended to by used from code libraries.

Better debugging, better environment. Visual Studio is great for larger and more complex code; iLogic Rule Editor is ~~great~~ ~~barely adequate~~ works OK for small things.

BE INVOKED FROM ILOGIC

You call/invoke the class library’s capabilities from iLogic.

What is a Class Library?

IT’S A DLL

Loaded into a “main” program

EXPOSES PUBLIC CLASSES

These can be easily used in iLogic rules

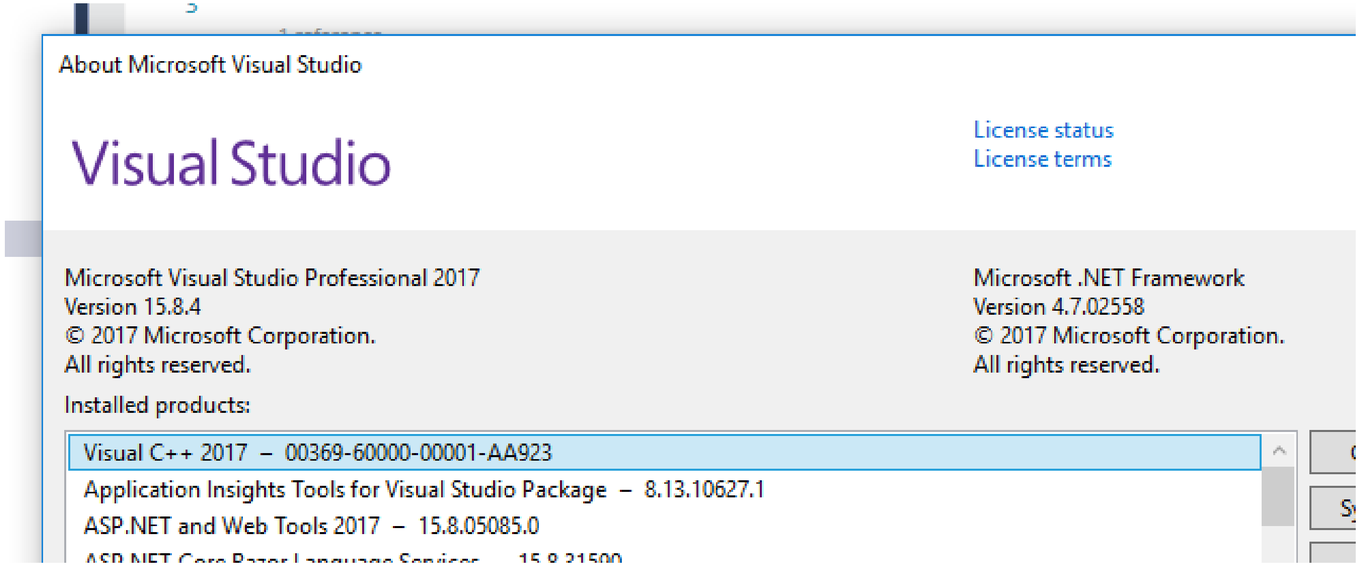
CLASSES HAVE PROPERTIES AND METHODS

…that’s how you use the classes from your rules.

Visual Studio

Class libraries are programs!

- Get Visual Studio
- VB.Net / C# / C++
- Source control



Only the most minimal intro to VS today!

1. Using a Custom Dialog

Rule (when complete)

New rule, suppressed

Component | Position | Relationships | Pattern | Manage | Productivity | Work Features | Simplification

Model | iLogic

Rules | Forms | Global Forms | External Rules

- boxAssy.iam
  - DimRangeCheck
  - ChangeHandleType
  - ToggleCover
  - ChangeDimsOfBox
  - SetupHandleTypes
  - ShowVersion
  - ConfigurationDialog
  - logo

iLogic Log

Edit Rule: ConfigurationDialog

Snippets

System | Custom

- Replace with LOD
- Replace iPart
- Color
- Visible
- MakePath
- SkipDocumentSave
- iLogic Assemblies/Components
- Positioning
- iProperties
- Excel Data Links
- iParts
- iFeatures
- Relationships (classic)
- Relationships (Add)
- Measure
- Forms
- MessageBox
- Document
  - Path
  - FileName
  - PathAndFileName
  - ChangeExtension
  - WorkspacePath
  - Launch Document
  - ThisDoc.Save
  - UpdateWhenDone
  - RuleParametersOutput

Model | File Tree | Files | Options | Search and Replace | Wizards

- boxAssy.iam
  - Model Parameters
  - User Parameters
  - Relationships
  - Origin

Parameters | Names

Parameter

If...Then...End If | Keywords | Operato

```
addreference "myclasslib_01"

iLogicVb.UpdateWhenDone = True

Dim dlg As New MyClassLib_01.BigDialog

dlg.dHeight = height
dlg.dWidth = width
dlg.dLength = length
dlg.Thickness = thickness
dlg.HandleOffset = handleOffset
dlg.handleTypes = MultiValue.List("handleType")
dlg.HandleType = handleType
dlg.HasCover = hasCover

Dim result = dlg.ShowDialog()

If (result = System.Windows.Forms.DialogResult.OK) Then
```

Rule: Using a Custom Dialog

Add reference

Make instance

Push values in

Show to user
(form_load, ok_click)

Pull values out

All other triggered
rules run

If...Then...End If ▾ Keywords ▾

```
addreference "myclasslib_01"

Dim dlg As New MyClassLib_01.BigDialog

dlg.dHeight = height
dlg.dWidth = width
dlg.dLength = length
dlg.Thickness = thickness
dlg.HandleOffset = handleOffset
dlg.handleTypes = MultiValue.List("handleType")
dlg.HandleType = handleType
dlg.HasCover = hasCover

Dim result = dlg.ShowDialog()

If (result = System.Windows.Forms.DialogResult.OK) Then
    height = dlg.dHeight
    width = dlg.dWidth
    length = dlg.dLength
    thickness = dlg.Thickness
    handleOffset = dlg.HandleOffset
    handleType = dlg.handleType
    hasCover = dlg.HasCover
End If
```

Start New Project

Language / type

Class Library

Name

Source Control

New Project

Recent

Installed

Visual C#

Visual Basic

Windows Universal

Windows Desktop

Web

.NET Core

.NET Standard

WCF

Visual C++

Visual F#

SQL Server

AWS

JavaScript

Python

TypeScript

Not finding what you are looking for?

Open Visual Studio Installer

Sort by: Default

Search (Ctrl+E)

WPF App (.NET Framework)	Visual Basic
Windows Forms App (.NET Framework)	Visual Basic
Console App (.NET Framework)	Visual Basic
Class Library (.NET Framework)	Visual Basic
Windows Service (.NET Framework)	Visual Basic
Empty Project (.NET Framework)	Visual Basic
WPF Browser App (.NET Framework)	Visual Basic
WPF User Control Library (.NET Framework)	Visual Basic
WPF Custom Control Library (.NET Framework)	Visual Basic
Windows Forms Control Library (.NET Framework)	Visual Basic

Type: Visual Basic

A project for creating a VB class library (.dll)

Name: MyClassLibrary

Location: C:\Users\balglej\source\repos

Browse...

Solution: Create new solution

Solution name: MyClassLibrary

Framework: .NET Framework 4.6.1

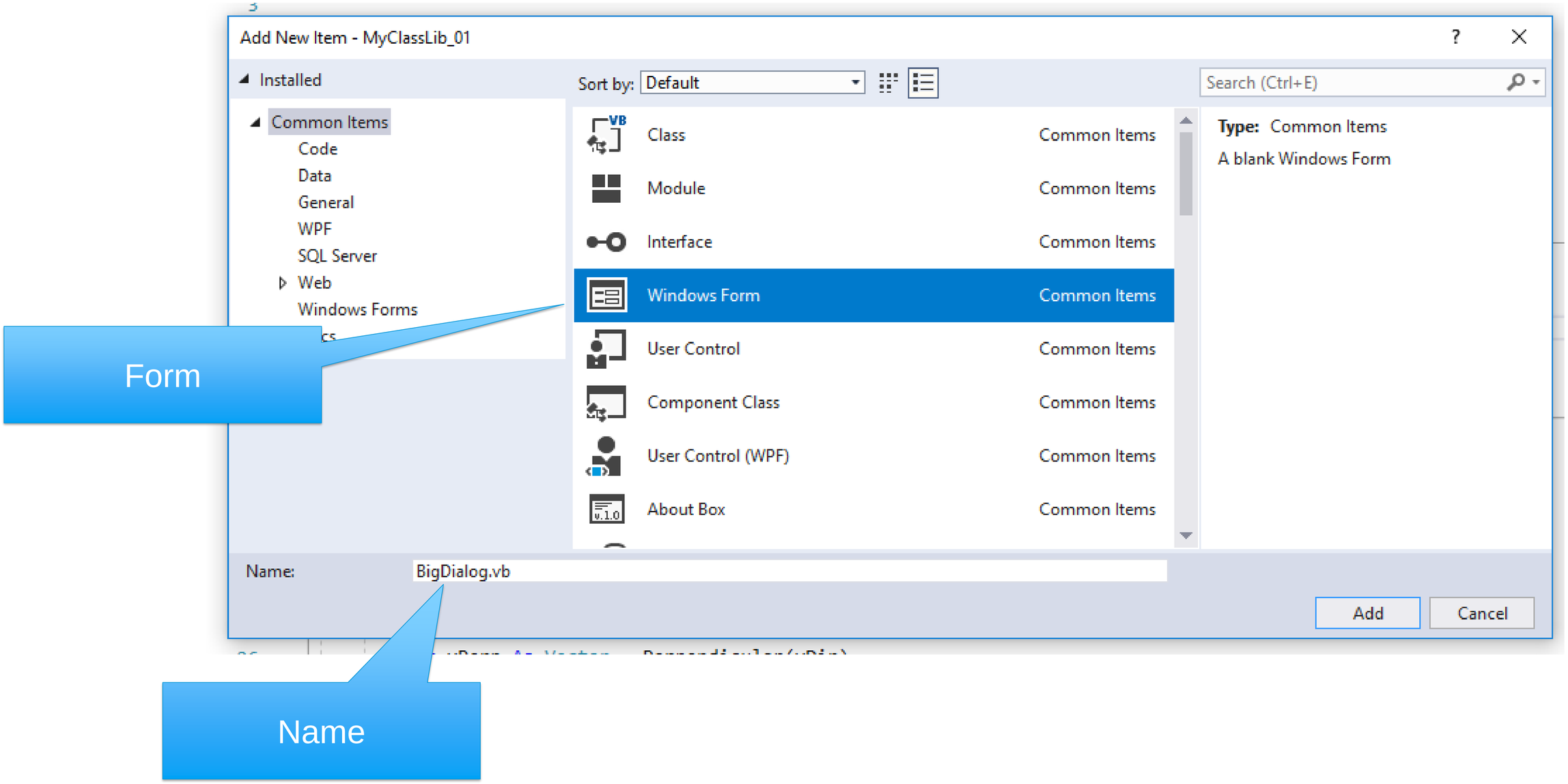
☑ Create directory for solution

☐ Add to Source Control

OK

Cancel

Add New Item (form)



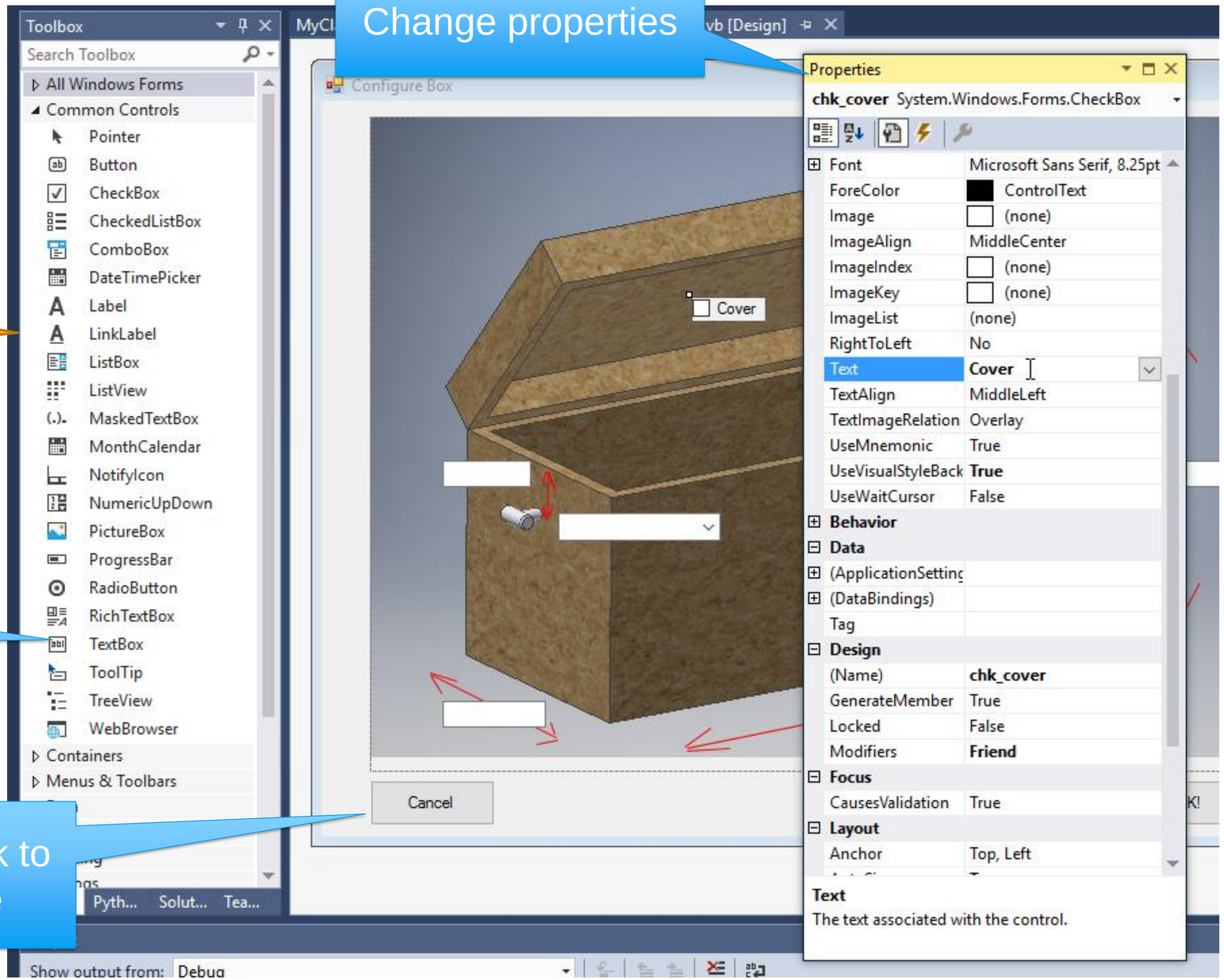
Design the Form

Using old-style "WinForms" for simplicity, and similarity to iLogic Forms

Change properties

Drag in new controls

Double-click to edit code



Define Properties on Form Class

Properties

Form’s class

Solution Explorer

Solution 'MyClassLib_01' (1 project)
- MyClassLib_01
  - My Project
  - References
  - Resources
    - boxAssy.png
  - BigDialog.vb
  - GeomTool.vb

crooked.ipt.OBB_usingClassLib.vb | GeomTool.vb | Dialog.vb [Design] | MyClass

MyClassLib_01 | btn_OK

```
1 reference
Public Class BigDialog
    Private m_handleTypes As ArrayList
    Private m_ht As String
    2 references
    Public Property dWidth As Double
    2 references
    Public Property dHeight As Double
    2 references
    Public Property dLength As Double
    1 reference
    Public Property HandleType As String ...
    2 references
    Public Property HasCover As Boolean
    2 references
    Public Property HandleOffset As Double
    2 references
    Public Property Thickness As Double
    0 references
    Public WriteOnly Property handleTypes As ArrayList ...
```

iLogic rule will push/pull values using these properties

Implement Events

Form is shown – populate controls from properties

```
0 references
Private Sub BigDialog_Load(sender As Object, e As EventArgs) Handles MyBase.Load
    'Main dimensions
    txt_Length.Text = dLength
    txt_Width.Text = dWidth
    txt_Height.Text = dHeight
    txt_handleOffset.Text = HandleOffset
    txt_thickness.Text = Thickness

    'Handles
    cmb_handle.Items.Clear()
    For Each item In m_handleTypes
        cmb_handle.Items.Add(item)
    Next item

    'Cover
    chk_cover.Checked = HasCover
End Sub
```

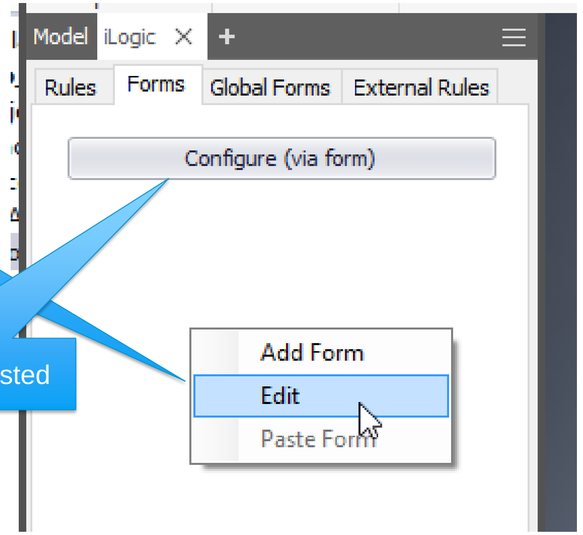
User clicks ‘ok’ -- populate properties from controls

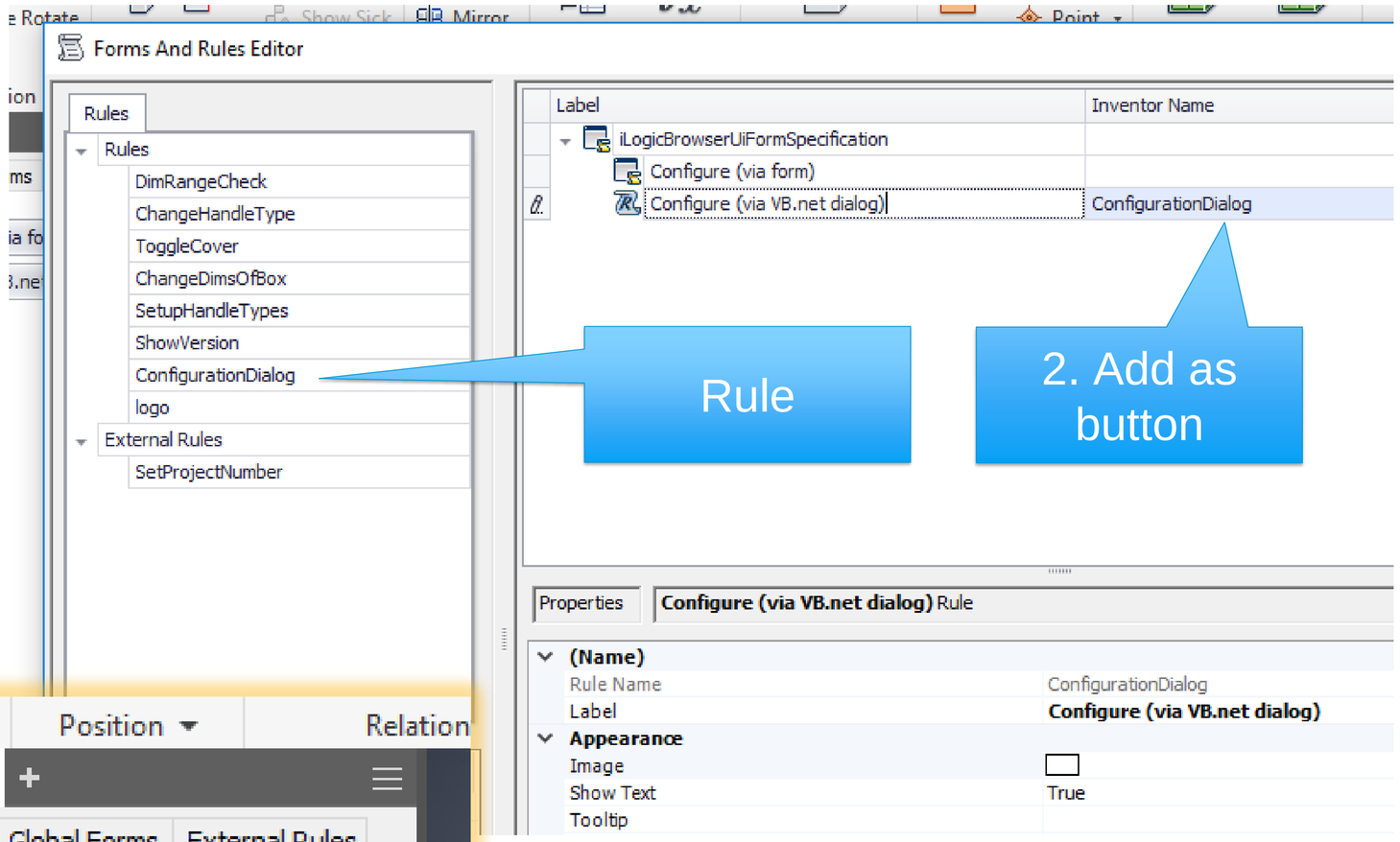
```
0 references
Private Sub btn_OK_Click(sender As Object, e As EventArgs) Handles btn_OK.Click
    Try
        dHeight = Double.Parse(txt_Height.Text)
        dLength = Double.Parse(txt_Length.Text)
        dWidth = Double.Parse(txt_Width.Text)
        HandleOffset = Double.Parse(txt_handleOffset.Text)
        Thickness = Double.Parse(txt_thickness.Text)
    Catch
        MsgBox("Something wasn't numeric")
        Me.DialogResult = Windows.Forms.DialogResult.Abort
        Exit Sub
    End Try

    Me.Hide()
End Sub
```

Access Rule from Forms Tab (!)

1. Forms Tab *is* a form, can “edit” it

Ordinary form already listed

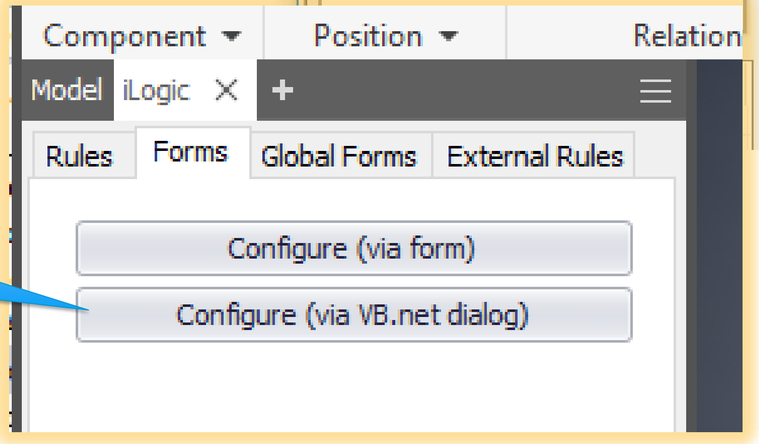




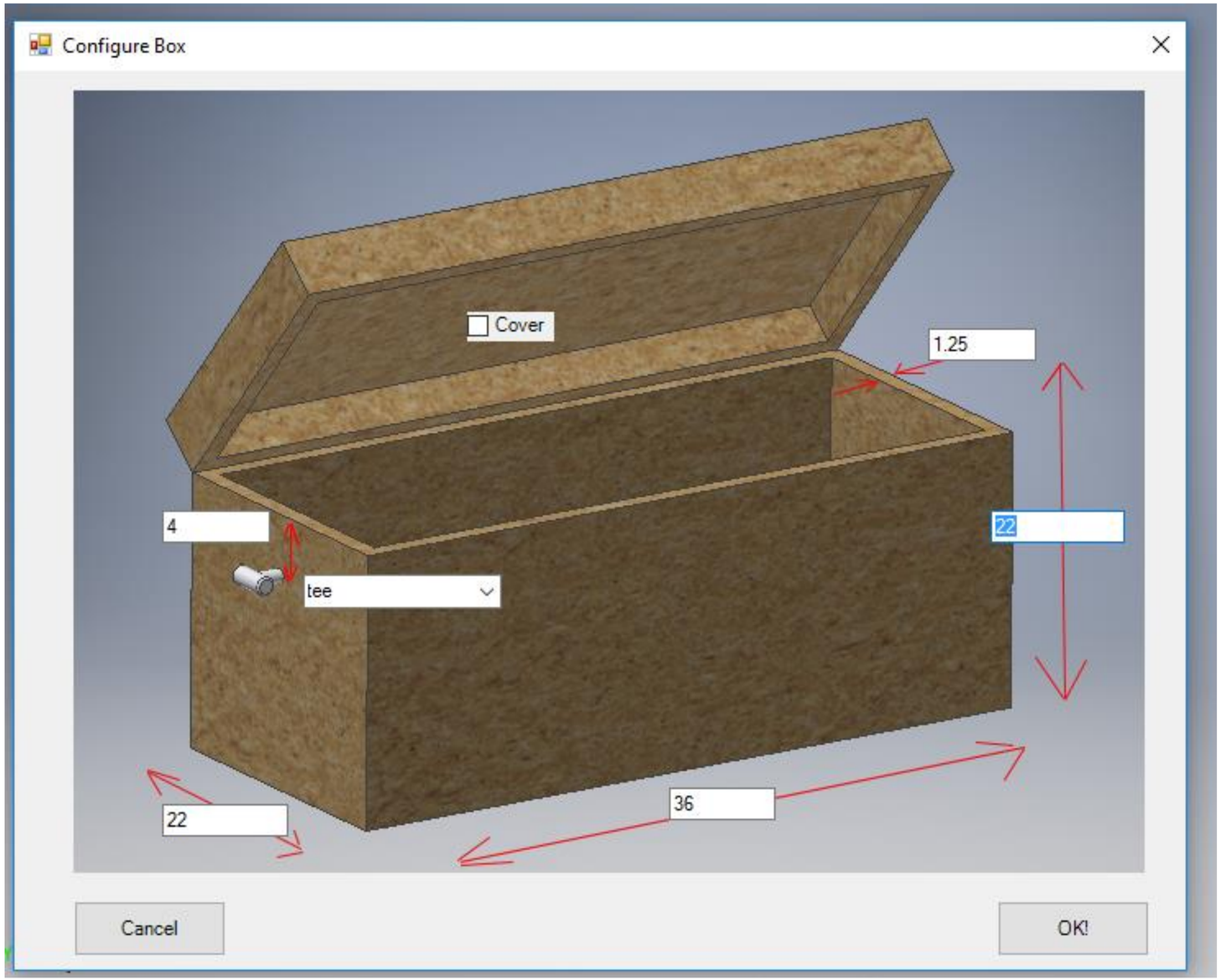
Rule

2. Add as button

3. Rule appears on Forms tab



Success!



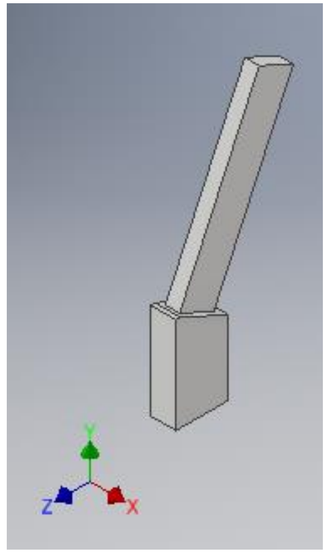
2. Using the Inventor API More Deeply

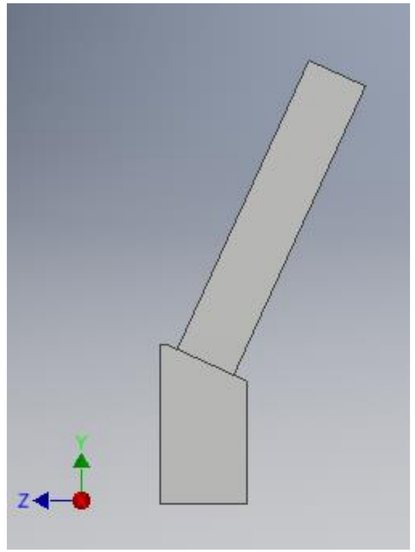
Problem:
Oriented Bounding Box

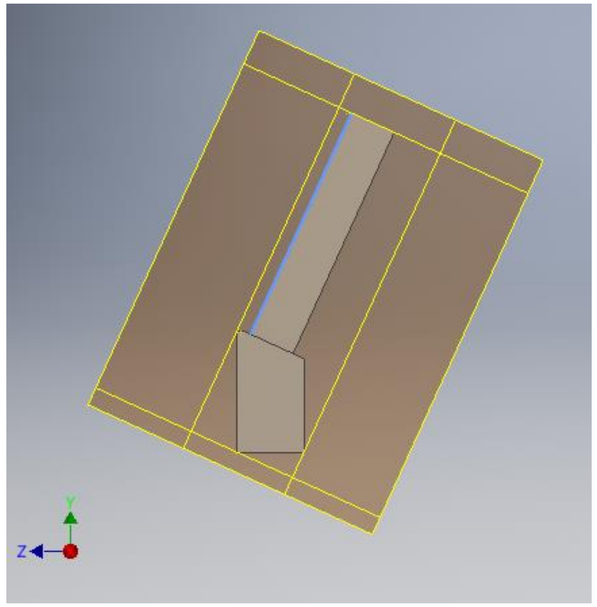
- Allow user to select linear edge
- Create workplanes at “far points”
- Find orthogonal directions
- Create workplanes at those “far points”

Plan

- Use existing Inventor API “farpoint” method.
- Make class & method to generate wp’s
- Do simple edge-selection from rule
- Call method from rule







Rule (when complete)

Add reference

Use API for simple selection

Confirm selection is an Edge

Get (unit) vector from edge

Make instance of class, call “entry point” method

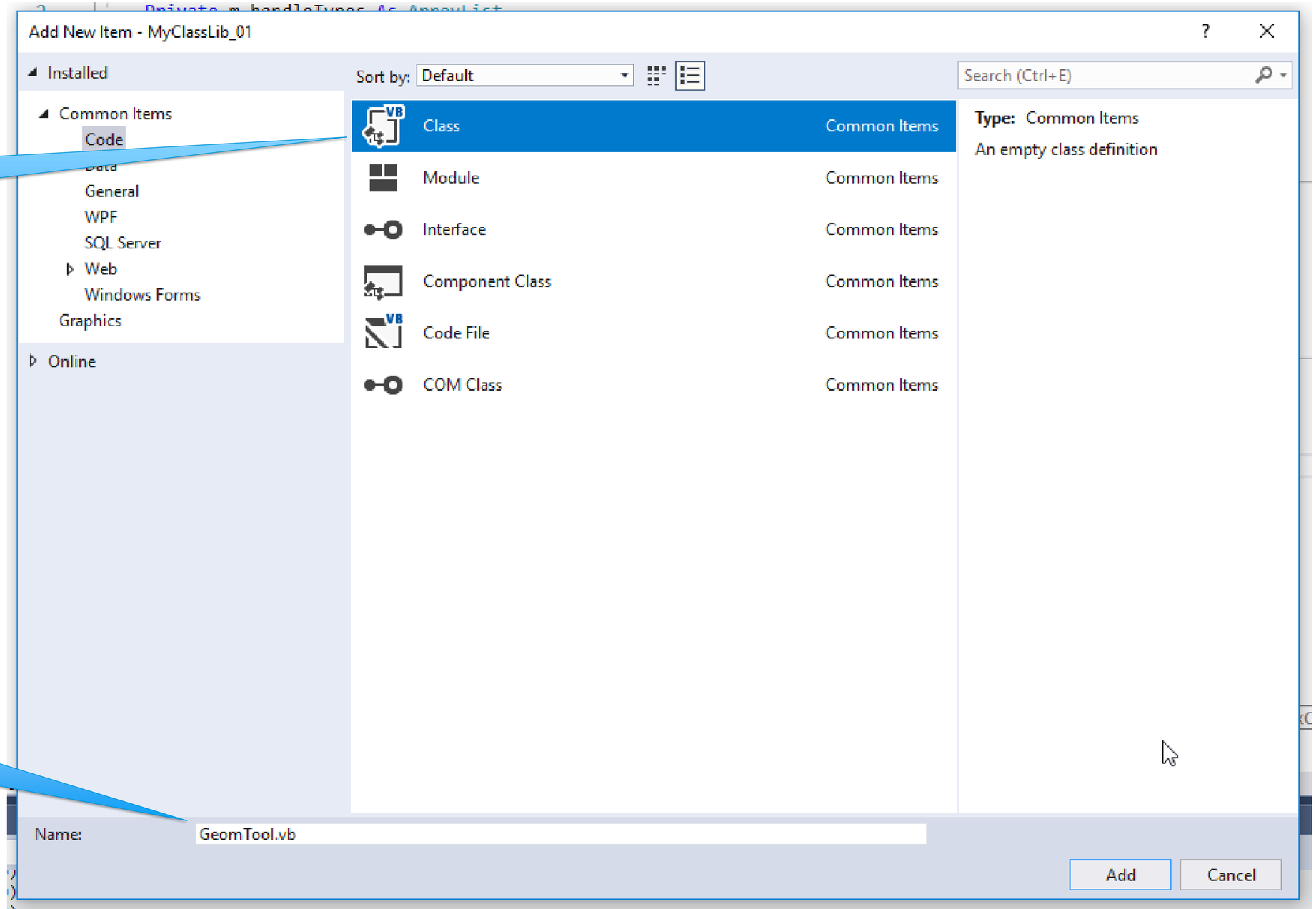
If...Then...End If ▾ Keywords ▾ Operators ▾ Header...

```
addreference "myclasslib_01"


Dim oTG As TransientGeometry = ThisApplication.TransientGeometry

Dim ss As SelectSet = ThisDoc.Document.SelectSet
If (ss.Count <> 1) Then
    MessageBox.Show("Please select a linear edge first")
    Exit Sub
Else
    'Assume it's a linear edge for now
    Dim edge As edge = ss(1)
    If (edge Is Nothing) Then
        MessageBox.Show("Please select a linear edge first")
        Exit Sub
    Else
        Dim vec As UnitVector = edge.Geometry.direction

        Dim geom As New MyClassLib_01.GeomTool(ThisApplication)
        geom.ShowBoundsAlongVector(ThisDoc.Document, vec.AsVector)

        End If
End If
```

Add New Item (Class)

Class Library

Name



Class / Method

Add reference

Class name and access

Member variables

Constructor

Entry point

Helper methods

GeomTool.vb | BigDialog.vb | MyClassLib_01 | MyClass_01.vb | BigDialog.v

MyClassLib_01 | GeomTool

```
1   Imports Inventor
2
3
    1 reference
4   Public Class GeomTool
5
6       Private m_app As Application
7       Private m_tg As TransientGeometry
8
9       Private m_doc As PartDocument
10      Private m_pcd As PartComponentDefinition
11      Private m_body As SurfaceBody
12
13
        0 references
14      Public Sub New(app As Application) ...
18
        0 references
19      Public Sub ShowBoundsAlongVector(doc As PartDocument, vDir A
35
        3 references
36      Private Sub ShowBoundsAlongVector_internal(vDir As Vector, v
59
60      ' Return an arbitrary vector perpendicular to 'v' ...
        1 reference
62      Public Function Perpendicular(v As Vector) As Vector ...
87
88  End Class
89
```

Class Constructor

Pass in API object

```
13
    0 references
14  Public Sub New(app As Application)
15      m_app = app
        m_tg = m_app.TransientGeometry
17  End Sub
18
```

Hold onto convenient objects

Entry Point Method – Call the helpers

```
0 references
Public Sub ShowBoundsAlongVector(doc As PartDocument, vDir As Vector)

    m_doc = doc
    m_pcd = m_doc.ComponentDefinition
    Dim bodies As SurfaceBodies = m_pcd.SurfaceBodies
    m_body = bodies(1)

    Dim vPerp As Vector = Perpendicular(vDir)
    Dim vCross As Vector
    vCross = vDir.CrossProduct(vPerp)

    ShowBoundsAlongVector_internal(vDir, vPerp, vCross)
    ShowBoundsAlongVector_internal(vPerp, vCross, vDir)
    ShowBoundsAlongVector_internal(vCross, vDir, vPerp)

End Sub
```

Pass in important data

Find and hold key data

Compute other two orthogonal vectors

Call helper 3x

Helper Method – Creates one pair

Convert to unit vec

Wants unit vector!

Can't scale unit vector!

Get the far-points

Add workplanes (vDir is 'normal')

Add workpoints

```
3 references
Private Sub ShowBoundsAlongVector_internal(vDir As Vector, vPerp As Vector, vCross As Vector)

    Dim unitDir As UnitVector = vDir.AsUnitVector
    Dim revDir As Vector = vDir.Copy
    revDir.ScaleBy(-1)
    Dim revUnitDir As UnitVector = revDir.AsUnitVector

    Dim pt1 As Point = m_tg.GetFarmostPoint(m_body, unitDir)
    Dim pt2 As Point = m_tg.GetFarmostPoint(m_body, revUnitDir)

    Dim wp1, wp2 As WorkPlane
    wp1 = m_pcd.WorkPlanes.AddFixed(pt1, vPerp.AsUnitVector, YAxis:=vCross.AsUnitVec
    'wp1.Name = "wp1"
    wp1.AutoResize = True

    wp2 = m_pcd.WorkPlanes.AddFixed(pt2, vPerp.AsUnitVector, YAxis:=vCross.AsUnitVector)
    'wp2.Name = "wp2"
    wp2.AutoResize = True

    m_pcd.WorkPoints.AddFixed(pt1)
    m_pcd.WorkPoints.AddFixed(pt2)

End Sub
```

Helper Method

Use the web!

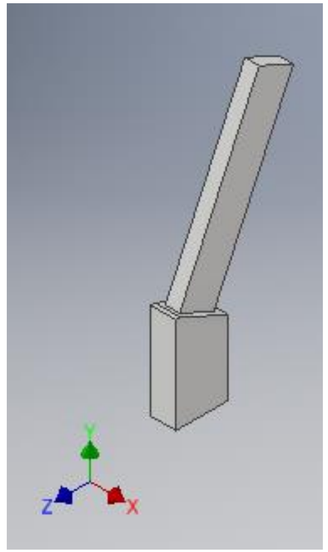
```
'Return an arbitrary vector perpendicular to 'v'
'Based on:  https://codereview.stackexchange.com/questions/43928/algorithm-to-get-an-arbitrary-perpendicular-vector
1 reference
Public Function Perpendicular(v As Vector) As Vector

    ' x = y = z = 0 Is Not an acceptable solution
    If ((v.X = v.Y) And (v.Y = v.Z) And (v.Z = 0)) Then
        Throw New System.Exception("zero-vector")
    End If

    'If one Then dimension Is zero, this can be solved by setting that To
    'non-zero And the others To zero. Example: (4, 2, 0) lies In the
    'x-y-Plane, so (0, 0, 1) Is orthogonal To the plane.

    If v.X = 0 Then
        Return m_tg.CreateVector(1, 0, 0)
    ElseIf v.Y = 0 Then
        Return m_tg.CreateVector(0, 1, 0)
    ElseIf v.Z = 0 Then
        Return m_tg.CreateVector(0, 0, 1)
    Else
        ' arbitrarily Set a = b = 1
        ' Then the equation simplifies To
        '     c = -(x + y)/z
        Return m_tg.CreateVector(1, 1, -1.0 * (v.X + v.Y) / v.Z)
    End If

End Function
```

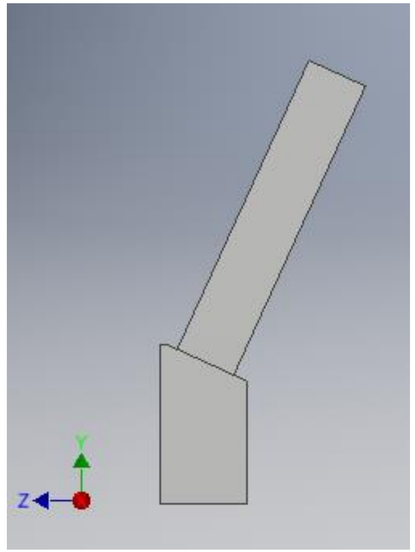
Maybe should have been Private?

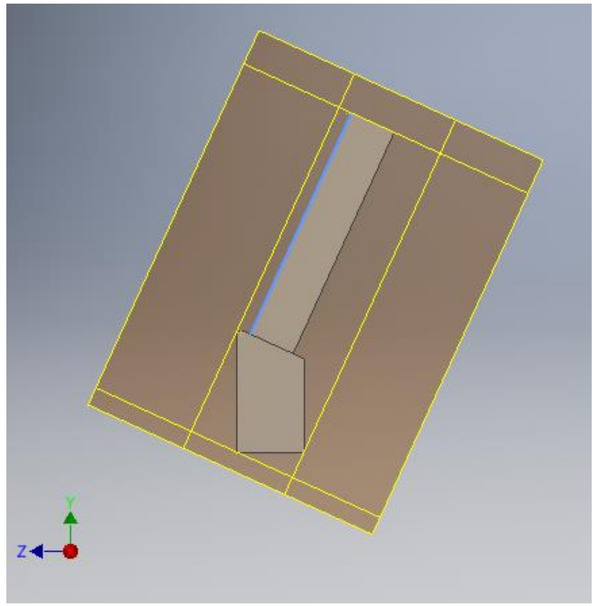
Uses member variable

Enough complexity to want a debugger!

Works!

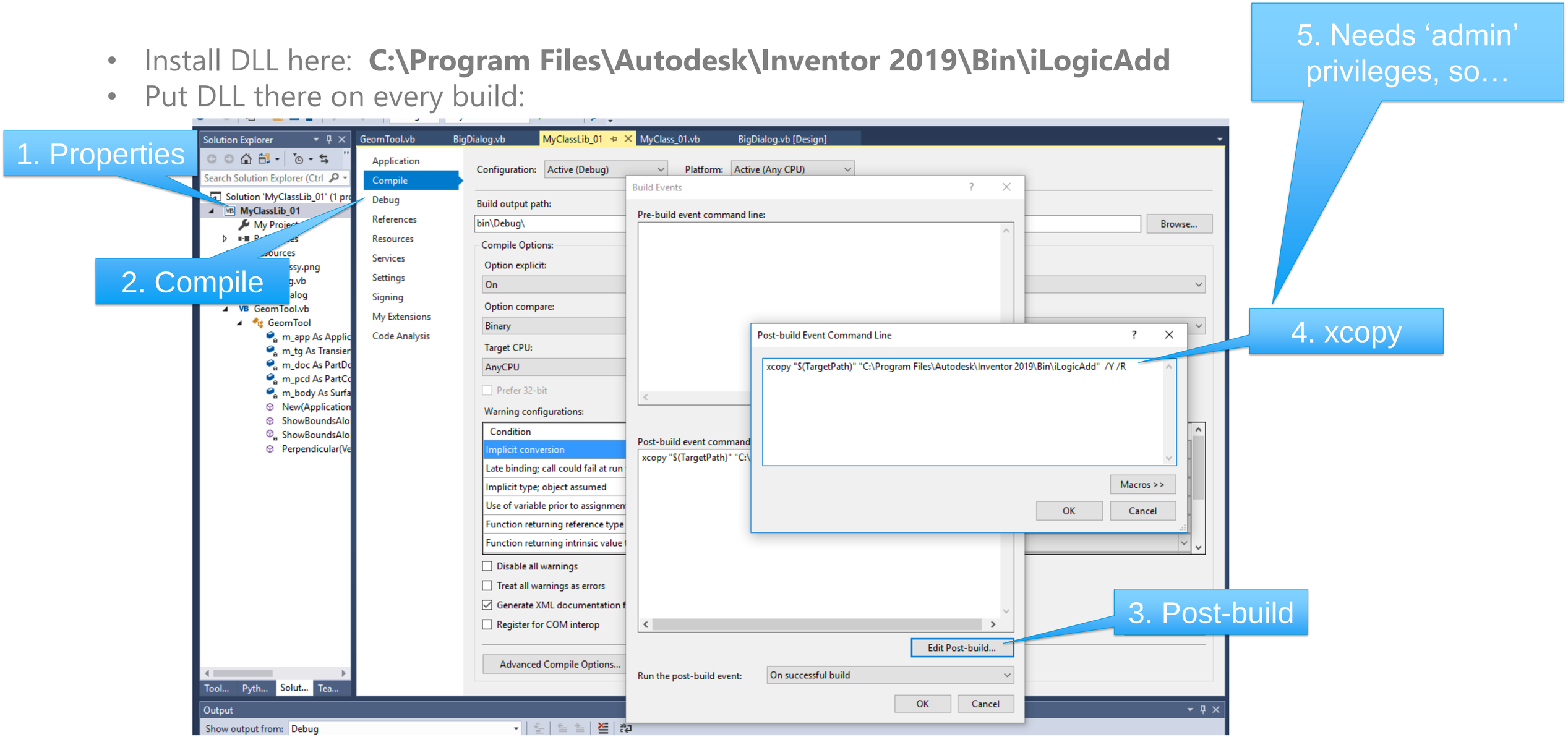






Key Details: DLL Location

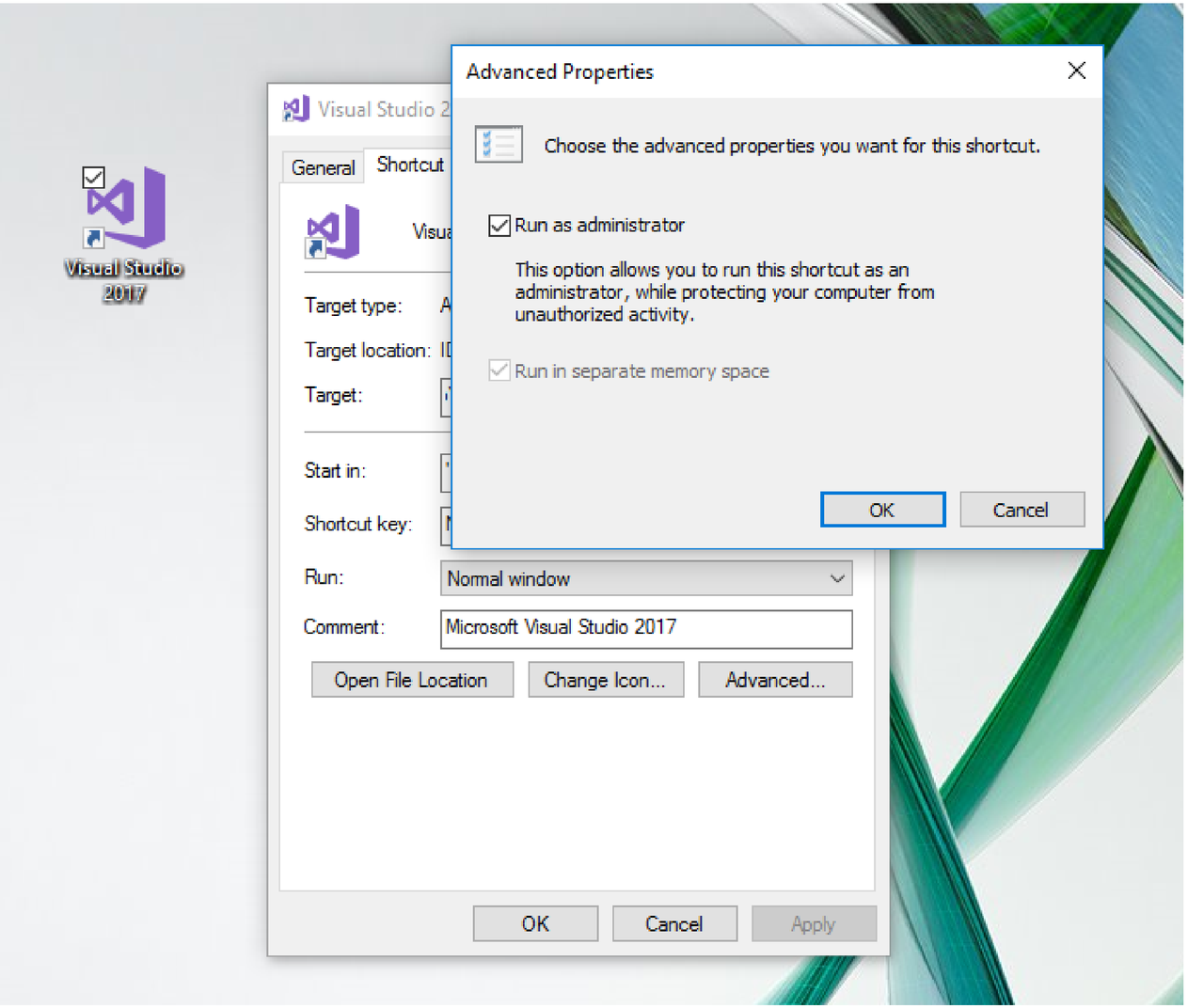
- Install DLL here: **C:\Program Files\Autodesk\Inventor 2019\Bin\iLogicAdd**
- Put DLL there on every build:



Key Details: Admin Shortcut

- Create shortcut to VS on desktop
- Properties…
- Advanced…
- Select “Run as administrator”
- Always run VS from this shortcut…

- …then the post-build xcopy will work.

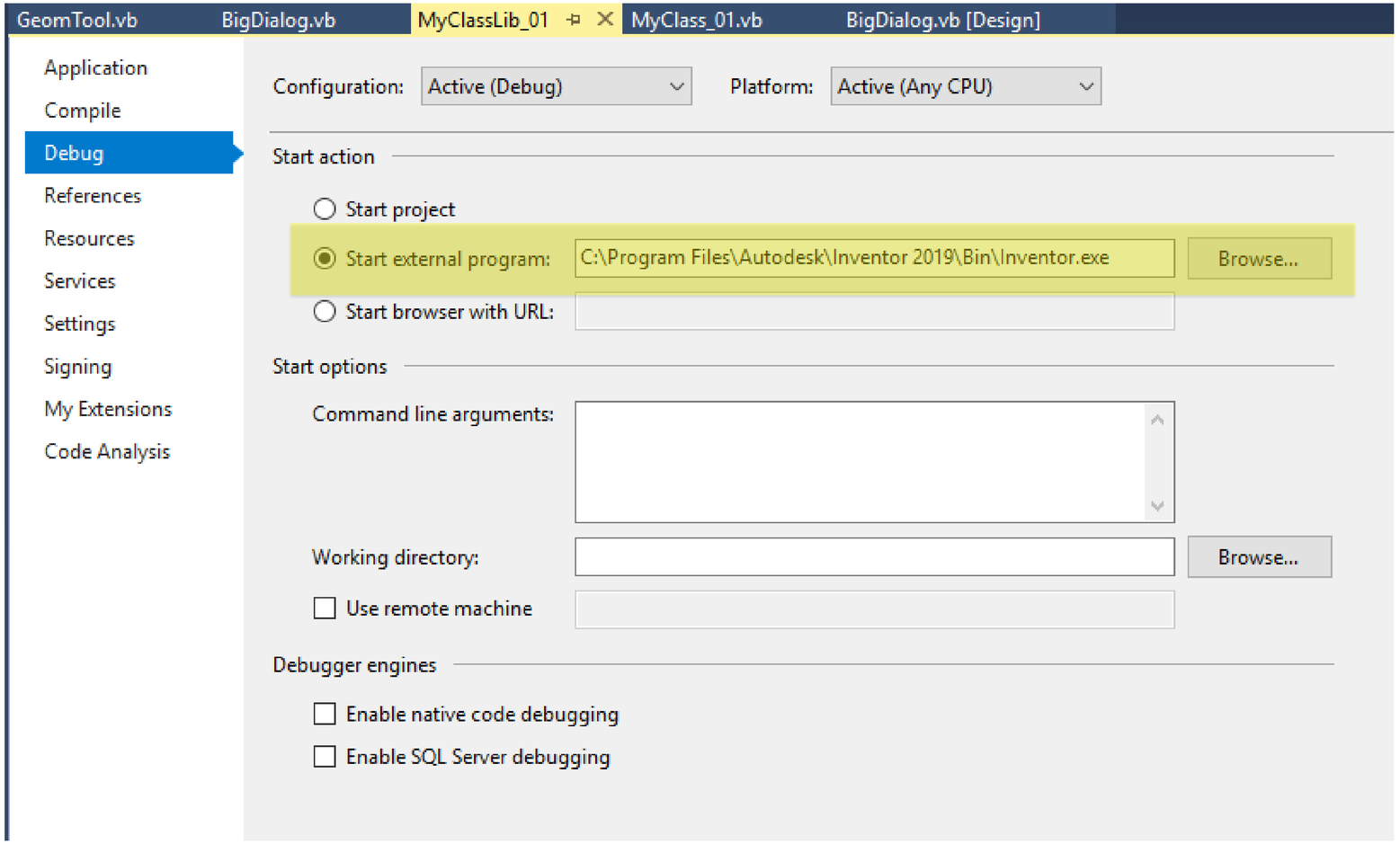


Key Details: Startup App

- Project Properties…
- Debug tab…
- Start action…
- Select “Start external program”
- Browse to “inventor.exe” in the Program Files folder.

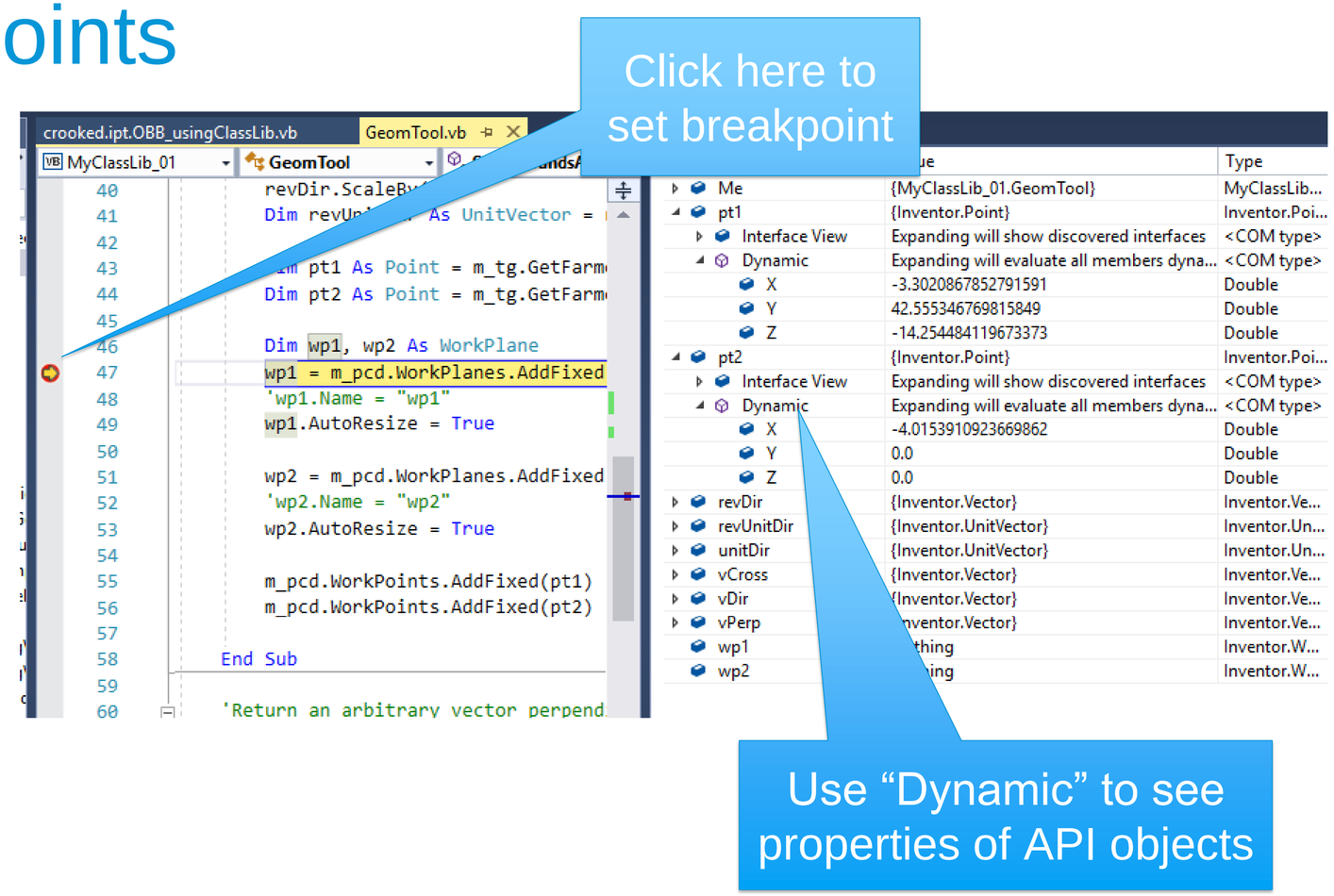
Then pressing “F5” in VS will:

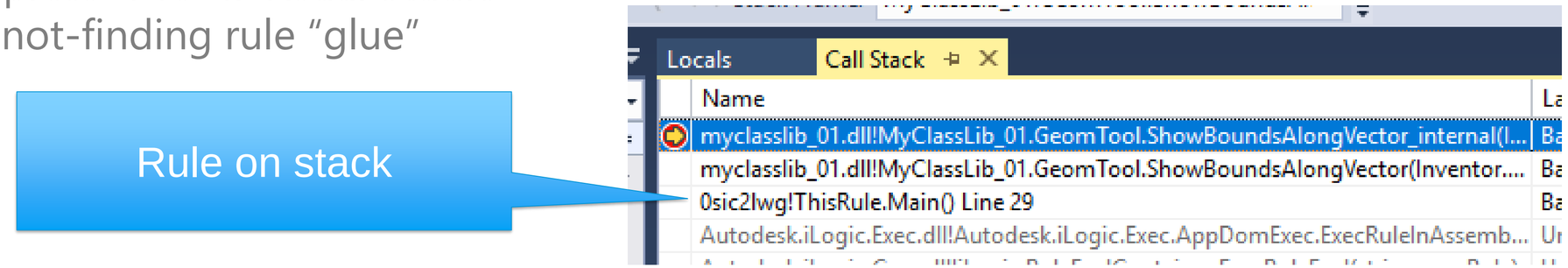
- Build/rebuild as necessary
- “Install” DLL to required folder
- start Inventor
- Be ready for breakpoints/exceptions



Side Note: Setting Breakpoints

- You can set breakpoints in your class library code, as normal for Visual Studio
- When you hit the breakpoint, you'll find the code for your iLogic rule on the stack too! You can step through this code.
- When working in Visual Studio like this, you can even insert a "break" [sic] statement in an iLogic rule, to cause the debugger to stop (unconditionally) at that point.
- You may have to press "F10" if Visual Studio complains about not-finding rule "glue" code.





Side Note: Setting Breakpoints in Rules

We're in Visual Studio here!

iLogic Rule code!

Stepping through iLogic code

Set explicit breakpoints

Misc minor differences

Cannot edit iLogic code here, read-only

crooked.ipt.OBB_usingClassLib.vb | GeomTool.vb | MyClassLib_01 | gDialog.vb [Design]

Miscellaneous Files | ThisRule | Main

```
19      'Assume it's a linear edge for now
20      Dim edge As edge = ss(1)
21      If (edge Is Nothing) Then
22          MessageBox.Show("Please select a linear edge first")
23          Exit Sub
24      Else
25          break
26          Dim vec As UnitVector = edge.Geometry.direction
27
28          Dim geom As New MyClassLib_01.GeomTool(ThisApplication)
29          geom.ShowBoundsAlongVector(ThisDoc.Document, vec.AsVector)
30
31          End If
32  End If
33
34
35
36  End Sub ''**iLogic system**
37  End Class ''**iLogic system**
38
```

Locals | Call Stack

Name

myclasslib_01.dll!MyClassLib_01.GeomToc
myclasslib_01.dll!MyClassLib_01.GeomToc
0sic2lwg!ThisRule.Main() Line 29
Autodesk.iLogic.Exec.dll!Autodesk.iLogic.
Autodesk.iLogic.Core.dll!iLogic.RuleEvalC
Autodesk.iLogic.Core.dll!iLogic.RuleEvalC
System.Windows.Forms.dll!System.Windo
System.Windows.Forms.dll!System.Windo
System.Windows.Forms.dll!System.Windo
System.Windows.Forms.dll!System.Windo
System.Windows.Forms.dll!System.Windo
System.Windows.Forms.dll!System.Windo
System.Windows.Forms.dll!System.Windo
System.Windows.Forms.dll!System.Windo
System.Windows.Forms.dll!System.Windo

- Note: Cannot set "ordinary" breakpoints in iLogic rules until iLogic code is loaded into VS (by an explicit "break" or breakpoint in Class Library code or unhandled exception)

Summary: Class Libraries

EASY WAY TO GET INTO PURE PROGRAMMING

Very little “special plumbing”

TAKE ADVANTAGE OF VISUAL STUDIO, SOURCE CONTROL

Breakpoints, better control over files of code

1. CUSTOM DIALOGS

Easy to build and use, more powerful than iLogic forms

2. PURE INVENTOR MODEL API

Easy to use VS to debug, while invoking from iLogic rule.

Jump to addins if you’re using Inventor API “UI functions”

Leap to Addins

What is an Addin

A SOFTWARE PACKAGE THAT RUNS INSIDE INVENTOR AND ADDS ADDITIONAL FUNCTIONALITY.

You can write your own addins, for specialized functionality.

You can use addins written by other people.

Some of Inventor’s functionality is delivered in the form of addins.

A DLL THAT FOLLOWS CERTAIN PROTOCOLS

These protocols allow Inventor to be aware of the addin, in contrast to class libraries.

Like class libraries, addins are written in VB.Net, C# and C++.

BIG

Generally, an addin is bigger and more comprehensive than an iLogic rule.

You can have lots of functionality in a single addin.

You might only ever write ONE addin!

What can an Addin Do?

EXTEND THE INVENTOR UI

Addins typically define new Inventor commands, and/or other extensions to the Inventor UI (e.g., browser panes, event-handlers, etc.)

MANAGE AND MODIFY INVENTOR FILES

Parts, assemblies, drawings

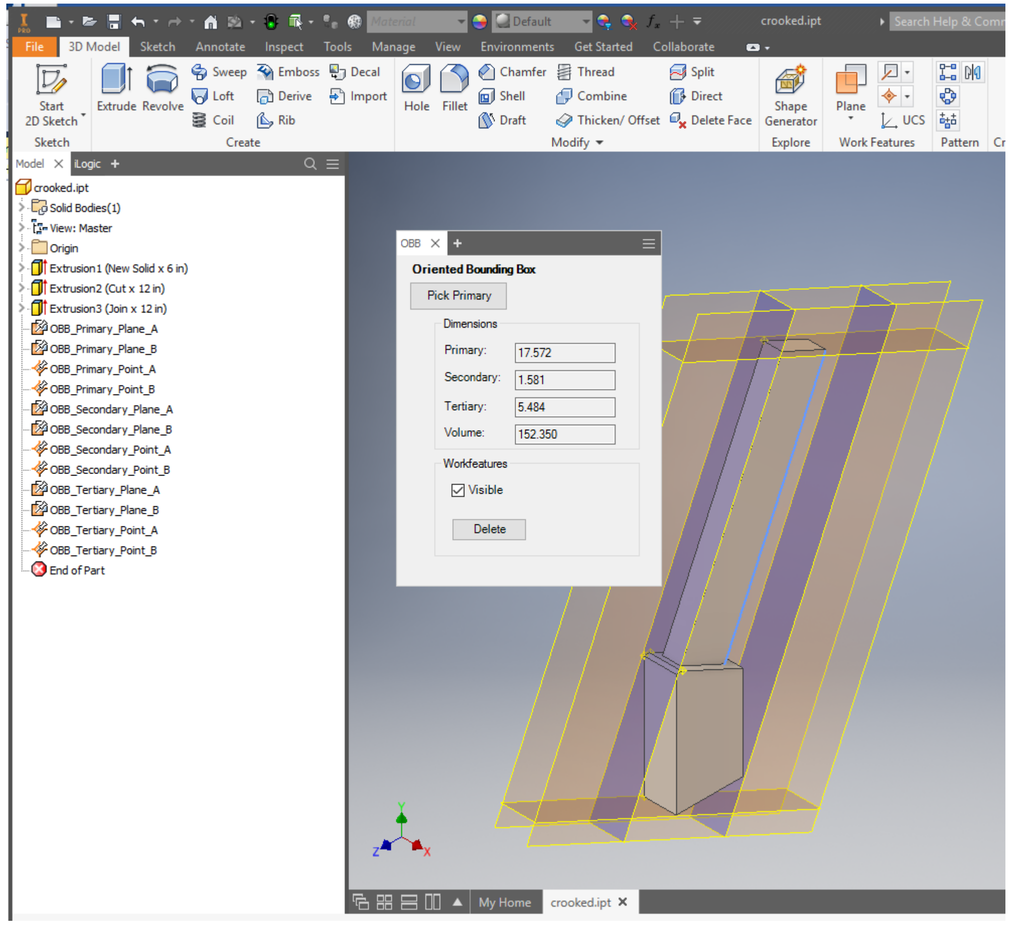
Add new items (sketches and features, components and constraints, sheets and views)

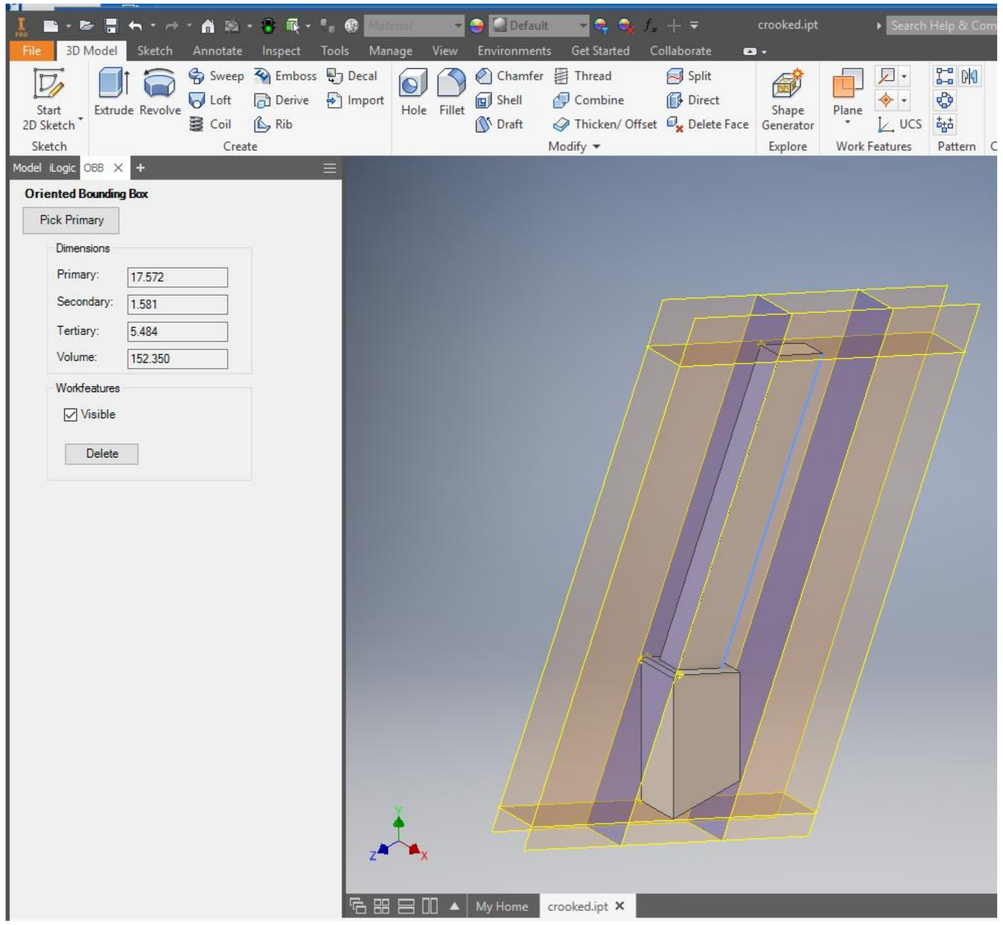
Modify existing items

USE THE INVENTOR API

The entire Inventor API is at your disposal. Addins are what the API was made for.

Example Addin – “OBB Dockable Window”

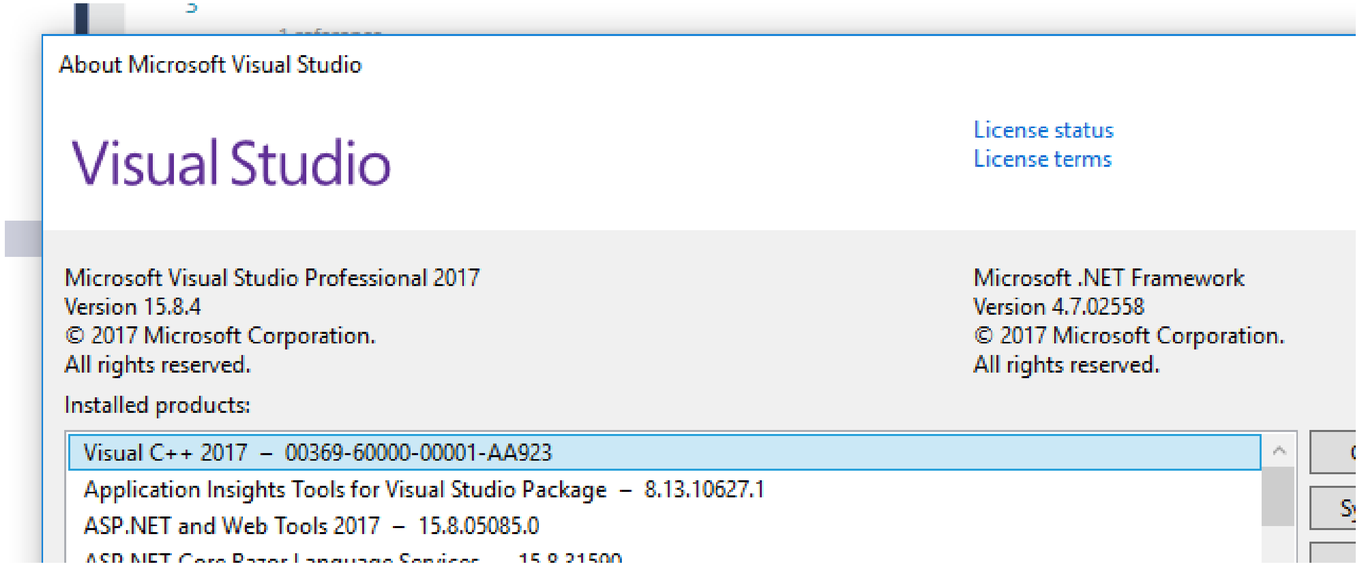




Adding a new capability to the Inventor user experience

Visual Studio

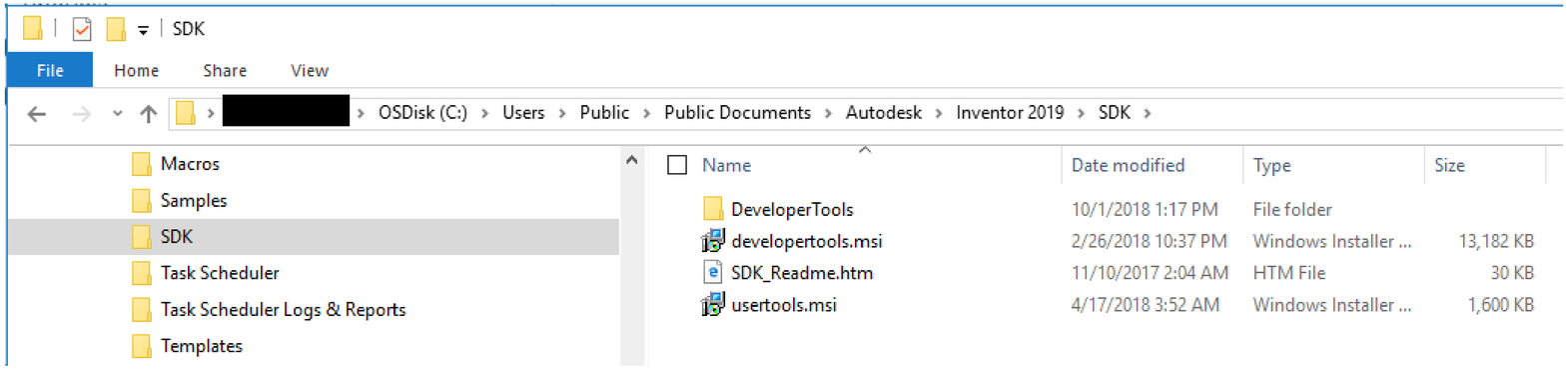
- You need Visual Studio to write an addin
- Very similar to what we saw in Class Libraries
- VB.Net, C# or C++



The Inventor SDK

- Secondary install process from:
  C:\Users\Public\Documents\Autodesk\Inventor 2019\SDK

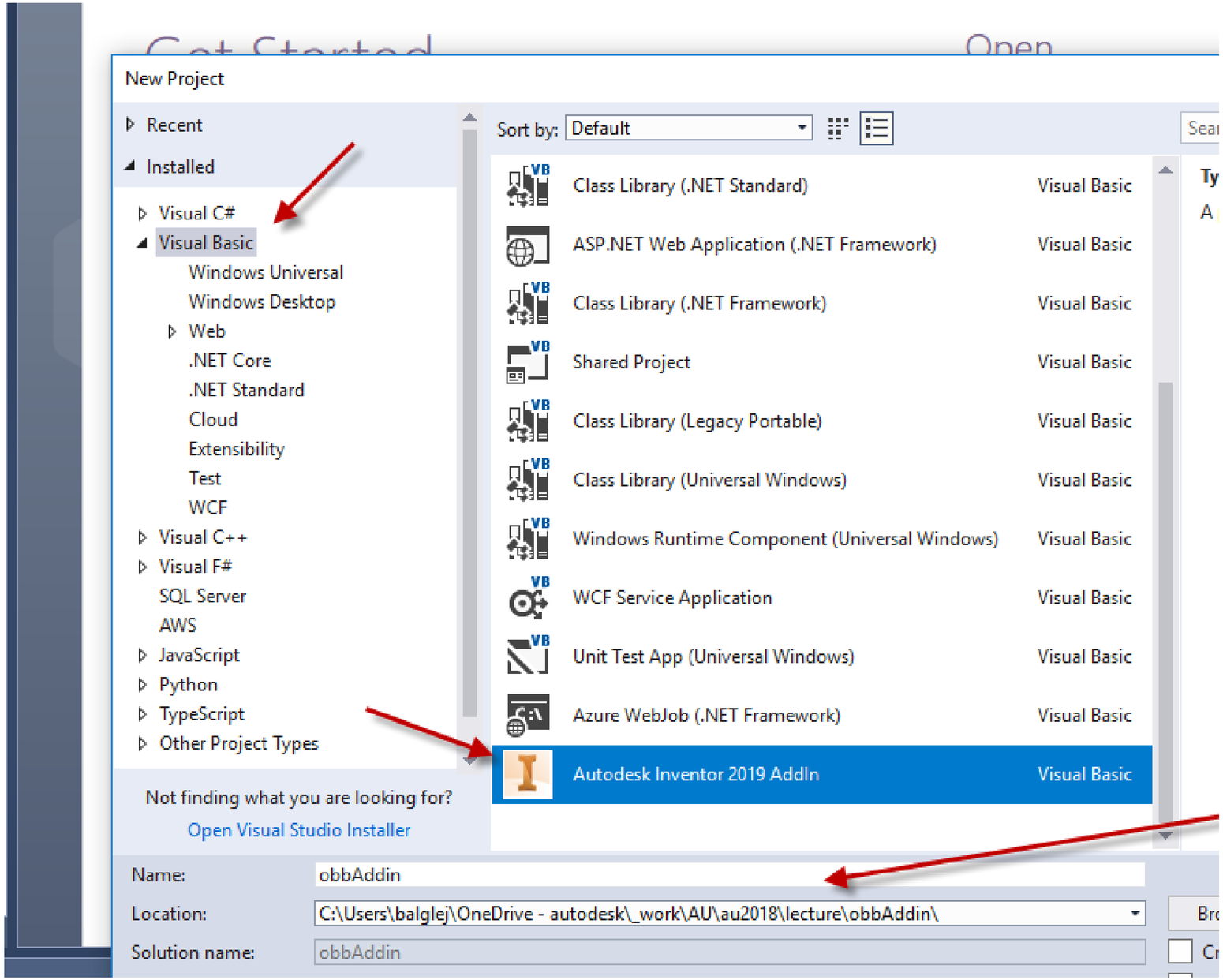
- Adds new project-type to VS
- Includes doc and examples



Start New Project

Similar to Class Library:

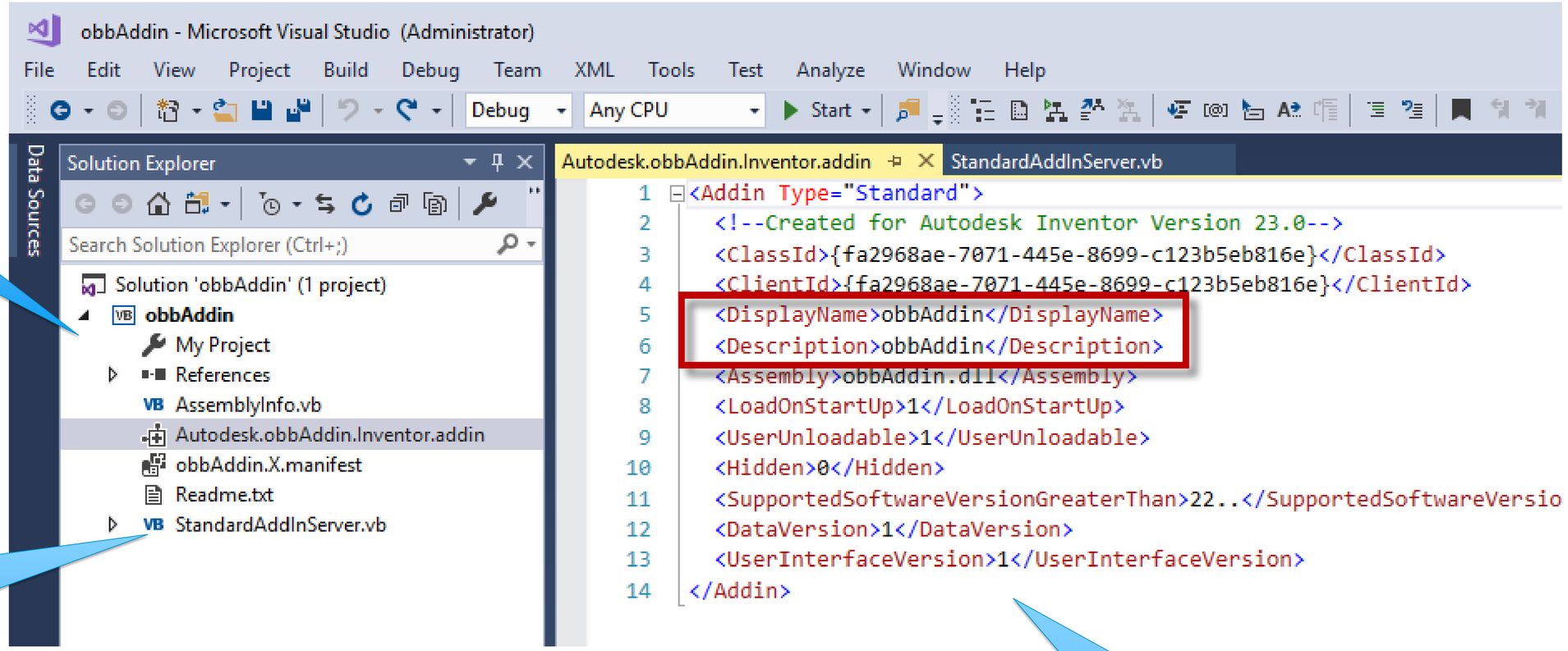
- Start new “Addin” project in VS
- Xcopy in post-build step
- Start-up program
- Run shortcut as admin



Initial contents – from SDK

SDK template creates a working project

Addin Server (next)



.Addin file

Initial contents con’t.

Solution Explorer

Search Solution Explorer (Ctrl+;)

- Solution 'obbAddin' (1 project)
  - obbAddin
    - My Project
    - References
    - AssemblyInfo.vb
    - Autodesk.obbAddin.Inventor.addin
    - obbAddin.X.manifest
    - Readme.txt
    - StandardAddInServer.vb

Autodesk.obbAddin.Inventor.addin | StandardAddInServer.vb

obbAddin | StandardAddInServer | ExecuteCommand

```vb
Imports Inventor
Imports System.Runtime.InteropServices
Imports Microsoft.Win32

Namespace obbAddin
    <ProgIdAttribute("obbAddin.StandardAddInServer"), _
    GuidAttribute("fa2968ae-7071-445e-8699-c123b5eb816e")> _
    1 reference
    Public Class StandardAddInServer
        Implements Inventor.ApplicationAddInServer

        Private WithEvents m_uiEvents As UserInterfaceEvents
        'Private WithEvents m_sampleButton As ButtonDefinition

#Region "ApplicationAddInServer Members"

        ' This method is called by Inventor when it loads the AddIn. The AddInSiteObject provides access
        ' to the Inventor Application object. The FirstTime flag indicates if the AddIn is loaded for
        ' the first time. However, with the introduction of the ribbon this argument is always true.
        0 references
        Public Sub Activate(ByVal addInSiteObject As Inventor.ApplicationAddInSite, ByVal firstTime As Boolean) Implements I

        ' This method is called by Inventor when the AddIn is unloaded. The AddIn will be
        ' unloaded either manually by the user or when the Inventor session is terminated.
        0 references
        Public Sub Deactivate() Implements Inventor.ApplicationAddInServer.Deactivate ...

        ' This property is provided to allow the AddIn to expose an API of its own to other
        ' programs. Typically, this  would be done by implementing the AddIn's API
        ' interface in a class and returning that class object through this property.
        0 references
        Public ReadOnly Property Automation() As Object Implements Inventor.ApplicationAddInServer.Autom on ...

        ' Note:this method is now obsolete, you should use the
        ' ControlDefinition functionality for implementing commands.
        0 references
        Public Sub ExecuteCommand(ByVal commandID As Integer) Implements Inventor.ApplicationAddInS ExecuteCommand
        End Sub
```

Addin Server class

Key method: “Activate”

Key method: “Automation”

Automatic assignment of GUID

```vb
<ProgIdAttribute("obbAddin.StandardAddInServer"), _
GuidAttribute("fa2968ae-7071-445e-8699-c123b5eb816e")> _
1 reference
Public Class StandardAddInServer
    Implements Inventor.ApplicationAddInServer
```

Initial contents

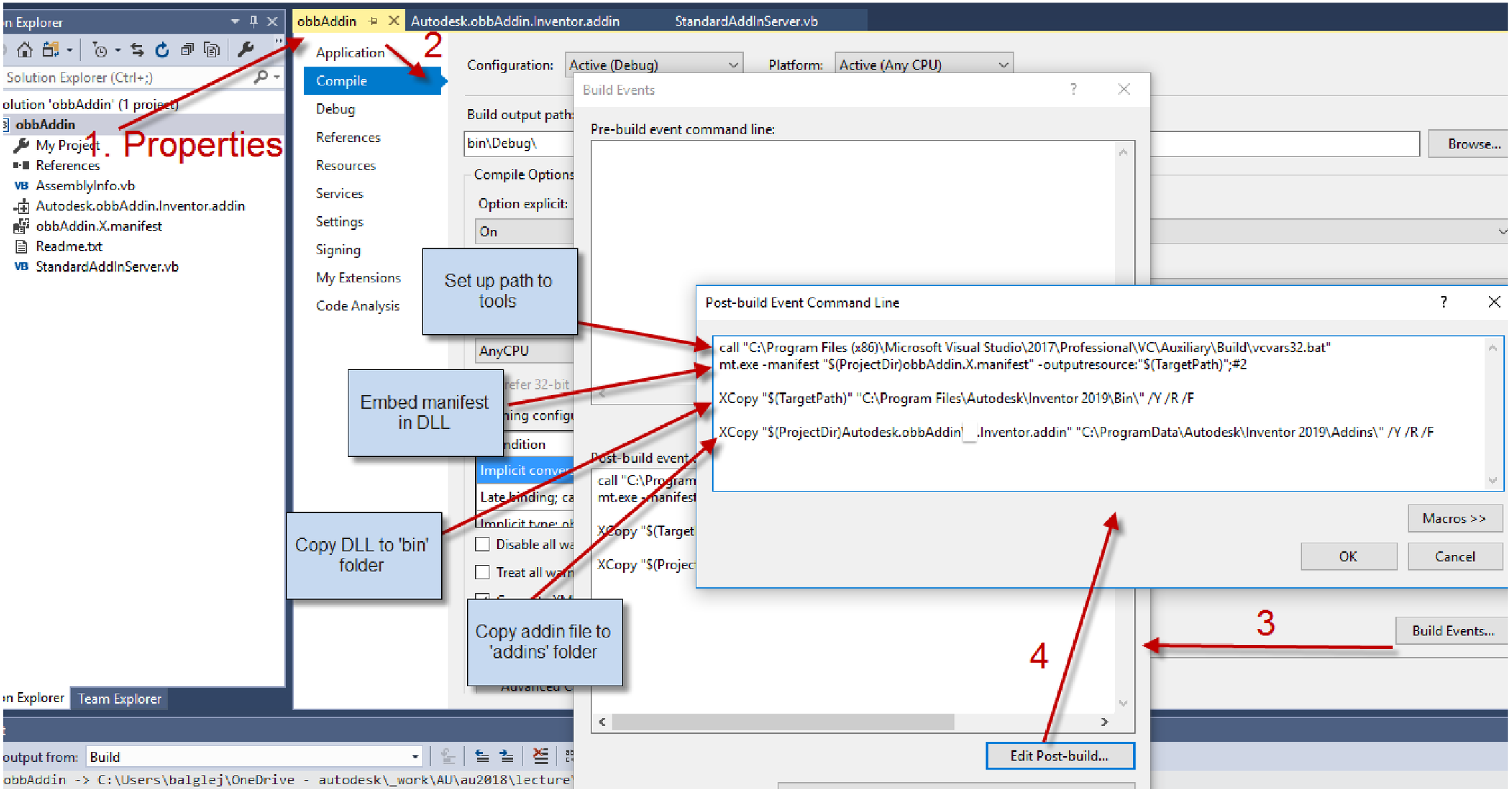
```
Public Sub Activate(ByVal addInSiteObject As Inventor.ApplicationAddInSite, ByVal firstTime As Boolean) Implements I
    ' Initialize AddIn members.
    g_inventorApplication = addInSiteObject.Application

    ' Connect to the user-interface events to handle a ribbon reset.
    m_uiEvents = g_inventorApplication.UserInterfaceManager.UserInterfaceEvents

    ' TODO: Add button definitions.

    ' Sample to illustrate creating a button definition.
    'Dim largeIcon As stdole.IPictureDisp = PictureDispConverter.ToIPictureDisp(My.Resources.YourBigImage)
    'Dim smallIcon As stdole.IPictureDisp = PictureDispConverter.ToIPictureDisp(My.Resources.YourSmallImage)
    'Dim controlDefs As Inventor.ControlDefinitions = g_inventorApplication.CommandManager.ControlDefinitions
    'm_sampleButton = controlDefs.AddButtonDefinition("Command Name", "Internal Name", CommandTypesEnum.kShapeEditCm

    ' Add to the user interface, if it's the first time.
    If firstTime Then
        AddToUserInterface()
    End If
End Sub
```

- Addin server class
- Un-comment to test

```
#Region "User interface definition"
        ' Sub where the user-interface creation is done.  This is called when
        ' the add-in loaded and also if the user interface is reset.
        2 references
        Private Sub AddToUserInterface()
            ' This is where you'll add code to add buttons to the ribbon.

            '** Sample to illustrate creating a button on a new panel of the Tools tab of the Part ribbon.

            '' Get the part ribbon.
            'Dim partRibbon As Ribbon = g_inventorApplication.UserInterfaceManager.Ribbons.Item("Part")

            '' Get the "Tools" tab.
            'Dim toolsTab As RibbonTab = partRibbon.RibbonTabs.Item("id_TabTools")

            '' Create a new panel.
            'Dim customPanel As RibbonPanel = toolsTab.RibbonPanels.Add("Sample", "MysSample", AddInClientID)

            '' Add a button.
            'customPanel.CommandControls.AddButton(m_sampleButton)
        End Sub

        0 references
        Private Sub m_uiEvents_OnResetRibbonInterface(Context As NameValueMap) Handles m_uiEvents.OnResetRibbonInterface
            ' The ribbon was reset, so add back the add-ins user-interface.
            AddToUserInterface()
        End Sub

        ' Sample handler for the button.
        'Private Sub m_sampleButton_OnExecute(Context As NameValueMap) Handles m_sampleButton.OnExecute
        '    MsgBox("Button was clicked.")
        'End Sub
#End Region
```

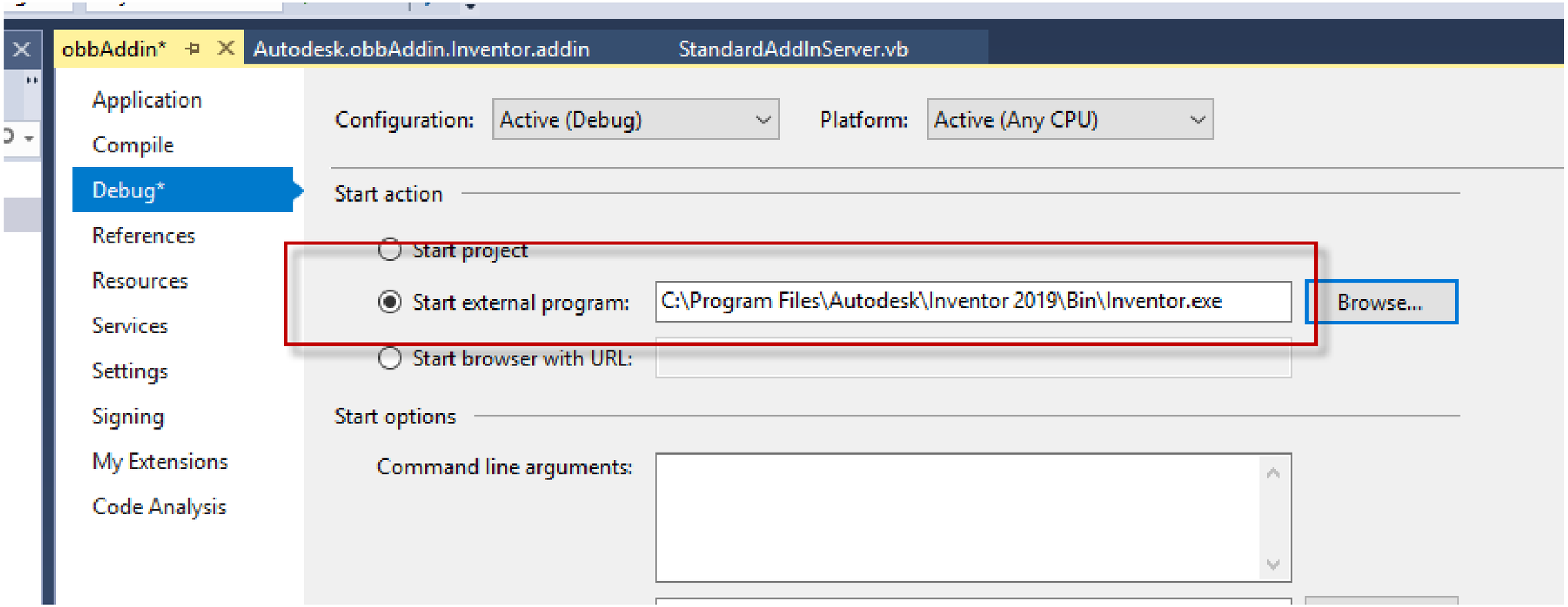
Set up build events

- A little different from class libraries



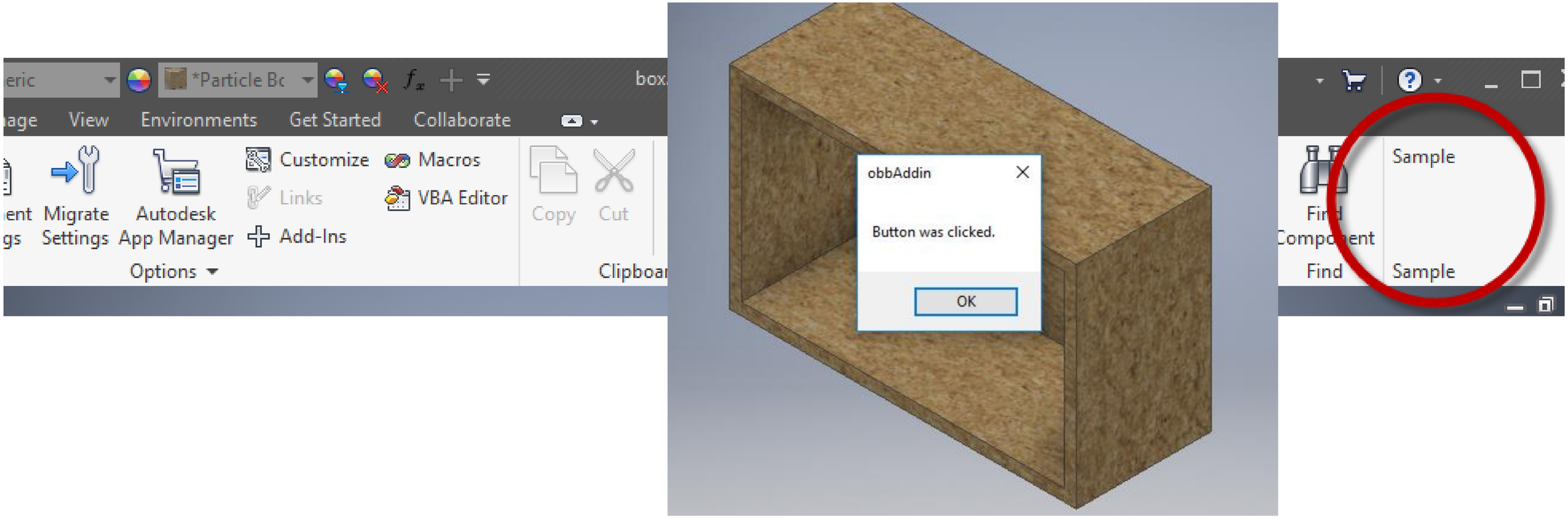
Run Inventor as Startup App

- Same as class libraries



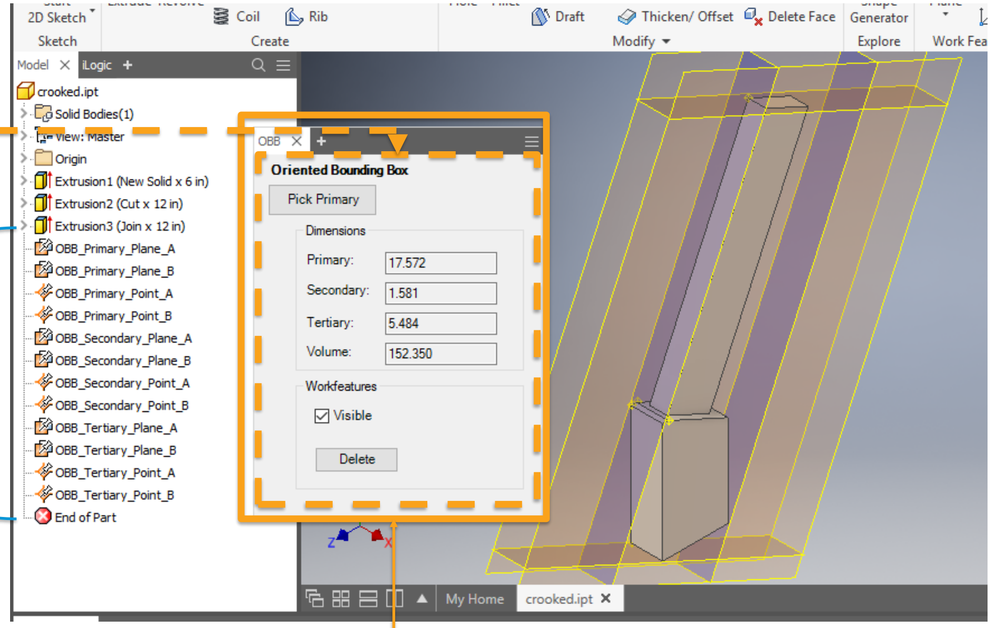
Run the Project

- Command appears in ribbon
- It works!



Plan for OBB Addin

- (Remove all sample code)
- “ObbUtility” class:
  - “attaches” to active document
  - Manages OBB workfeatures
- “ObbForm” class
  - Entry point for user to pick edge
  - Shows dimension measurements
  - Visibility/Delete buttons
  - Calls into ObbUtility
- Addin server class
  - Creates dockable window with form
  - Sets up “active document” linkage



Addin Server Class

- Created automatically by template
- Defines GUID
- Sets things up

Additional members

Additional members

Important member (from SDK)

Additional members

```
Namespace obbAddin
    <ProgIdAttribute("obbAddin.StandardAddInServer"), _
    GuidAttribute("fa2968ae-7071-445e-8699-c123b5eb816e")> _
    1 reference
    Public Class StandardAddInServer
        Implements Inventor.ApplicationAddInServer

        Private WithEvents m_uiEvents As UserInterfaceEvents
        Private WithEvents m_appEvents As ApplicationEvents

        Private m_window As DockableWindow = Nothing
        Private m_form As ObbForm = Nothing

#Region "ApplicationAddInServer Members"

        ' This method is called by Inventor when it loads the AddIn. The AddInSiteObject provides access
        ' to the Inventor Application object. The FirstTime flag indicates if the AddIn is loaded for
        ' the first time. However, with the introduction of the ribbon this argument is always true.
        0 references
        Public Sub Activate(ByVal addInSiteObject As Inventor.ApplicationAddInSite, ByVal firstTime As Bool
            ' Initialize AddIn members.
            g_inventorApplication = addInSiteObject.Application

            ' Connect to the user-interface events to handle a ribbon reset.
            m_uiEvents = g_inventorApplication.UserInterfaceManager.UserInterfaceEvents
            m_appEvents = g_inventorApplication.ApplicationEvents

        End Sub

        ' This method is called by Inventor when the AddIn is unloaded. The AddIn will be
```

OnActivateDocument event

- Called when a document is activated
- Shows OBB window when IPT is activated (always! too aggressive!)

```
'Use this to show the dockable window only for Part documents.  Hide it for
0 references
Private Sub m_appEvents_OnActivateDocument(doc As Document, timing As EventT
    If (timing = EventTimingEnum.kBefore) Then
        Dim pDoc As PartDocument = Nothing
        Try
            pDoc = doc
        Catch ex As Exception

        End Try

        Dim isPartDoc As Boolean = (pDoc IsNot Nothing)
        If ((Not isPartDoc) And (m_window IsNot Nothing)) Then
            m_window.Visible = False
        ElseIf (isPartDoc) Then
            'Window is created when first part doc is activated.
            CreateObbWindow(pDoc)
            m_window.Visible = True
        End If
    End If

End Sub
```

Take action when a document is newly activated

OnActivateDocument event, continued

- Very specific to "dockable windows"

UI elements sometimes need addin "ID"

Create dockable window (once only)

Create form

Attach form to window

Connect form with active document (every time)

```
64      Private Sub CreateObbWindow(pDoc As PartDocument)
65
66          If m_window Is Nothing Then
67
68              Dim oUserInterfaceMgr As UserInterfaceManager
69              oUserInterfaceMgr = g_inventorApplication.UserInterfaceManager
70
              ' Create a new dockable window
72              m_window = oUserInterfaceMgr.DockableWindows.Add(AddInClientID(), "SomeInternalName", "OBB")
73
74              m_form = New ObbForm(g_inventorApplication)
75              'frm.TopMost = True  (Can do this in the designer)
                m_form.Show()
7
8               ' Get the hwnd of the dialog to be added as a child
79              Dim hwnd As Long
0               hwnd = m_form.Handle
1
2               ' Add the dialog as a child to the dockable window
83              m_window.AddChild(hwnd)
84
85              m_window.SetMinimumSize(350, 260)
86
87              ' Don't allow docking to top and bottom
88              m_window.DisabledDockingStates = DockingStateEnum.kDockTop + DockingStateEnum.kDockBottom
90          End If
91
92          m_form.Attach(pDoc)
93      End Sub
```

Addin Server, Automation property

- When your addin exposes its own API

- Not used in this example

```
44
45      ' This property is provided to allow the AddIn to expose an API of its own to other
46      ' programs. Typically, this  would be done by implementing the AddIn's API
47      ' interface in a class and returning that class object through this property.
        0 references
48      Public ReadOnly Property Automation() As Object Implements Inventor.ApplicationAddInServer.Automation
49          Get
50              Return Nothing
51          End Get
52      End Property
```

ObbUtility Class

- Add Item … Class … →
- Manages the workfeatures
- “Show…” function returns measurements

bForm.vb | ObbUtility.vb* | obbAddin | Autodesk.obbAddin.Inventor.addin | StandardAddInServer.vb | ObbForm.vb [Des

obbAddin | ObbUtility

```
1    Imports Inventor
2
     3 references
3    Public Class ObbUtility
4
5    Members
16
17       'Creates a new instance and initializes some things.  Could have used the "singleton pattern"
18       'only ever create one instance.
         1 reference
19       Public Sub New(app As Application) ...
26
27       'Called by event-handler to inform "me" that the active document has changed.  Returns
28       'sizes of the obb-dimensions, so UI can be updated.
         1 reference
29       Public Function Attach(pdoc As PartDocument) As Double() ...
34
35       ' Determines whether the document has OBB workfeatures.  Assumes that if one is there,
36       'they're all there (not safe, really)
         0 references
37       Public ReadOnly Property HasObb As Boolean ...
47
48       'Sets the visibility of ALL OBB-workfeatures.  Doesn't have a "Get".
         1 reference
49       Public WriteOnly Property Visibility As Boolean ...
61
62       'Main method that actually creates the workfeatures.
         1 reference
63       Public Function ShowBoundsAlongVector(doc As PartDocument, vDir As Vector) As Double() ...
85
         2 references
86       Public Sub DeleteWorkfeatures() ...
104
```

ObbUtility.Attach

```
26
27      'Called by event-handler to inform "me" that the active document has changed.  Returns
28      'sizes of the obb-dimensions, so UI can be updated.
        1 reference
29      Public Function Attach(pdoc As PartDocument) As Double()
30          m_doc = pdoc
31          m_pcd = m_doc.ComponentDefinition
32          Return obbDims
33      End Function
```

- Note: iLogic doesn’t have an “OnActivateDocument” event,
  so this is only feasible for addins.

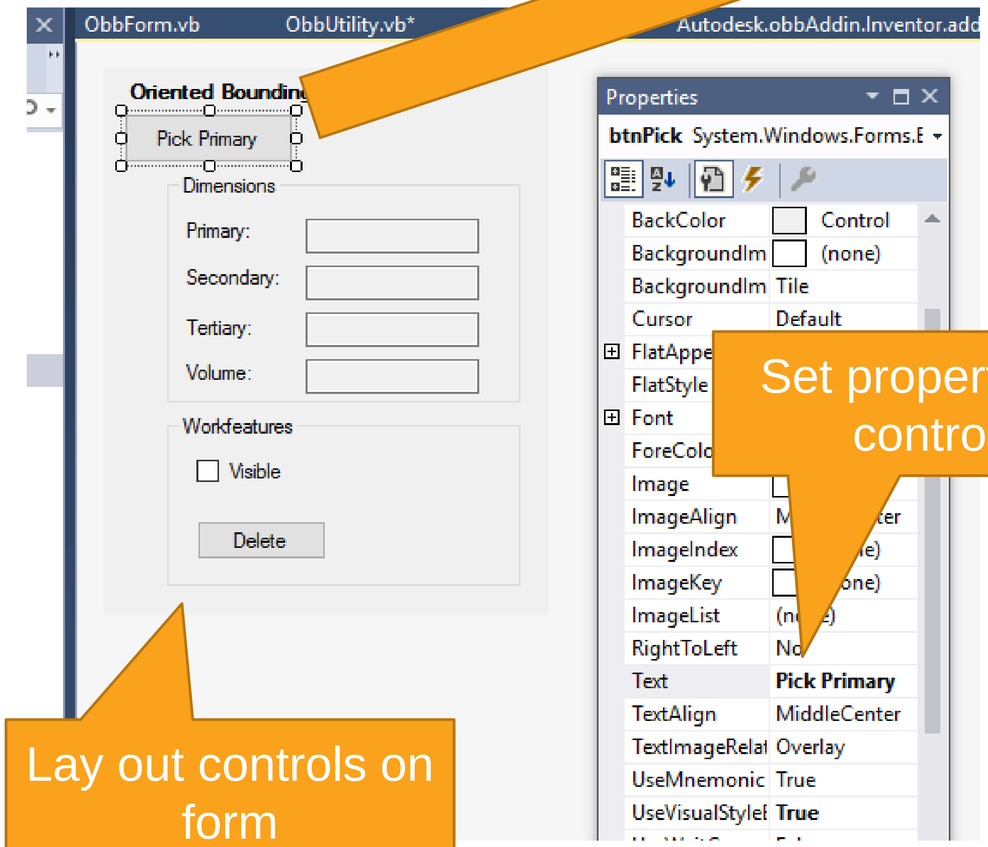
ObbUtility.ShowBoundsAlongVector

- Similar to method in “class library” example
- Uses “helper” function 3x to create workfeatures
- Returns array of three doubles (measurements)

```
'Main method that actually creates the workfeatures.
1 reference
Public Function ShowBoundsAlongVector(doc As PartDocument, vDir As Vector) As Double()

    m_doc = doc
    m_pcd = m_doc.ComponentDefinition
    Dim bodies As SurfaceBodies = m_pcd.SurfaceBodies
    m_body = bodies(1)

    Dim vPerp As Vector = Perpendicular(vDir)
    Dim vCross As Vector
    vCross = vDir.CrossProduct(vPerp)

    DeleteWorkfeatures()

    Dim result(3) As Double

    result(0) = ShowBoundsAlongVector_internal(doc, vDir, vPerp, vCross, 0)
    result(1) = ShowBoundsAlongVector_internal(doc, vPerp, vCross, vDir, 1)
    result(2) = ShowBoundsAlongVector_internal(doc, vCross, vDir, vPerp, 2)

    Return result

End Function
```

ObbForm Class

- Manages the UI
- (This example uses ancient VB “forms” for simplicity)
- Uses Inventor API

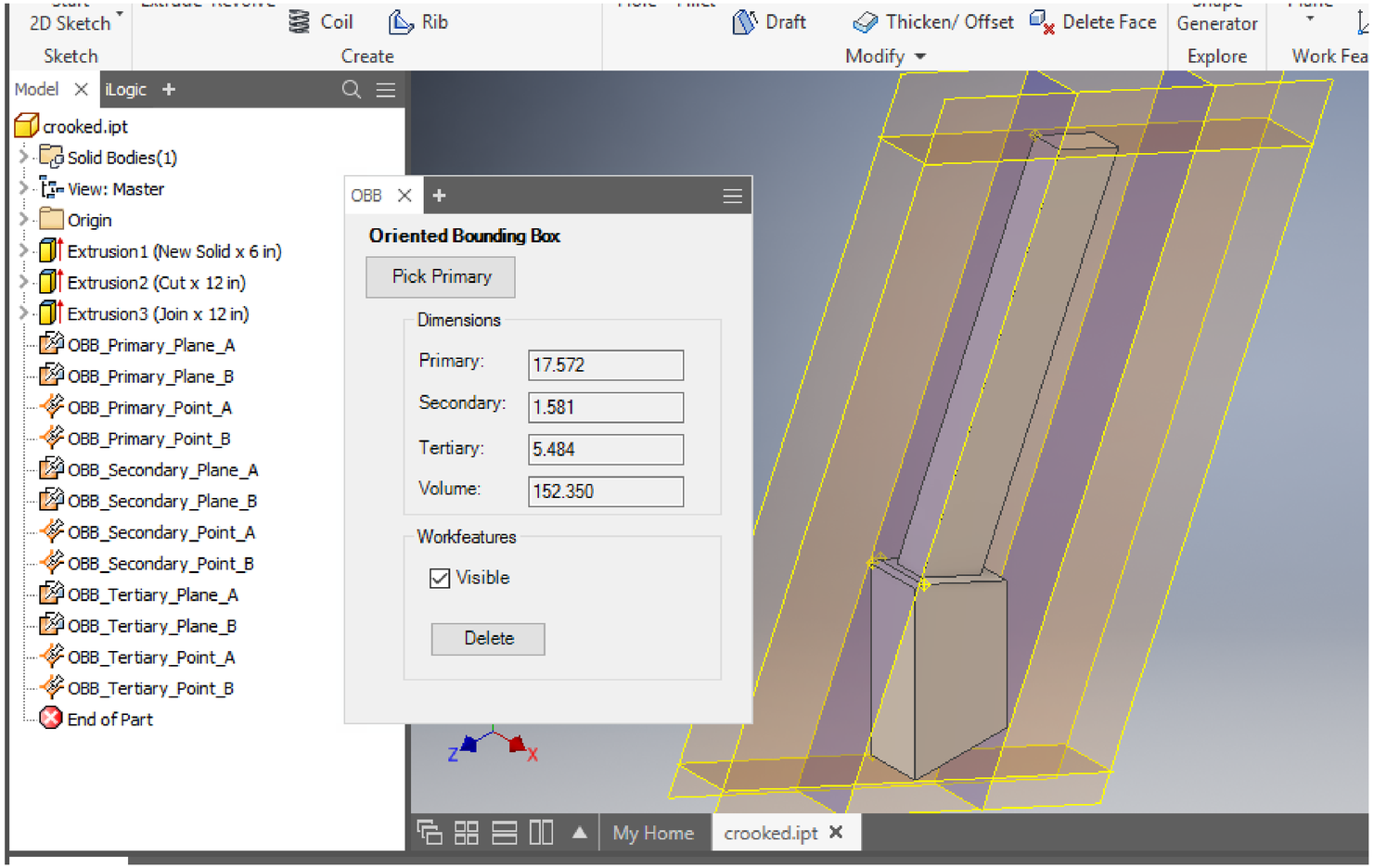


Set properties of controls

Lay out controls on form

```
0 references
Private Sub btnPick_Click(sender As Object, e As EventArgs) Handles btnPick.C
    txt_1.Select() 'Make the button lose focus

    Dim edge As Edge = Nothing

    'Check for pre-selection
    Dim ss = m_doc.SelectSet
    If (ss.Count = 1) Then
        edge = TryCast(ss(1), Edge)
    End If

    Dim l As LineSegment = Nothing
    If (edge IsNot Nothing) Then
        l = TryCast(edge.Geometry, LineSegment)
    End If

    'No valid pre-selection, let user pick
    If (edge Is Nothing) Or (l Is Nothing) Then
        Dim cmdMgr = m_app.CommandManager
        edge = cmdMgr.Pick(SelectionFilterEnum.kPartEdgeLinearFilter, "Select
    End If

    'Still no valid selection (probably canceled), just exit.
    If edge Is Nothing Then
        Exit Sub
    End If

    Dim uVec As UnitVector
    uVec = edge.Geometry.direction
    Dim vec = uVec.AsVector
    Dim lengths As Double()
    lengths = m_util.ShowBoundsAlongVector(m_doc, vec)
    SetTextFields(lengths)
    chkVisible.Checked = True

End Sub
```

Success!


2D Sketch
Sketch
Coil
Rib
Create
Draft
Thicken/ Offset
Delete Face
Modify
Generator
Explore
Work Fea
Model
iLogic
crooked.ipt
Solid Bodies(1)
View: Master
Origin
Extrusion1 (New Solid x 6 in)
Extrusion2 (Cut x 12 in)
Extrusion3 (Join x 12 in)
OBB_Primary_Plane_A
OBB_Primary_Plane_B
OBB_Primary_Point_A
OBB_Primary_Point_B
OBB_Secondary_Plane_A
OBB_Secondary_Plane_B
OBB_Secondary_Point_A
OBB_Secondary_Point_B
OBB_Tertiary_Plane_A
OBB_Tertiary_Plane_B
OBB_Tertiary_Point_A
OBB_Tertiary_Point_B
End of Part
OBB
Oriented Bounding Box
Pick Primary
Dimensions
Primary: 17.572
Secondary: 1.581
Tertiary: 5.484
Volume: 152.350
Workfeatures
Visible
Delete
Z
X
My Home
crooked.ipt

Comparison

Programming Skills

Can you do this?
Or should you hire outside help?

iLogic Rules	Class Libraries & Addins
Limited programming skill required. Forms – none.	Strong programming skills, including one of the relevant languages and all of the relevant tools. But easy to merely *use* a class library within iLogic rules

Required Tools ($)

Costs, training/expertise, management

iLogic Rules	Class Libraries & Addins
Does not require anything beyond **Inventor itself**. Rules are stored in the Inventor documents or in ordinary text files, and the “Rule Editor” is the programming environment.	Typically requires a subscription to **Microsoft Visual Studio** (VS). Besides learning the programming language itself, VS is relatively simple to use, IF you limit yourself to straightforward use of the Inventor API. You’ll want “**source control**” of some kind. There are many options. A popular one is “GitHub” – you can use the public github if you don’t mind sharing with the world, or you can host it yourself on your own servers, or you can use a private hosting on Github’s servers. Costs vary. Learning to use a source control system is an issue of its own!

Amount/Type of UI

Matching technology to requirements

iLogic Rules	Class Libraries	Addins
You can create “forms” and simple dialogs (MessageBox and InputBox, etc). As you’re probably aware, these are quite simplistic. If you’re tried to make a big complex form, you know that you’re pushing against the limits.	Simple to add “somewhat more complex” forms, when following a certain pattern Great for non-UI extensions	Inventor is *designed* to let addins create their own commands, including ribbon panels, browser panes, etc. If you need your own commands, then you need an addin. “That’s what they’re for!”

Amount/complexity of code

Matching technology to requirements

iLogic Rules	Class Libraries & Addins
Small. Great for relatively small little automation tasks Using “external” rules will help stretch to larger, more complex code	**Large**. • Use a source control system • Code modularity is easier • Debugging is easier

Source control

Matching technology to requirements

iLogic Rules	Class Libraries & Addins
Source control is Vault. Share external rules (change once, change everywhere)	Standard software development tools, e.g., Github (i.e., a well-understood problem/solution with a huge user base)

Future on web

Option for the future.

- All three technologies can work “in the cloud”.
- UI, or lack thereof, is a factor. Keep “business logic” and “UI” code separate.

Conclusions

Conclusions

OVERLAP

All of these techniques are usable; it’s just a matter of which is best. (square peg … round hole.)

ILOGIC

Great for simple things, especially simple “checking”, simple “configuration”, and simple “forms”

Almost don’t have to be a programmer

ILOGIC + CLASS LIBRARY

Great for isolating “hard stuff” away from simpler iLogic rules

Great for extending iLogic-based UI “a little” beyond simple “iLogic forms”

ADDIN

Great for extending Inventor UI “seamlessly”

Great for harder problems, with more programming

Thanks!

Questions?

Other relevant classes:

PM224007 - Taking It to the Next Level: Drawing Automation with Inventor	Wed, 2:45-3:45
IM221410 - The Power of iLogic Design Automation: How Did We Get Here?	Wed, 4:30-5:30
SD224078 - Creating Add-Ins for Inventor	Thurs, 10:30-12
MFG227120 - Automate CNC Machining By Using the Inventor HSM API and iLogic for CAM Products	Thurs, 2:45-3:45

AUTODESK®

Make anything™

Autodesk and the Autodesk logo are registered trademarks or trademarks of Autodesk, Inc., and/or its subsidiaries and/or affiliates in the USA and/or other countries. All other brand names, product names, or trademarks belong to their respective holders. Autodesk reserves the right to alter product and services offerings, and specifications and pricing at any time without notice, and is not responsible for typographical or graphical errors that may appear in this document.

© 2018 Autodesk. All rights reserved.