MFG319306

Taking It to The Next Level: Drawing Automation with Autodesk Inventor

Thomas Fitzgerald
Autodesk

Learning Objectives

- Learn the common Drawing Automation structure
- Become intimate with Product Design
- Capture critical Design information
- Improve Design quality and consistency

Description

We've all heard it before: "You can't automate drawings with iLogic." Well, we're here to tell you that you can! In this instructional demo, you will learn the basic techniques of developing the logic required to automate 2D drawings with iLogic and the Inventor API. From creating views, adding dimensions and balloons, and working with sketched symbols and even parts lists. This demo will shed an abundant amount of light on how to develop your drawing automation, as well as what to avoid, ensuring a robust and stable automation project.

Speaker

Thomas Fitzgerald is a Senior Implementation Consultant specializing in Inventor Automation and Data Management. Thomas has consulted with a large number of companies with a very diverse exposure to both large and small engineering departments. His work experience includes mechanical design within the Automotive, Shipbuilding, Mining and Material Handling industries as well as custom configuration applications utilizing the Inventor API, iLogic, Microsoft SQL Server, and Microsoft Visual Studio.

Thomas has over 20 years of experience within the mechanical design and manufacturing industries using numerous Autodesk products. He is an Autodesk Certified Instructor as well as having his Microsoft Certified Systems Administrator credentials.

AUTODESK. UNIVERSITY

Table of Contents

Taking It to The Next Level: Drawing Automation with Autodesk Inventor 1
WHAT is Inventor Automation? 4
WHY should I embrace Inventor Automation? 4
The 4 Key Aspects of Inventor Automation 5
- Logic 5
- Model Development 5
- External Data Sources 5
- User Interface 5

In This Class… 6
Planning 7
Logic 7
  Where Do I Start? 7
  To iLogic or not to iLogic? 7
  The Inventor API 8
  Code Structure and Organization 10
  Naming Convention 10
Drawing Automation 11
  What is Drawing Automation? 11
  Templates 11
  Sheets 11
  Views 11
  Parts Lists 12
  Dimensions 12
  Balloons 13
  Sketched Symbols 13
Model Development 14
  Attributes 14
  Work Points 14
Finally 14
  External Data Sources 14
  User Interface 14
  How About an Inventor Add-In? 15

Links 15
My Favorites 15
Code Examples 16
Today is your Lucky Day! 16
The Easy Stuff 16
Sheets 16
Views 17
Parts List 19
Reorder Parts List 19
The Hard Stuff 20
Balloons 20
Dimensions 22
Sketched Symbols 24

WHAT is Inventor Automation?

For the most part, people who use any type of desktop application understands what automation is. If you've used Microsoft Excel you may have heard of Macros, tools developed and designed within Excel to accomplish a specific task. Inventor Automation is very much the same in the sense that while the automation can take the form of many different things, in essence, it is a tool or a series of tools to automatically accomplish a specific task or tasks.

Some Inventor Automation can be extremely simple, like using parameters within a part model that can Suppress or Unsuppress a Feature depending on the value of those parameters.

You could develop a series of iLogic rules to accomplish tasks, like updating iProperties based upon different model criteria or Replacing Components in an assembly based upon selections made in an iLogic Form, even update text blocks within an associated Drawing.

Or you could go all out and attempt to syntax your entire engineering development process using Microsoft Visual Studio to create input forms and develop Inventor API logic to create components on the fly, assemble the necessary components per industry and business specifications, and finally develop all the associated output drawings and documentation, essentially removing the "man" from the process.

WHY should I embrace Inventor Automation?

Now that we understand *WHAT* Inventor Automation is, let's take a look at the reasons *WHY* you might want to incorporate automation into your engineering processes.

First, in my experiences working with manufacturing companies big and small, all over this world, fabricating and manufacturing many different types of products, one thing always rings true, there are patterns and repeatable plays in every environment. The key is to find the ones where Inventor Automation can be of assistance. This simple task requires an intimate knowledge of all the stages that Inventor plays a part in your process.

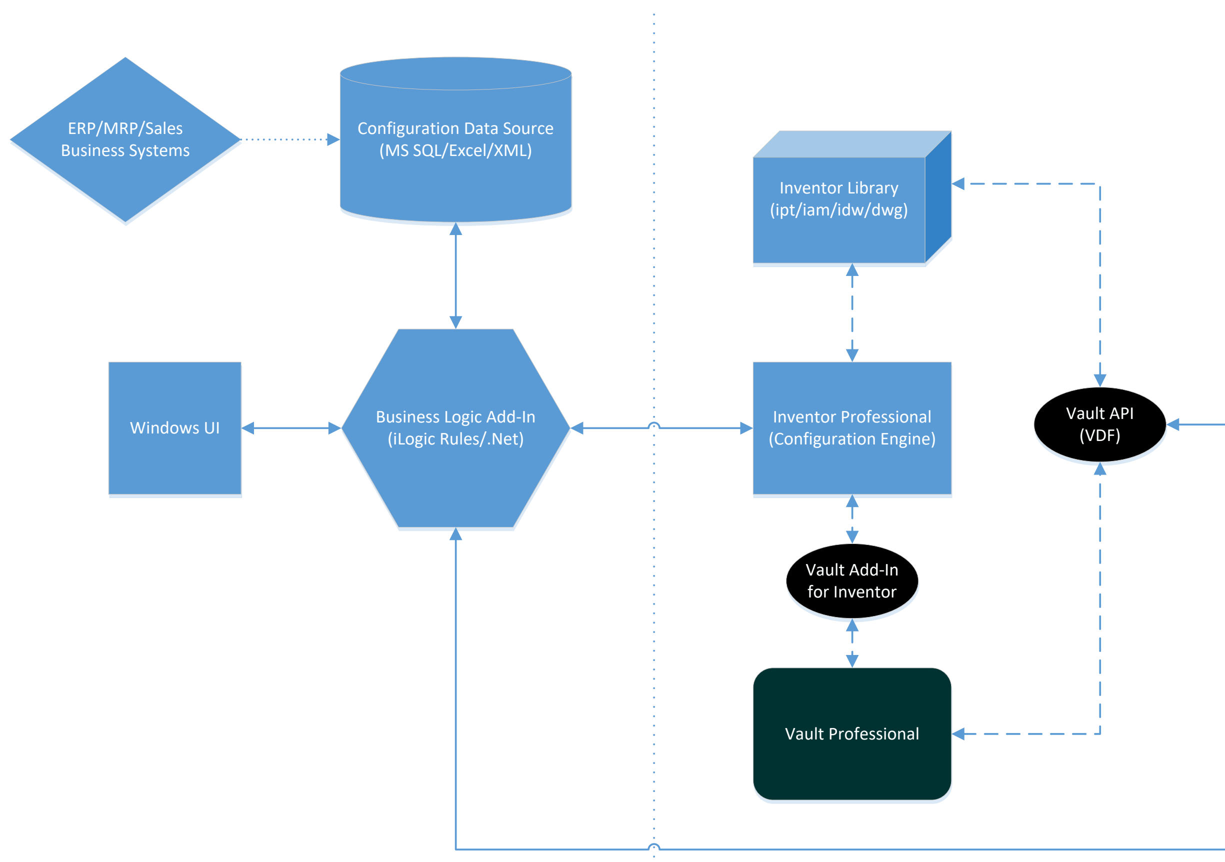
Say for instance, you have a specific format for the iProperty Description of your 3D models or any of the iProperties for that matter. If the formatting is predictable, if it is Standardized, then this is a situation where Inventor Automation can come to the rescue. You could develop logic to collect information from the model, transform that information, and then overwrite the iProperties with the correct, newly formatted information. It is always correct, it is always consistent, and it never asks for a coffee break.

Another example, one that I find an extreme amount of value in is Drawing Automation. The act of taking properly developed and designed 3D models, generating drawing views, adding dimensions and balloons, updating Parts Lists, adding and manipulating Sketched Symbols, this is the arena where Inventor Automation can do a lot of work and be ultimately beneficial. But this little gem is hard to come by. It takes a very large amount of time to plan, prepare, and process all the requirements to implement a project of that nature. But fear not, it can be done.

The 4 Key Aspects of Inventor Automation

Let me preface this by saying that any successful Inventor Automation project will require a plan. A plan will require knowledge and intelligence, the kind of intelligence that provides status and the current state of affairs. You must be intimately familiar with your data or surround yourself with the right team of people that do. You must know where you're starting from before you can determine how to get to where you want to be.

Autodesk Inventor Automation Data Flow



- **Logic**
- **Model Development**
- **External Data Sources**
- **User Interface**

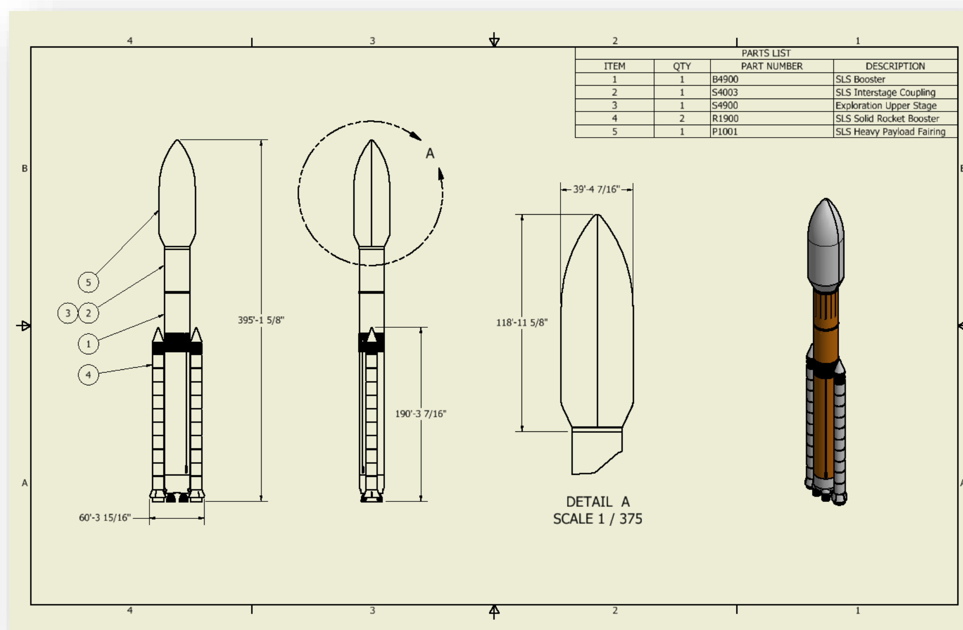
In This Class…

Knowing what we know about Inventor Automation and how vast the horizon is with its scope, let's get an understanding as to what we will be discussing in this AU class.

One of the steps within design processes companies ask me about on a frequent basis is Model Configuration. In the past, I had created a couple demonstration datasets that displayed some of the Out of the Box features of iLogic and how anyone can build a rudimentary Product Configurator. Being able to take a pool or collection of part and assembly models and put them together in an accurate and consistent manner, obeying best modeling and assembling practices ensuring the stability and integrity of the 3D information. This is definitely an area in the development process where everyone can relate.

This class will demonstrate the process of using a developed model configuration and "automatically" generate the 2D design documentation used for quoting, fabrication, or manufacturing. Topics of discussion will be around the essential features needed to develop 2D drawings, to include:

- Creating, naming, and activating Sheets
- Defining the model reference
- Creating, naming, locating, and scaling Views
- Creating and modifying Parts Lists
- Adding Dimensions
- Placing Balloons
- Adding and locating Sketched Symbols
- Working with iProperties and Parameters for Title Blocks



Planning

For any automation project to be successful, you must plan. Having a plan, with objectives to monitor and manage success metrics, will allow for contingencies to be accounted for, as well as having some type of road map to follow for the design. How many Sheets are needed for the Views and Dimensions needed to capture a design? What notes need to be included? Do the notes vary depending on the type of design needing to be documented? Do we know enough to predict how the drawings need to be configured? Does a user need to interact with the process? There are many questions that you must ask yourself, a plan will help you understand what those questions are.

Drawing Automation is a departure from Model Automation. In my previous AU classes, I discussed how to develop Product Configurators, taking models and putting them together in specific configurations to represent what it is you produce or manufacture. 3D models are engineering "gold" when it comes to conceptualizing and visualizing how to put something together. It is, without a doubt, essential in performing analysis, reducing costly prototype creation, and expediting roll out time from concept to production. But having the detailed drawings of those configurations is where most of us find the value. Without the drawings, humans have a challenging time manufacturing anything with quality and consistency. Imagine building a house without blueprints! Good Luck.

To pull off a Drawing Automation project in Inventor requires understanding of how Inventor works. Remember, iLogic does not change how Inventor works, it only changes how you work Inventor. Without knowing that a View requires a reference, or Dimensions need 2 points selected, or anything of the sort, it will be extremely difficult to understand what is required to automate the process. Also, Drawing References are Models, don't forget to give those Models some special attention. Look for the Model Development section below to understand how to prepare those valuable 3D models.

Logic

Where Do I Start?

I'll admit, I'm not a programmer. I've never taken a class, I've never gone to school to write code, I didn't have a Yoda to seek advice from, well unless you consider Google a Yoda like figure. The internet has a wealth of information available to do certain little steps with whatever code language you would like to learn. I started learning Visual Basic .Net a while ago because of the need to understand iLogic. But, depending on your Inventor Automation architecture, you can use C# .Net as the code language for the logic, but the first step is to understand what language you want to use.

To iLogic or not to iLogic?

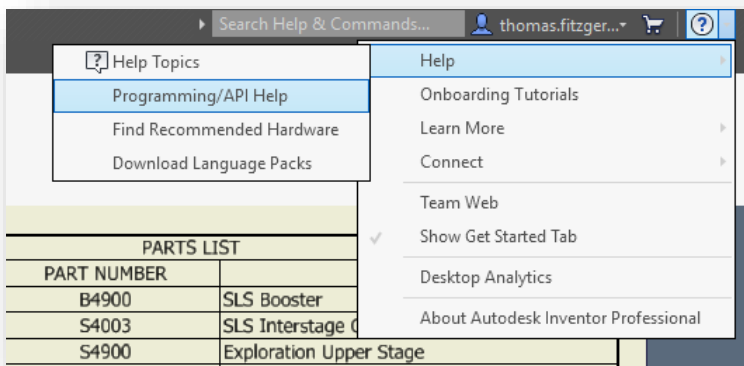
I get this question all the time. "Which is better, iLogic or the Inventor API?" Honestly, they are one in the same. I've always considered iLogic like a door to the Inventor API. It's just an environment with a library of commands that leverage the Inventor API to do Inventor tasks.

The key to make the decision of whether to use iLogic or not depends on several different factors. If you want to accomplish simple little tasks like updating iProperty information or if you want to update Part Feature data or maybe even if you want to create a pseudo iAssembly, basically controlling the Suppression state of components within an assembly.
Another factor that should be taken into consideration is security. Because of its readily available tools and interface, most companies when designing and developing their iLogic code, use the default settings and store their iLogic code within the files themselves. This presents a couple undesirable situations. First, the code is exposed to anyone that has access to the file, which in most instances, is fine. But one can spend a lot of time and effort putting together the code in such a way that other personnel should not be tinkering with it. Often, there could be sensitive information involved with the code that for obvious reasons, we don't want out there and available to just anyone. External Rules can alleviate this problem to some degree by securing the Rule logic out on the network somewhere, and us the network security to prevent prying eyes from seeing exactly what is in the code.

If you're looking to do a more complex series of tasks with your Inventor Automation, like connecting to other business systems, or if you need real time updated database information, or if want to accomplish something like what I have demonstrated in this class, then I'm going to suggest using Visual Studio and the Inventor API. Creating an Inventor Add-In can provide you the security and single environment to do all your UI and Logic tasks needed for your automation project.

The Inventor API

API, or Application Programming Interface, is the recipe for ***Doing*** in Inventor. If you want to draw a line in a sketch, you select the Line button and then you input what Inventor requires. First, you select a start-point, then you select an end-point. Each task in Inventor has a set of Functions to do everything from drawing lines, to creating assembly constraints, to scaling drawing views. Understanding the intricate steps of each task in Inventor is crucial for planning and implementing an automation project.

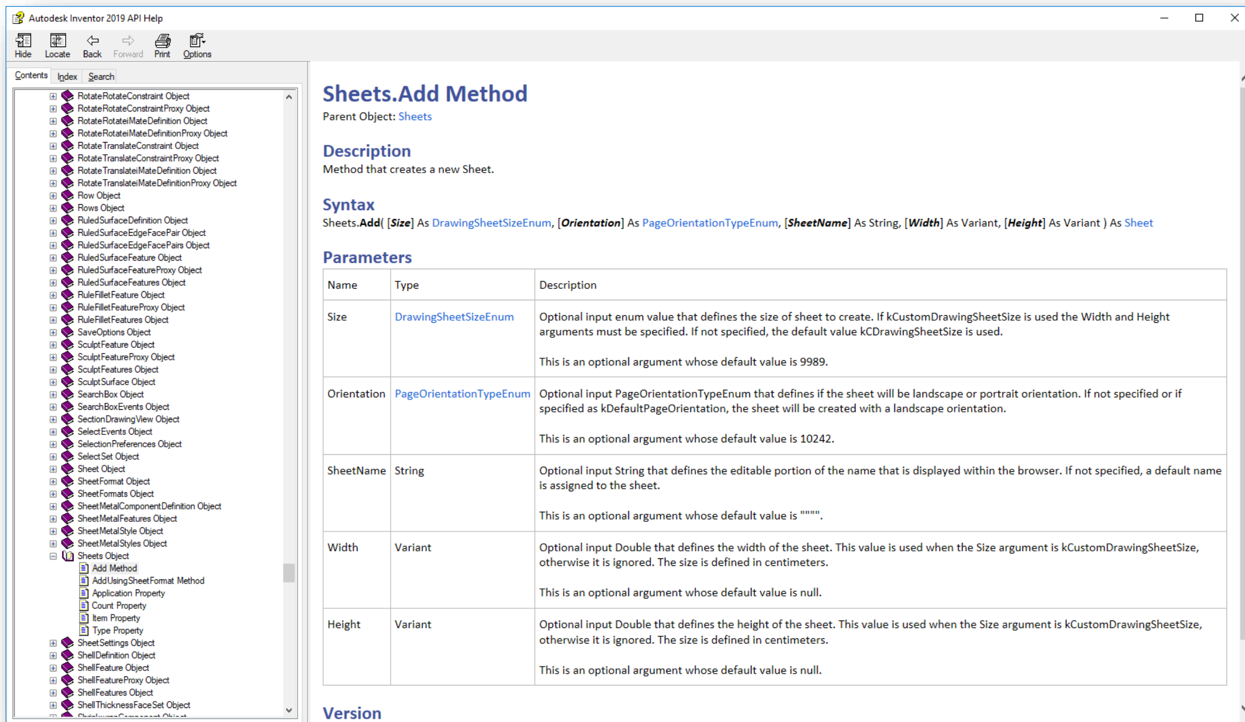


You can also access it online:

http://help.autodesk.com/view/INVNTOR/2019/ENU/?guid=GUID-DE98632B-3DC0-422B-A1C6-8A5A15C99E11

During the planning stage of your automation, outlining the Inventor steps from the perspective of the Inventor API will greatly expedite the logic development. Take for instance the concept of developing your drawing automation. One of the first steps when manually creating a drawing is defining what template to use, naming it, and then saving it. After that, how many sheets are needed?

Look in the Inventor API help to see the function to create a Sheet.



Now, I know, it's daunting. I was lost and confused the first dozen times I looked at it too. But, there is a lot of information out there on how to read and use the Inventor API. Look it up, do a search on YouTube, you'll be surprised on the number of examples, tutorials, and videos available.

Code Structure and Organization

What's the best way to structure your code?

What I've learned through my experiences is, the more generic, the better. Each Function, Subroutine, or Rule that you will create should be as simplistic and ambiguous as possible. Each one will have an input, the data that is required for the nature of the Function. Data mining, manipulation, and interpretation is at its most prevalent in this phase of the project.

Even if you intend on using iLogic as your logic tool, organize your Rules in such a way that they perform a specific function or task. You can call on those rules at any time from another rule to better control when your logic fires specific routines. It's always better to have a lot of small rules then one large, complex rule. Your Functions and Subroutines should be written in the same manner.

Naming Convention

The ability to identify files to use in Inventor Automation is one of the key points to be aware of for Model Development. As an example, if you as an Inventor user decided that you wanted to add a component to an assembly, you need two bits of information; File Name and File Location. Some companies use what is called a "smart file naming convention" where each character in the File Name has some sort of meaning, typically followed by some sequential number to ensure uniqueness. Some companies use an arbitrary or random combination of Alpha and Numeric characters to name their files. Typically, these companies use a Document Management system like Autodesk Vault where database properties can be searched to identify files. Instead of relying on File Names to be logical and have meaning, companies using a PDM can create search queries to find files based upon engineering or design information.

Whatever process you use to name files truly doesn't matter. What is important to focus on is what the convention might be. Similarly, Drawing Automation may require Sheets, Views, Work Features, and Entities to be named to identify and access them for different logic processing. If you want to add a Dimension to a View, what View? Adding a Balloon to a component in an Assembly that is being referenced in a View? You might need to add an Attribute tag to a Face in the component. What's the name of the Attribute you want to find? Giving them logical names makes this exercise much simpler.

Drawing Automation

What is Drawing Automation?

The title of this section really tells the story. What do we have to do to make sure drawings get created automatically? Automatically is a word that is subject to interpretation. To most, it means that something happens without any human interaction. That would be in a perfect world. However, in engineering, we simply want to minimize the amount of human interaction, particularly if a process is repeatable, predictable, and relatively consistent. If we can automate the process, then we can ensure quality and consistency by removing the possibility for human error. It also allows us to utilize our staffing resources much more efficiently.

In Autodesk Inventor, Drawing Automation has specific steps to follow to accommodate the associations tasks have with one another. As in the example of generating a View, we first must identify a reference for that View. The next sections will discuss, in more detail, the process of developing your Drawing Automation.

Templates

Defining a template or a series of templates to create your drawings from is the absolute first step in the process. You can use your default templates or create a whole new series of templates specific for the Drawing Automation. It doesn't matter if you use the IDW or DWG file extension. Record the file path location of your templates, it will be necessary later.

Depending on the scale of your Drawing Automation, you may need many different templates. The template you use for one type of product may differ from another simply because the level of detail required. Creating a drawing of a single part file obviously would be different then a drawing of a complex assembly. Most of the time, you can accommodate a lot of these differences.

Sheets

After we identify what drawing template to use, next is Sheets. What number of Sheets do we need? What size Sheets do we need? What name do we want to apply to the Sheets? What about Borders and Title Blocks? The following items need to be considered:

- Naming Convention for your Sheets
- Sizing you Sheets, A size, B size, etc.
- Activating a specific Sheet
- Deleting Sheets

Views

Creating various Views allows us to capture the necessary geometric information to accurately define how something is configured. Inventor creates Views based upon Model Reference. We as end users can define which Model Reference to use for each View that is generated, depending on the type of View that is needed. The Inventor API allows us to define that Model Reference as well.

If you are detailing a single Part file, well, the Model Reference would be consistent throughout the drawing. Well, what about a complex assembly? There are many different components, subassemblies, orientations, visibility options, etc., that need to be taken into consideration. This is where understanding the power and strengths of Inventor will help you out. The following items need to be considered:

- Naming Convention for your Views
- What type of View is needed, Base, Projected, Section, etc.
- View Scaling
- View Positioning
- View Orientation

Use a powerful Naming Convention to identify Parts and Assemblies that need the Visibility turned off from within each View. There are methods to iterate through a View and turn off the Visibility of select components and subassemblies based upon their name. Leverage Level of Detail Representations in your Assemblies to control what graphical information is present within the Views as well.

Tips!!!!
When working with Views, there are a few things that aren't apparent when it comes to writing the logic code. First, Views are placed based upon 2D Points, essentially an X and Y. Second, View scale is difficult to judge, I always manage scaling based upon Sheet size. This way no matter what Sheet size I use, the Views will always fit.

Parts Lists

For the most part, if you configure your Views correctly, the Parts List will take care of itself. But there are occasions where the Parts List does not reflect that in which you want to represent. All of us as one point have created a View based upon a Level of Detail, then placed a Parts List. Why do the components and subassemblies that I have suppressed still show up in the Parts List? Sometimes, we must filter components and subassemblies out because Level of Detail does not control the Parts List, View Representation filters do. We can also write code to do the same thing. We can also populate the Parts Lists with custom or "dynamic" information if we'd like. The following items need to be considered:

- Parts Lists location
- Content filtering
- Item renumbering

Dimensions

The topic of creating and placing dimensions in an automated fashion has challenged consultants like me for a while. An end user can simply look at a View and with the basic of information, apply the dimensions needed to detail it out. But automation cannot look at a View and process it visually. Code needs other criteria, other information to make determinations about how to place dimensions. A method I use is by applying Work Points to the components or subassemblies that graphically compose where my dimensions should go. After I create the Work Points, I rename them following a predetermined Naming Convention so when my code needs them, it can find them.

Depending on the type or category of file that needs 2D documentation, the number of Dimensions can be immense. Typically for a quote, you may just need overall dimensions, a footprint, connectivity information, etc. But for fabrication or manufacturing documentation, well, you can imagine. How do we easily control, manage, and consume the information needed to place all those Dimensions? As I discussed in my previous AU classes, I like to leverage External Data Sources for situations like this. Using Microsoft Excel or SQL databases allow for the flexibility and user friendliness needed to input and edit large amounts of data. In my example, I use an Excel spreadsheet for my Dimension information.

Tips!!!!
When working with Dimensions, there are a few things that aren't apparent when it comes to writing the logic code. Dimension placement is based upon 2D Points, once again, X and Y. You need a start-point, an end-point, and a point for the text placement. Because the points are dependent upon 3D graphical information, points need to be "projected" into 2D space. Get familiar with Geometry Intent in the Inventor API.

Balloons

When I first started using Inventor, I was just amazed at the fact that if I point a Balloon at something else in a View, the Balloon number updated and reflected the corresponding item number in the Parts List. Coming from AutoCAD where quadruple checking the manual table we called a BOM to make sure item numbers matched was extremely tedious, time consuming, and prone to mistakes. As I started working with Drawing Automation, I found that working with Balloons was much easier than I had anticipated. Identifying what needed a Balloon was a simple as Attribute tagging. In previous versions of Inventor, this was accomplished by using a utility developed by Brian Ekins called Attribute Helper. It provides a simple, easy to use interface to select Faces, Edges, Points, etc. and apply a named "tag" to it. Then all you need to do is iterate through the files and find the Attribute, access its object and viola, you now know what to attach a Balloon to. The hardest part of it all is placing the Balloon itself, ensuring you don't interfere with other annotations in the drawing.

Tips!!!!
When working with Balloons, there are a few things that aren't apparent when it comes to writing the logic. The Balloon placement can be controlled based upon the attachment point or relative to the View. I find it easier to use the attachment point as reference and controlling the direction of the Balloon based upon where the attachment point entity is. If the entity is right of the View centerline, then the direction of the Balloon is to the right. If it's to the left, then left. You get the point.

You can also control the Balloon shape type as well as the Balloon value.

Sketched Symbols

In Inventor, we control text blocks by using Sketched Symbols. They are predefined, stored within the drawing, easy to use, and can even be dynamic. They can even have Leaders to attach to geometry and move around. In my opinion, working with Sketched Symbols is the easiest of them all. You simply call the name of the symbol you want and define its location relative to the Sheet. Add Prompted Entry fields if you want to have dynamic Sketched Symbols.

Model Development

Attributes

Have you ever just wanted to give a Face or an Edge a name? Well, now you can! Ok, you have been able to for a long time now, but now we can do it easily. In Inventor 2019, if you select a Face or an Edge, and Right Click, you will see in the menu the opportunity to Assign Name. The name you provide will be the named Attribute to search for, as I explained when applying Balloons.

Another way to add Attributes, if you're not using Inventor 2019, is by using the Attribute Helper:

https://modthemachine.typepad.com/AttributeHelperSetup_2.6.zip

The key of using Attributes for designating faces and edges is to have an adequate Naming Convention and understanding which files will be involved with the Balloon process as well as their orientation.

Work Points

I have found great success with using Work Points to define where the start-point and end-point of any dimension is located. I create the appropriate Work Point in the necessary file, give it a logical name based upon a Naming Convention, and turn off the visibility. Because all dimensions have 2 points, I postfix each corresponding Work Point with a 1 and a 2. It allows for looping within the code much easier. I theorize that using Attributes to tag vertices will work, although I have not tried the workflow.

Finally

External Data Sources

As I mentioned before, External Data Sources can be extremely valuable, particularly if your Drawing Automation has aspects that include numerous variables. Microsoft Excel and Microsoft SQL Server are tools to use to accomplish storing and retrieving variable information, on demand. They also provide a user-friendly means of adding, updating, and administrating the design specific information as your Drawing Automation scales and grows.

User Interface

Not all Drawing Automation projects will require a User Interface. Typically, a User Interface will be created and used when developing Design Automation, like a product configurator. If this is the case, the Drawing Automation will usually an extension of the Design Automation and leverage its UI. However, a UI can be created to initiate the Drawing Automation process and provide a means to apply critical information to satisfy the inputs for the Drawing Automation.

How About an Inventor Add-In?

The next logical step in terms of complexity, level of effort, and functionality is developing an Inventor Add-In. Running iLogic rules by manually triggering them, or developing an iLogic form or forms, or creating windows forms in Visual Studio and using an iLogic rule to run them is very common ways of interfacing with your end users. But some companies have a need or desire to streamline how end users work in the Inventor environment. For those that want to go all out and truly "Customize" Inventor, developing an Inventor Add-In is the way to go.

My friend Brian Ekins has talked about this subject so many times and all the information needed to get started creating an Inventor Add-In is available online. Here are some helpful links:

http://modthemachine.typepad.com/files/VBAtoAddIn.pdf

http://au.autodesk.com/au-online/classes-on-demand/class-catalog/2016/inventor-professional/sd17917#chapter=0

Links

My Favorites

https://stackoverflow.com

http://modthemachine.typepad.com/my_weblog/2013/02/inventor-api-training-lesson-1.html

Code Examples

Today is your Lucky Day!

Here is the code I used in my Data Set. Use it as you wish. Obviously, the code samples below are specific to the dataset I created and will need some tweaking to work in your situation. If you'd like a copy of the entire Data Set, contact me at: thomas.fitzgerald@autodesk.com

The Easy Stuff

Sheets

```
Sub Main
        Dim sheetNames As New List(Of String)
        sheetNames.Add("MAIN")
        sheetNames.Add("Detail")

        Dim sheetSize As DrawingSheetSizeEnum = 9988

        Dim invDoc As DrawingDocument = ThisApplication.ActiveDocument
        Dim oSheet As Sheet = Nothing

        '*** Add the number of Sheets and name the Sheets
        For index As Integer = 1 To oSheets.Count
                oSheet = invDoc.Sheets.Add(sheetSize, ,sheetNames(index - 1))
        Next

        '*** Delete Sheet:1
        For Each oSheet In invDoc.Sheets
                If oSheet.Name = "Sheet:1" Then
                        oSheet.Delete
                End If
        Next

        '*** Activate the MAIN Sheet
        For Each oSheet In invDoc.Sheets
                If oSheet.Name = oSheets(0) & ":1" Then
                        oSheet.Activate
                End If
        Next
End Sub
```

Views

```
Sub Main
        Dim oDoc As DrawingDocument = ThisApplication.ActiveDocument
        Dim oSheet As Sheet = oDoc.ActiveSheet

        DeleteView(oSheet)

        '*** Define View Locations
        Dim sheetCenterVertical As Double = ((oSheet.Height / 2) - 1.5)
        Dim baseViewHorizontal As Double = (oSheet.Width * .17)
        Dim projViewHorizontal As Double = (oSheet.Width * .4)
        Dim detailViewHorizontal As Double = (oSheet.Width * .6)
        Dim isoViewHorizontal As Double = (oSheet.Width * .85)

        '*** Define Model Reference
        Dim modelReference As Document =
ThisApplication.Documents.Open(SharedVariable("currentPath") & "\" &
SharedVariable("newFileName") & ".iam", False)
        Dim oScale As Double = 1

        '*** Create Points for View Locations
        Dim oPoint(4) As Point2d
        oPoint(1) =
ThisApplication.TransientGeometry.CreatePoint2d(baseViewHorizontal,
sheetCenterVertical)
        oPoint(2) =
ThisApplication.TransientGeometry.CreatePoint2d(projViewHorizontal,
sheetCenterVertical)
        oPoint(3) =
ThisApplication.TransientGeometry.CreatePoint2d(detailViewHorizontal,
sheetCenterVertical + 2)
        oPoint(4) =
ThisApplication.TransientGeometry.CreatePoint2d(isoViewHorizontal,
sheetCenterVertical + 1)

        '*** Define View Orientation
        Dim viewOrientation(4) As ViewOrientationTypeEnum
        viewOrientation(1) = 10764
        viewOrientation(4) = 10759

        '*** Define View Style
        Dim viewStyle(4) As DrawingViewStyleEnum
        viewStyle(1) = 32258
        viewStyle(2) = 32258
        viewStyle(3) = 32258
        viewStyle(4) = 32259

        '*** Create the Views
        Dim oBaseView As DrawingView = Nothing
        Dim oProjView As DrawingView = Nothing
        Dim oDetailView As DrawingView = Nothing
        Dim oIsoView As DrawingView = Nothing
        Dim detailCenter As Point2d = oPoint(2)
```

```
        For i = 1 To 4
                If i = 1 Then
                        oBaseView =
oSheet.DrawingViews.AddBaseView(modelReference, oPoint(i), oScale,
viewOrientation(i), viewStyle(i))
                        oBaseView.Name = "BaseView" & i
                        Call SetViewScale(oSheet, oBaseView)
                ElseIf i = 2 Then
                        oProjView =
oSheet.DrawingViews.AddProjectedView(oBaseView, oPoint(i), viewStyle(i))
                        oProjView.Name = "ProjView" & i
                ElseIf i = 3 Then
                        detailCenter.Y = detailCenter.Y + 6.1
                        oDetailView =
oSheet.DrawingViews.AddDetailView(oProjView, oPoint(i), viewStyle(i), True,
detailCenter, 3.2)
                        oDetailView.Name = "View" & i
                        oDetailView.Scale = oProjView.Scale * 2
                ElseIf i = 4 Then
                        oIsoView =
oSheet.DrawingViews.AddBaseView(modelReference, oPoint(i), oScale,
viewOrientation(i), viewStyle(i))
                        oIsoView.Name = "IsoView" & i
                        Call SetViewScale(oSheet, oIsoView)
                End If
        Next

        iLogicVb.RunRule("Add Parts List")
End Sub

Function DeleteView(oSheet As Sheet)
        For Each view As DrawingView In oSheet.DrawingViews
                View.Delete()
        Next
End Function

Function SetViewScale(oSheet As Sheet, oView As DrawingView)
        Dim viewScale As Double = Nothing
        Dim trueViewHeight As Double = Nothing
        Dim viewName As String = oView.Name
        Dim viewHeight As Double = Nothing

        If Not viewName.Contains("Iso") Then
                viewHeight = (oSheet.Height * .6)
        Else
                viewHeight = (oSheet.Height * .45)
        End If

        trueViewHeight = oView.Height
        viewScale = (viewHeight / trueViewHeight)
        oView.Scale = viewScale
End Function
```

Parts List

```
Sub Main
    Dim invDoc As DrawingDocument = ThisApplication.ActiveDocument
    Dim oSheet As Sheet = invDoc.ActiveSheet

    Try
        DeletePartsList(oSheet)
    Catch
    End Try

    Dim oDrawingView As DrawingView = oSheet.DrawingViews(1)
    Dim oBorder As Border = oSheet.Border
    Dim oPlacementPoint As Point2d

    If Not oBorder Is Nothing Then
        oPlacementPoint = oBorder.RangeBox.MaxPoint
    Else
        oPlacementPoint =
ThisApplication.TransientGeometry.CreatePoint2d(oSheet.Width, oSheet.Height)
    End If

        Dim partsListBomType As PartsListLevelEnum = 46593
    Dim oPartsList As PartsList = oSheet.PartsLists.Add(oDrawingView,
oPlacementPoint, partsListBomType)
End Sub

Private Sub DeletePartsList(oSheet As Sheet)
    Dim oPartList As PartsList
    For Each oPartList In oSheet.PartsLists
        oPartList.Delete()
    Next
End Sub
```

Reorder Parts List

```
Dim oDrawDoc As DrawingDocument = ThisApplication.ActiveDocument
Dim oPartList As PartsList = oDrawDoc.ActiveSheet.PartsLists.Item(1)
oPartList.Renumber()
oPartList.SaveItemOverridesToBOM()
```

AUTODESK UNIVERSITY

The Hard Stuff

Balloons

```
Sub Main
    Dim oDrawDoc As DrawingDocument = ThisApplication.ActiveDocument
    Dim oActiveSheet As Sheet = oDrawDoc.ActiveSheet
        Dim oDrawingView As DrawingView = oActiveSheet.DrawingViews.Item(1)
        Dim oAssemblyDoc As AssemblyDocument =
oDrawingView.ReferencedDocumentDescriptor.ReferencedDocument
        Dim oTG As TransientGeometry = ThisApplication.TransientGeometry

        oDrawingView.ViewStyle = 32257

        '*** Iterate through Assembly to find Parts
        Dim oOccs As ComponentOccurrences =
oAssemblyDoc.ComponentDefinition.Occurrences
        For Each oOcc As ComponentOccurrence In oOccs
                If oOcc.DefinitionDocumentType = 12291 Then
                        Call TraverseSubAssy(oActiveSheet, oDrawingView, oTG,
oOcc.SubOccurrences)
                Else
                        Call CreateBalloon(oActiveSheet, oDrawingView, oTG,
oOcc)
                End If
        Next

        oDrawingView.ViewStyle = 32258
End Sub

Private Sub TraverseSubAssy(oActiveSheet As Sheet, oDrawingView As
DrawingView, oTG As TransientGeometry, oOccs As ComponentOccurrences)
        For Each oOcc As ComponentOccurrence In oOccs
                If oOcc.DefinitionDocumentType = 12291 Then
                        Call TraverseSubAssy(oActiveSheet, oDrawingView, oTG,
oOcc.SubOccurrences)
                Else
                        Call CreateBalloon(oActiveSheet, oDrawingView, oTG,
oOcc)
                End If
        Next
End Sub

Public Function CreateBalloon(oActiveSheet As Sheet, oDrawingView As
DrawingView, oTG As TransientGeometry, oOcc As ComponentOccurrence)
        Dim oModelDoc As Inventor.PartDocument
        oModelDoc = oOcc.Definition.Document

        '*** Find the tagged Faces
        Dim oObjs As ObjectCollection =
oModelDoc.AttributeManager.FindObjects("Balloon")
        If oObjs.Count = 0 Then
```

```
            Exit Function
        End If

        Dim oFace As Inventor.Face
        oFace = oObjs.Item(1)

        Dim oFaceProxy As Inventor.FaceProxy
        Call oOcc.CreateGeometryProxy(oFace, oFaceProxy)

        Dim oDrawCurves As Inventor.DrawingCurvesEnumerator
        oDrawCurves = oDrawingView.DrawingCurves(oFaceProxy)

        Dim oDrawingCurve As Inventor.DrawingCurve
        oDrawingCurve = oDrawCurves.Item(1)

        Dim midPoint As Point2d = Nothing
        midPoint = oDrawingCurve.MidPoint

        Dim oLeaderPoints As ObjectCollection =
ThisApplication.TransientObjects.CreateObjectCollection

        '*** Locate where the Balloon will be placed
        If midPoint.X > oDrawingView.Position.X Then
            oLeaderPoints.Add(oTG.CreatePoint2d(midPoint.X + 2, midPoint.Y
- 1))
        Else
            oLeaderPoints.Add(oTG.CreatePoint2d(midPoint.X - 2, midPoint.Y
- 1))
        End If

        '*** Must be the LAST point added to the array.  This is the Balloon
Leader attachment point.
        Dim geoIntent As Inventor.GeometryIntent =
oActiveSheet.CreateGeometryIntent(oDrawingCurve, .5)
        oLeaderPoints.Add(geoIntent)

        '*** Create the balloon
        Dim oBalloon As Inventor.Balloon
        oBalloon = oActiveSheet.Balloons.Add(oLeaderPoints)
End Function
```

Dimensions

```
Sub Main
	Dim oDoc As DrawingDocument = ThisApplication.ActiveDocument
	Dim dimNumber As Integer = 0
	Dim dimPt(1) As String
	Dim oOcc(1) As Object
	Dim viewName As String = String.Empty
	Dim sheetNumber As Integer = Nothing
	Dim dimType As String = String.Empty

	'*** Make connection to Excel File
	Dim xVal As String =
GoExcel.CellValue("C:\Vault\Designs\Customers\Autodesk University\AU
2018\Dataset\Dimensions.xlsx", "Sheet1", "A1")

	'*** Set Variables
	Dim cellVal As Object = Nothing
	Dim inc As Integer = 2

	'*** Loop through Excel File Until empty row
	Do
		dimPt(0) = GoExcel.CellValue("A" & inc)
		dimPt(1) = GoExcel.CellValue("B" & inc)
		viewName = GoExcel.CellValue("C" & inc)
		sheetNumber = GoExcel.CellValue("D" & inc)
		dimType = GoExcel.CellValue("E" & inc)

		ActiveSheet = ThisDrawing.Sheet("Sheet:" & sheetNumber)

		Dim oView As DrawingView = ActiveSheet.View(viewName).View
		Dim oSheet As Sheet = oDoc.Sheets.Item(sheetNumber)

		For count As Integer = 1 To 2
			oOcc(count - 1) = FindPoints(oView, dimPt(count - 1))
		Next

		If oOcc(0).Name = String.Empty Or oOcc(1).Name = String.Empty
Then
			MessageBox.Show("Points Could Not Be Found")
		Else
			Call CreateLinearDimension(oDoc, oSheet, oView,
dimPt(0), dimPt(1), oOcc(0), oOcc(1), dimType)
		End If

		inc = inc + 1
		cellVal = GoExcel.CellValue("A" & inc)
	Loop Until (cellVal Is Nothing OrElse
String.IsNullOrEmpty(cellVal.ToString()))
End Sub

Private Function FindPoints(oView As DrawingView, dimPoint As String)
	Dim oAssyDoc As AssemblyDocument =
oView.ReferencedDocumentDescriptor.ReferencedDocument
```

```
        Dim occ As Object
        Dim oDef As Document = Nothing

        For Each oOcc As ComponentOccurrence In
oAssyDoc.ComponentDefinition.Occurrences
                oDef = oOcc.Definition.Document
                For Each oWorkPoint As WorkPoint In
oDef.ComponentDefinition.WorkPoints
                        If oWorkPoint.Name = dimPoint Then
                                occ = oOcc
                        End If
                Next
        Next
        Return occ
End Function

Function CreateLinearDimension(oDoc As DrawingDocument, oSheet As Sheet,
oView As DrawingView, dimPt1 As String, dimPt2 As String, oOcc1 As
ComponentOccurrence, oOcc2 As ComponentOccurrence, dimType As String)
        Dim oTG As TransientGeometry = ThisApplication.TransientGeometry
        Dim oDocDef1 As Document = oOcc1.Definition.Document
        Dim oDocDef2 As Document = oOcc2.Definition.Document

    oWP1 = oDocDef1.ComponentDefinition.WorkPoints.Item(dimPt1)
    oWP2 = oDocDef2.ComponentDefinition.WorkPoints.Item(dimPt2)

        Dim wpProxy1 As WorkPointProxy
        Dim wpProxy2 As WorkPointProxy
        Call oOcc1.CreateGeometryProxy(oWP1, wpProxy1)
        Call oOcc2.CreateGeometryProxy(oWP2, wpProxy2)

        '*** Place Centermarks where the Work Points reside
    Dim oCM1 As Centermark = oSheet.Centermarks.AddByWorkFeature(wpProxy1,
oView)
    Dim oCM2 As Centermark = oSheet.Centermarks.AddByWorkFeature(wpProxy2,
oView)
        oCM1.Visible = False
        oCM2.Visible = False

    '*** Declare variables for dimension endpoints
    Dim oShtPt1 As Point2d = oView.ModelToSheetSpace(oWP1.Point)
    Dim oShtPt2 As Point2d = oView.ModelToSheetSpace(oWP2.Point)

    '*** Create Geometry Intent of the Centermarks
    Dim oGeomIntent1 As GeometryIntent = oSheet.CreateGeometryIntent(oCM1)
    Dim oGeomIntent2 As GeometryIntent = oSheet.CreateGeometryIntent(oCM2)

    '*** Define Dimension Orientation variables
    Dim dimTypeEnum As DimensionTypeEnum = Nothing
    If dimType = "Vertical" Then
        dimTypeEnum = 60163
    ElseIf dimType = "Horizontal" Then
        dimTypeEnum = 60162
    End If
```

```
        '*** Control where the Dimension Text is located
    Dim oDimX As Double = oShtPt1.X - 4
    Dim oDimY As Double = oShtPt1.Y

        Dim oPtText As Point2d = oTG.CreatePoint2d(oDimX, oDimY)

    '*** Create the Dimension
    Dim oDimension As DrawingDimension
    oDimension = 
oSheet.DrawingDimensions.GeneralDimensions.AddLinear(oPtText, oGeomIntent1, 
oGeomIntent2, dimTypeEnum)
    oDimension.CenterText
End Function
```

Sketched Symbols

```
Sub Main
        Dim symbolCount As Integer = 2
        Dim sketchSymbolName As String = "TestSymbol"
        Dim someValue1 As String = "This Works!"
        Dim someValue2 As String = "You Betcha"
        Dim rotation As Double = 0
        Dim scaleFactor As Double = 1
        Dim xValue As Integer = 5
        Dim yValue As Integer = 2

    Dim oDrawDoc As DrawingDocument = ThisApplication.ActiveDocument
    Dim oSketchedSymbolDef As SketchedSymbolDefinition = 
oDrawDoc.SketchedSymbolDefinitions.Item(sketchSymbolName)
    Dim oSheet As Sheet = oDrawDoc.ActiveSheet

    ' This sketched symbol definition contains one prompted string input. An 
array must be input that contains the strings for the prompted strings.
    Dim oPromptStrings(symbolCount - 1) As String
    oPromptStrings(0) = someValue1
        oPromptStrings(1) = someValue2

    Dim oTG As TransientGeometry = ThisApplication.TransientGeometry
    Dim oSketchedSymbol As SketchedSymbol
    oSketchedSymbol = oSheet.SketchedSymbols.Add(oSketchedSymbolDef, 
oTG.CreatePoint2d(xValue, yValue), rotation, scaleFactor, oPromptStrings)
End Sub
```