IM124871

Inventor iLogic: Beyond the Basics

Jon Balgley
Autodesk, Inc.

Learning Objectives

- Write iLogic rules with multiple procedures and functions, and/or other advanced capabilities
- Discover the difference between internal and external iLogic rules
- Learn how to invoke the Inventor API from iLogic rules
- Learn how to find errors in rules, and discover techniques to help minimize errors

Description

You are a proficient user of Inventor software who's looking for ways to make yourself and your colleagues more productive, efficient, and accurate. You've written simple iLogic rules, and now you're ready to go deeper. In this class, you'll learn how to create larger, more sophisticated rules that let you automate entire workflows and models. You'll see how to use iLogic features such as multiple procedures and functions, shared variables, try/catch, and more. You'll understand how to organize the use of multiple rules, and when you need to "update." You'll learn how to make your rules less prone to error. You'll see how to trap and debug errors. You'll learn how to seamlessly integrate the Inventor API and Microsoft VB.NET into your iLogic rules to go beyond the built-in capabilities. Finally, you'll see working examples that show how these techniques can make you dramatically more productive. There will also be some time allocated to reviewing new iLogic capabilities.

Speaker

Jon Balgley has been with Autodesk since 2005, and has worked on CAD-related software since the 1980's. He was one of the original developers of Inventor ETO, and now works on Configurator 360, iLogic, and other forward-looking projects.

Note: Formatted for printing in landscape mode, 11x8.5"

Jon Balgley
Autodesk, Inc.

iLogic: Tee-Shirt Sizes

iLogic usage often comes in three sizes: Small, Medium and Large.

Small usage:

- Human is still involved in the loop; iLogic manages only “nitpicky” stuff e.g., keeping iProperties in sync with parameters, range-checking.
- Very much event-driven / parameter-triggered

Medium usage:

- Essentially creating a single new “command” (which Inventor doesn’t provide OOTB)
- Perhaps deployed as button on a Form

Large usage:

- Human is “out of the loop” for substantial amount of computation / design
- Example: Configuring assemblies
- Necessity of using LODs and component-suppression … unless delving into API … trying to avoid delving *deeply* into API

SMALL usage

Example: Range-checking

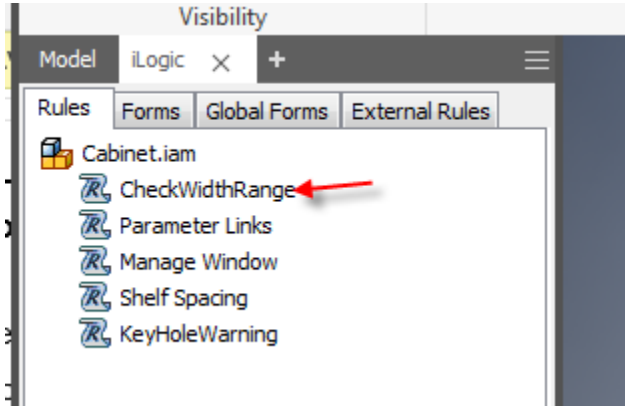
Suppose you want to limit the values for a parameter to a minimum and maximum value. Let's say it's a box, and there are length, width and height parameters.

So first, what does "limit" mean?

- Clamp into range. This is a common technique / tactic, especially where the out-of-range value comes directly from a user (e.g., via an input-box on a form). And/or where the out-of-range value will cause the model to fail in some way.

- Anything else won't clamp the value, and you'll end up with the out-of-range value in the parameter:

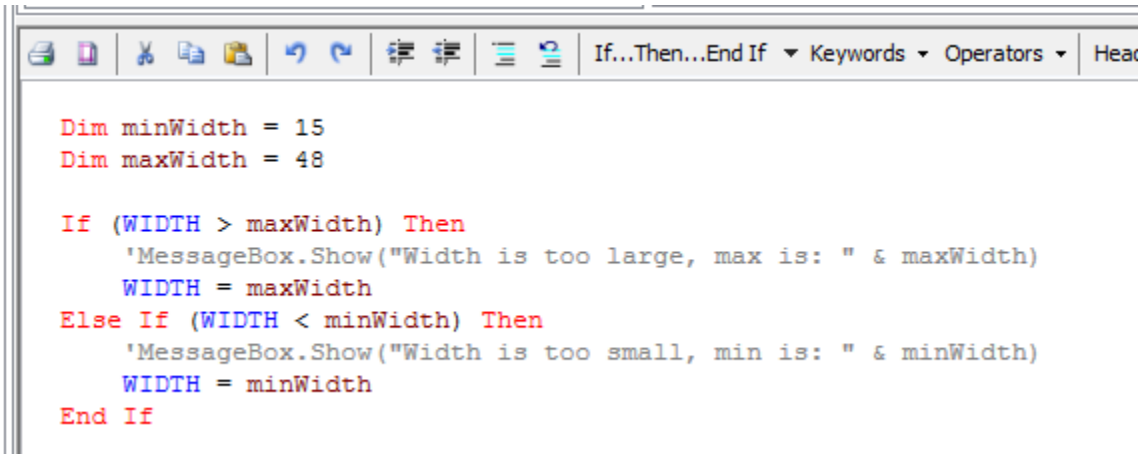
    - Pop up a message. Another common technique, where the out-of-range value only needs a warning, AND there is a user to see it.
    - Rarely, something else?
        - Throw an error (more serious, not that much more valuable than a message). Advantages: Any subsequent iLogic code will stop. Can use try/catch to handle and ignore it. Maybe.
        - log a message to a file (less serious, needs a "system" to log the message)



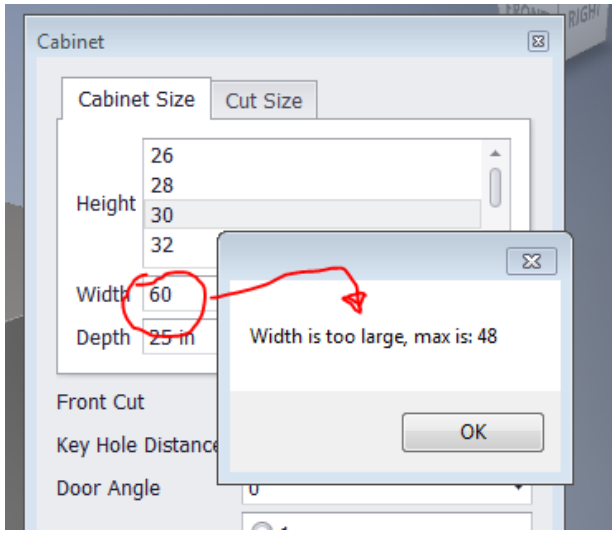
In this rule, depending on what is desired:

- Pop up a warning message (with useful info!)
- Clamp value to min/max
- Or both.

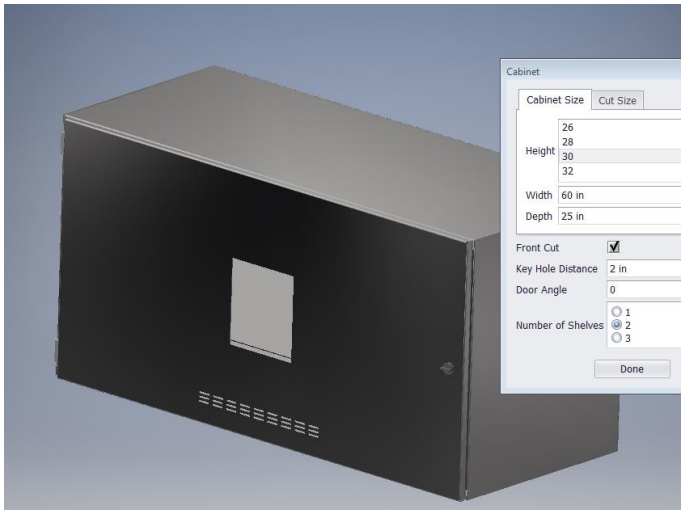
Here it just forces the value to be within range, and then goes ahead:



If commented-out the other way:

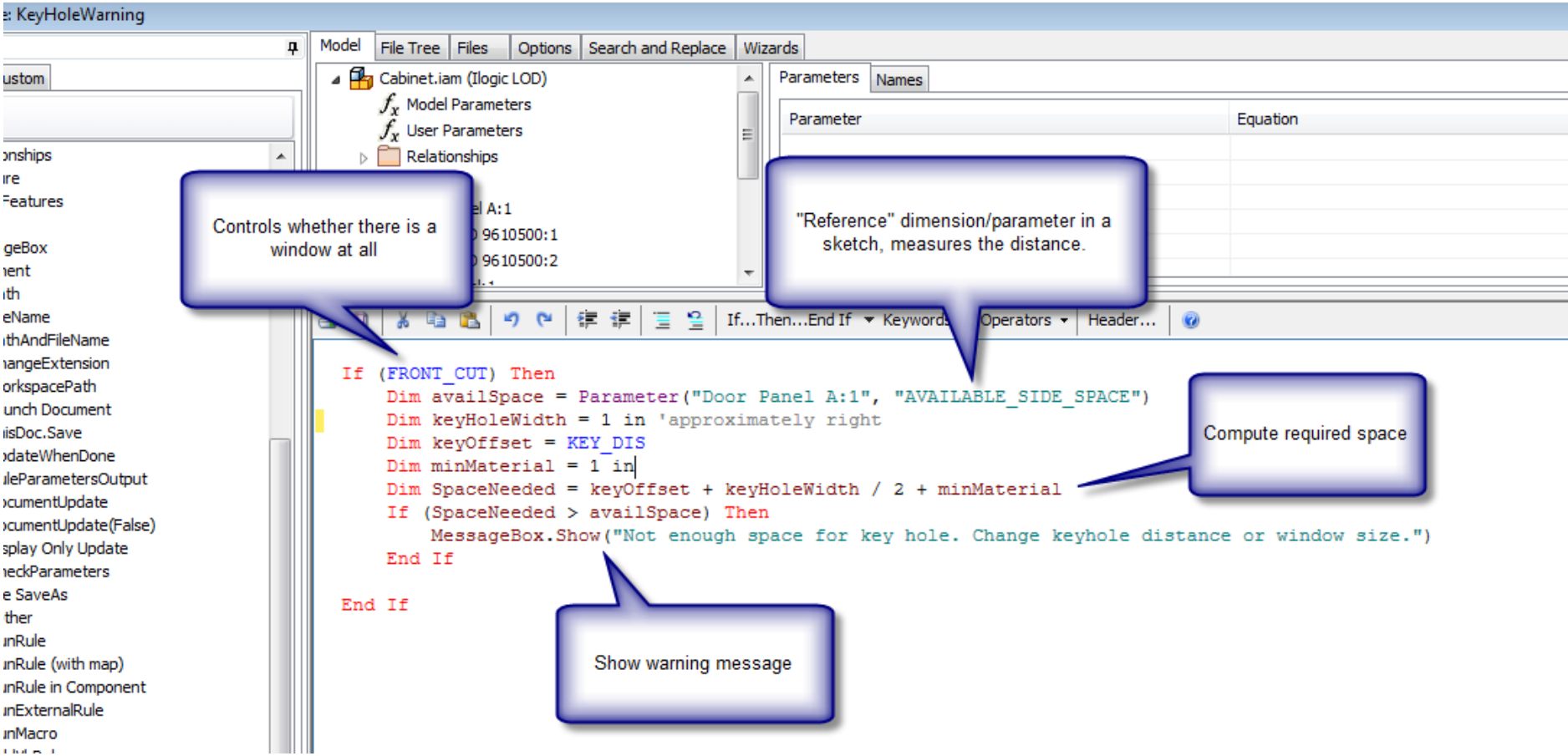


But goes ahead and builds it anyway, with too-large value:



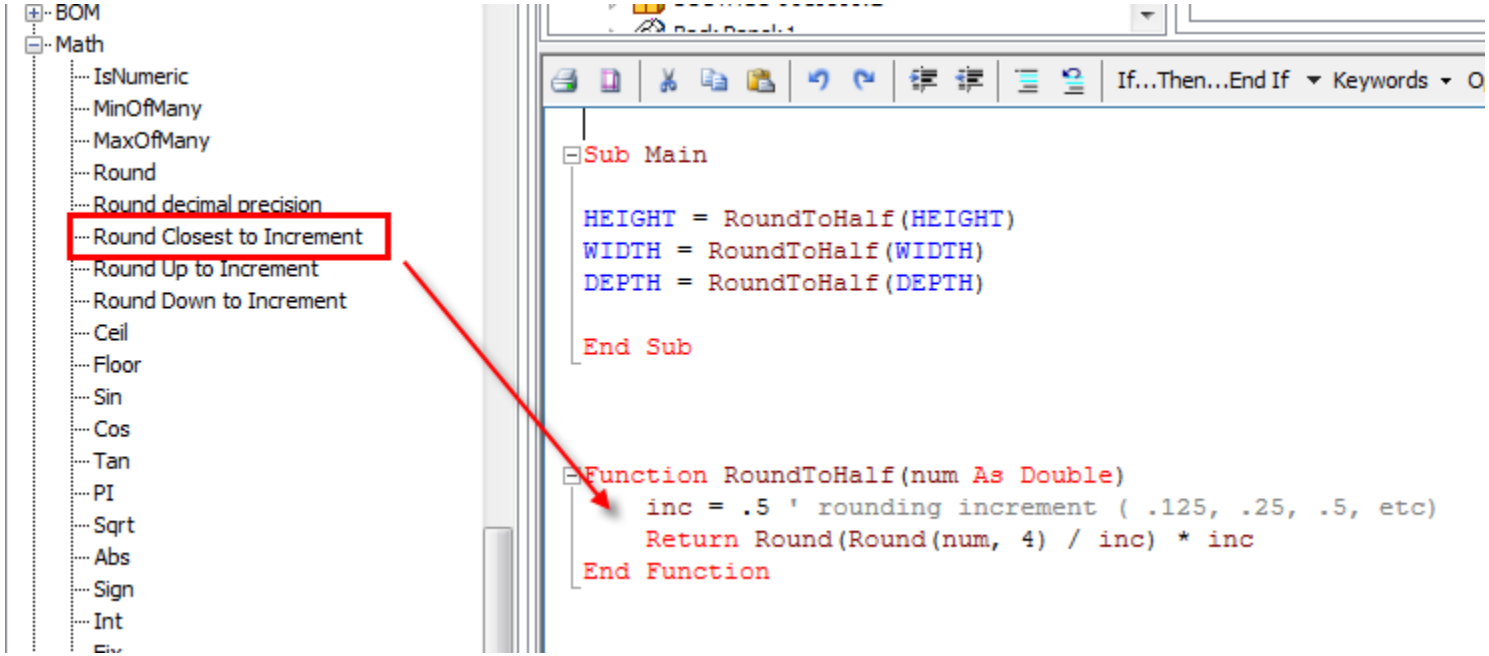
Example: Range-checking with multiple interacting dimensions

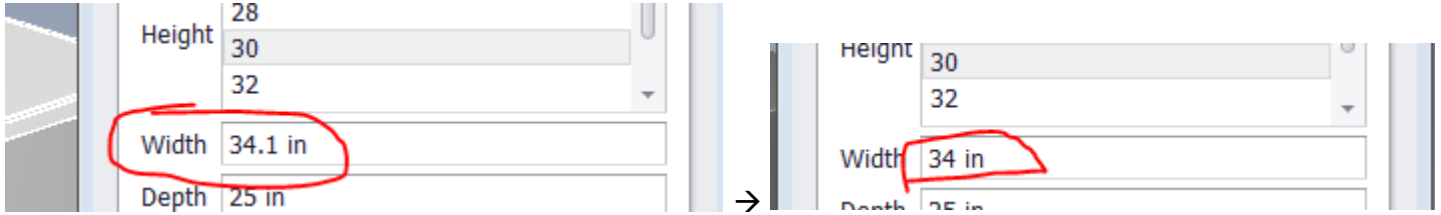
Parametric “cabinet” assembly. Handle and window can interfere with each other, if given “wrong” dimensional-values.


KeyHoleWarning
Model
File Tree
Files
Options
Search and Replace
Wizards
Cabinet.iam (Ilogic LOD)
Model Parameters
User Parameters
Relationships
Parameters
Names
Parameter
Equation
Controls whether there is a window at all
"Reference" dimension/parameter in a sketch, measures the distance.
If...Then...End If
Keywords
Operators
Header...
If (FRONT_CUT) Then
Dim availSpace = Parameter("Door Panel A:1", "AVAILABLE_SIDE_SPACE")
Dim keyHoleWidth = 1 in 'approximately right
Dim keyOffset = KEY_DIS
Dim minMaterial = 1 in
Dim SpaceNeeded = keyOffset + keyHoleWidth / 2 + minMaterial
If (SpaceNeeded > availSpace) Then
MessageBox.Show("Not enough space for key hole. Change keyhole distance or window size.")
End If
End If
Compute required space
Show warning message

Example: Use rules to round-off

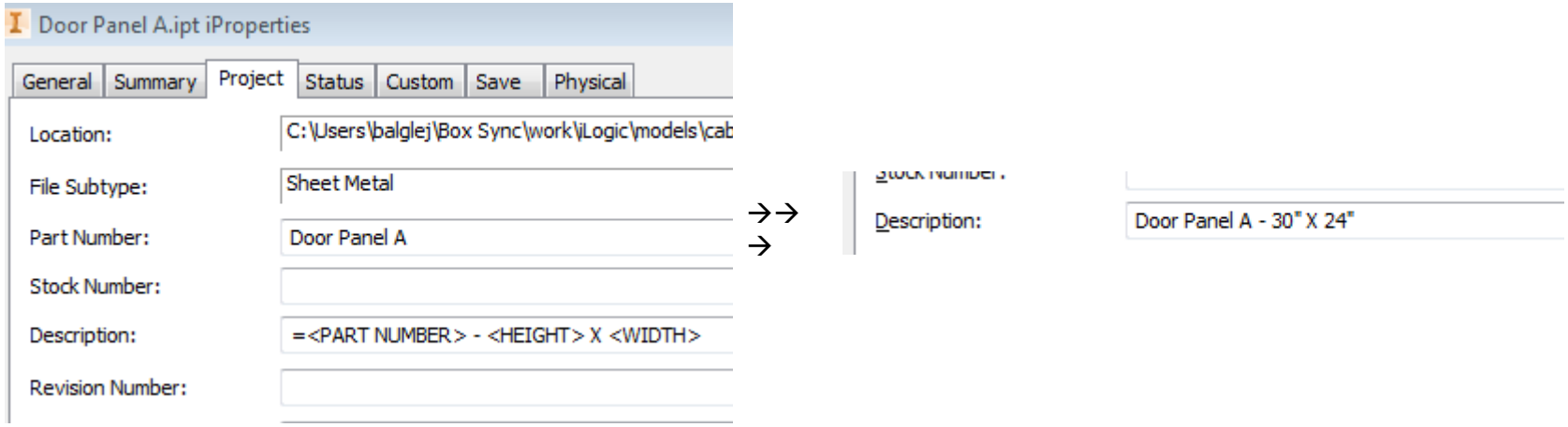
Sometimes you want to expose decimal precision input to the user, but only want certain increments. "Small" rules can help. Here I used a "snippet" to round to closest increment. Then I realized I was doing the same thing 3 times, so I put it into a separate function. This is good practice. "Don't Repeat Yourself" aka "the DRY principle".



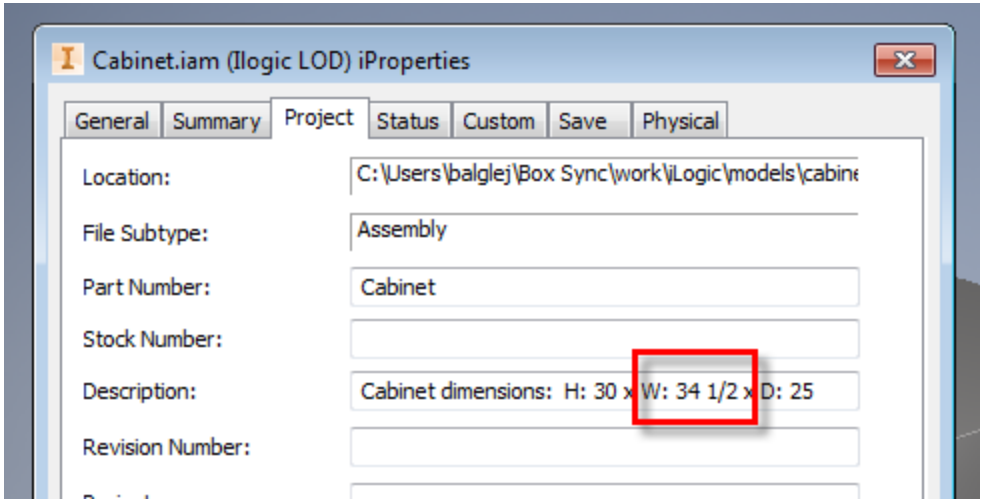


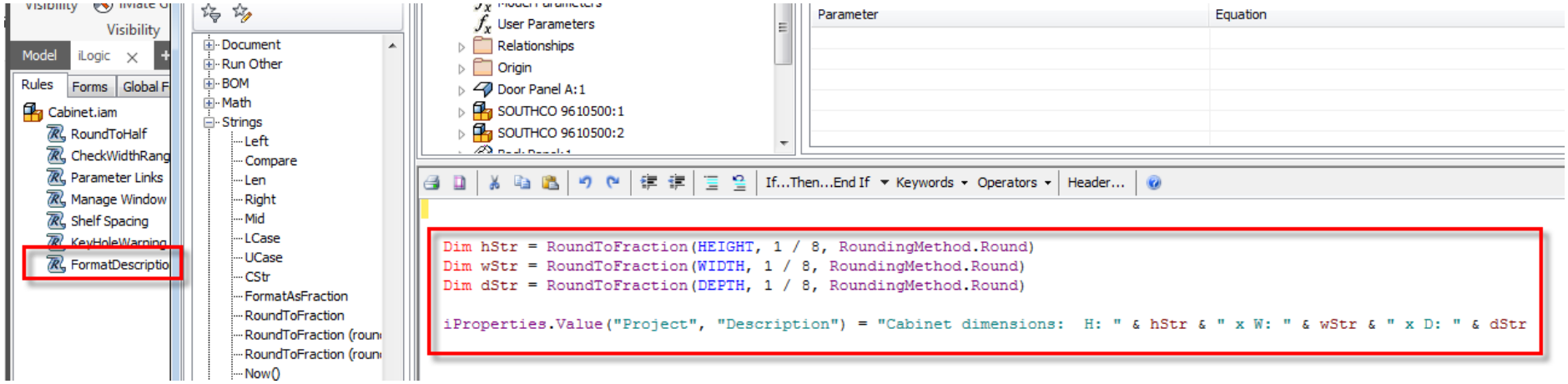
Example: Use "small" rules to keep iProperties in sync with parameter values.

Note that for the simplest changes to 'description' or other iProperties, you don't even need a rule, and can just use the "Formula" option:



But if you want special formatting or other controls, then iLogic comes into play. Here (in a different file) I'm formatting the numbers in the string using fractions, no "inch" symbol, and H/W/D suffixes:



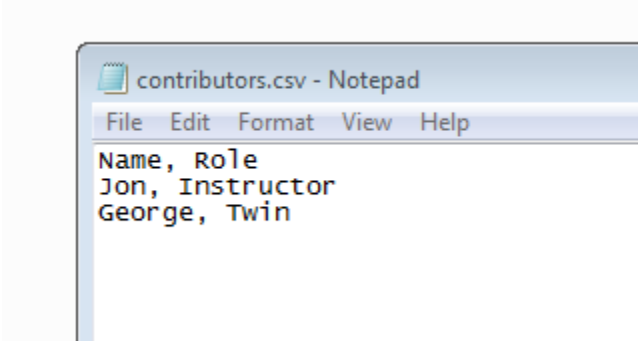


MEDIUM usage

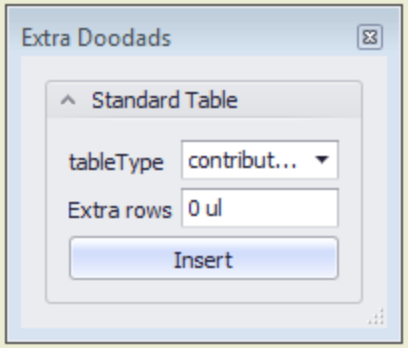
Insert custom table into a drawing

Hey, I have these little tables of data that I need to insert into drawings, sometimes. Wouldn't it be nice to have a special command for that?

Sample file:



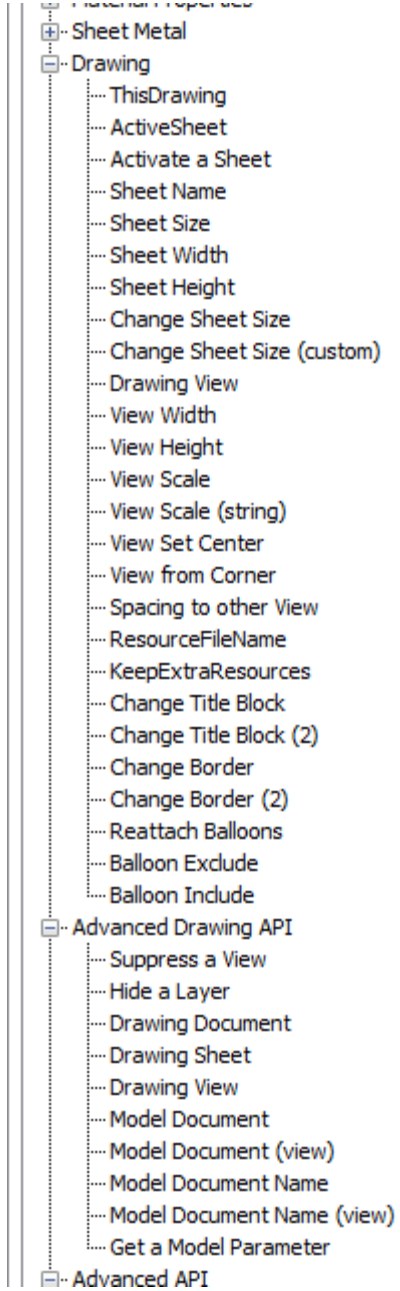
Desired result:



contributors	
Name	Role
Jon	Instructor
George	Twin

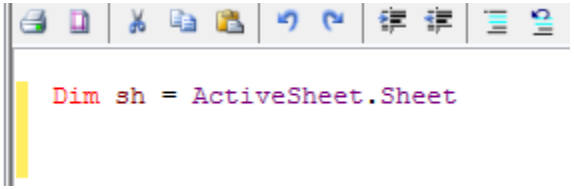
OK, let's try a simple rule to insert a custom table:

Hmm, doesn't seem to be a built-in snippet:

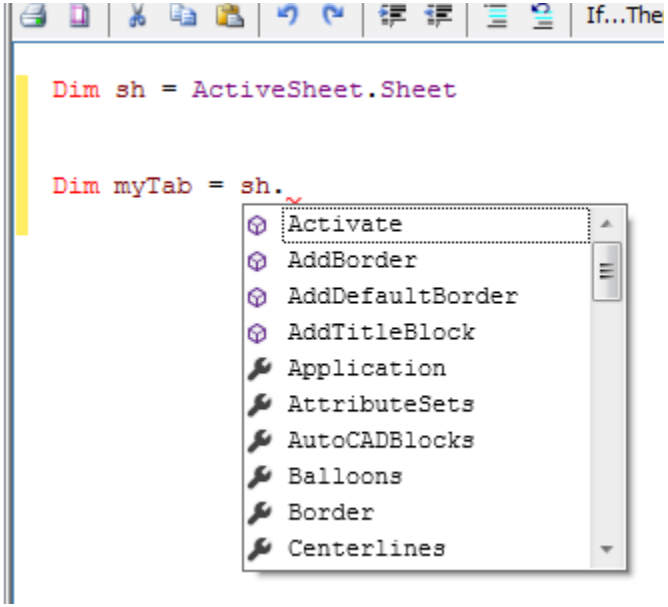


So we're going to have to use the Inventor API: (Unfortunately. There's a lot of complexity in the API. iLogic tries to expose a simpler interface. New iLogic "auto-complete" helps you to navigate the more-complex Inventor API.)

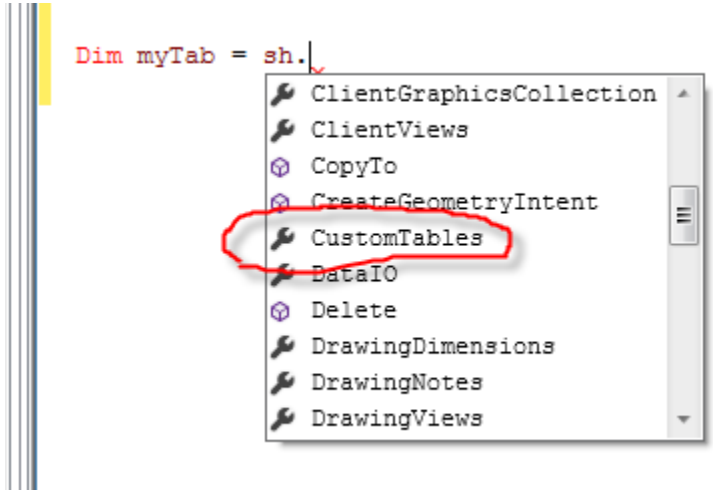
Most iLogic objects have a way to get the underlying Inventor API object:



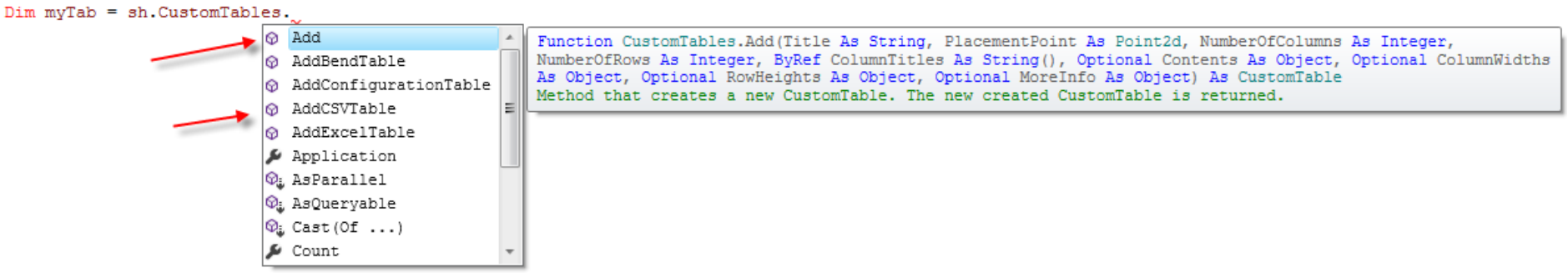
Use auto-complete to help find the way to do it: (press "." to see list of choices)



Found it:



Most collection-properties (which often have a “plural” name) have an “Add” method, so I look for that and it looks good. *But unfortunately I don’t notice the “AddCSVTable” which would be better*. So go ahead with the Add.



Hmm, second argument is a “Point 2d”:

```
Dim myTab = sh.CustomTables.Add("Test", )
```

```
Function CustomTables.Add(Title As String, PlacementPoint As Point2d, Numb
ColumnTitles As String(), Optional Contents As Object, Optional ColumnWidths A
As Object) As CustomTable
Method that creates a new CustomTable. The new created CustomTable is returnec
PlacementPoint: Input Point2d object that defines the placement point of the t
```

So I need to make one of those. I’ll put it in the center of the sheet, then drag it afterwards.

“Transient” geometry doesn’t appear anywhere in the model. These objects are just used temporarily in programming the API. We need a 2D-point so we’ll use CreatePoint2D:

```
Dim sh = ActiveSheet.Sheet

Dim pt = ThisApplication.TransientGeometry.

Dim myTab = sh.CustomTables.Add("Test",
```

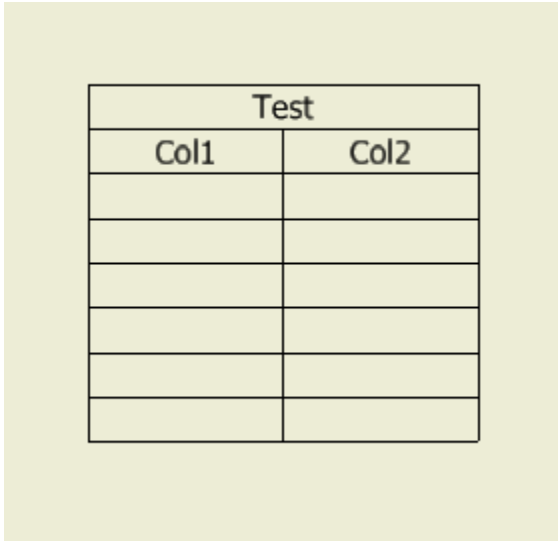
```
CreateOrientedBox
CreatePlane
CreatePlaneByThreePoints
CreatePoint
CreatePoint2d
CreatePolyline2d
CreatePolyline2dFromCurve
CreatePolyline3d
CreatePolyline3dFromCurve
CreateSphere
```

```
Dim sh = ActiveSheet.Sheet

Dim pt = ThisApplication.TransientGeometry.CreatePoint2d(sh.Width/2, sh.Height/2)

Dim myTab = sh.CustomTables.Add("Test", pt, 2, 6)
```

```
Function CustomTables.Add(Title As String, PlacementPoint As Point2d, NumberOfColumns As Intege
ColumnTitles As String(), Optional Contents As Object, Optional ColumnWidths As Object, Optional R
As Object) As CustomTable
Method that creates a new CustomTable. The new created CustomTable is returned.
NumberOfRows: Input Long that specifies the number of rows in the table.
```

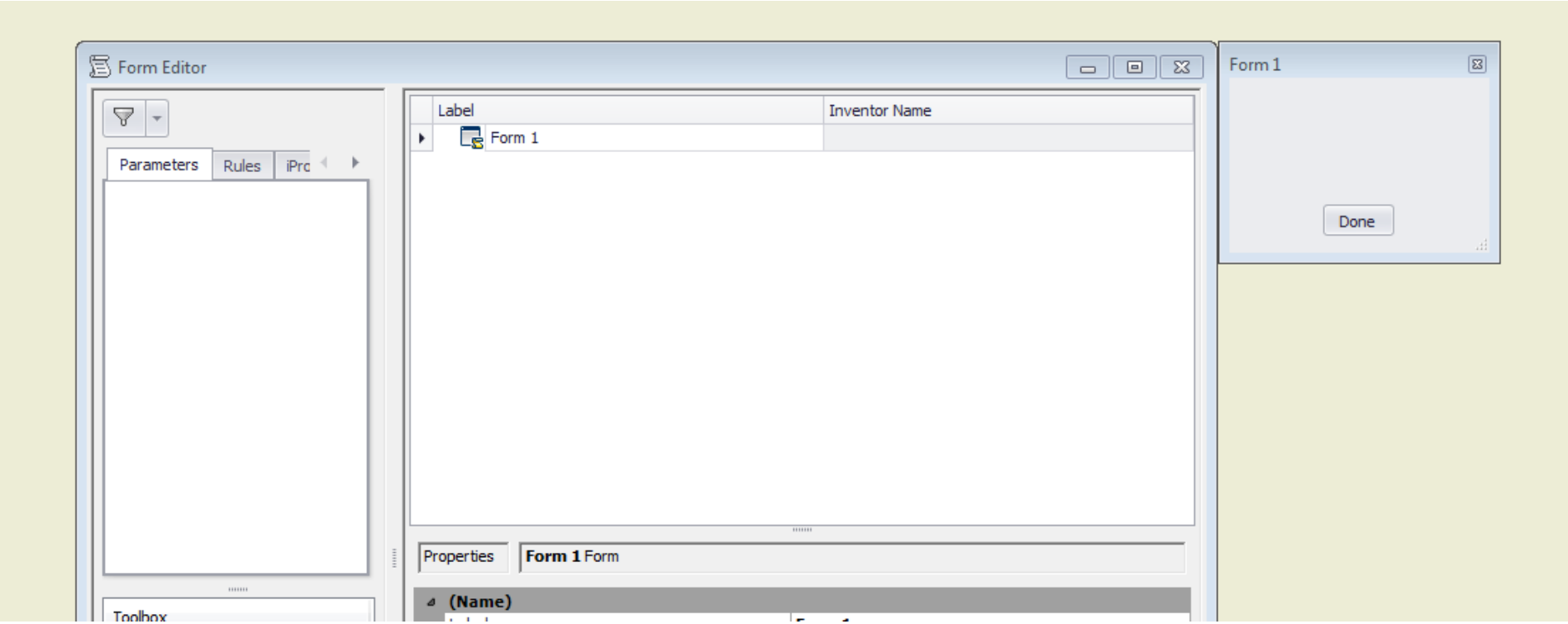
```
Dim sh = ActiveSheet.Sheet

Dim pt = ThisApplication.TransientGeometry.CreatePoint2d(sh.Width/2, sh.Height/2)

Dim myTab = sh.CustomTables.Add("Test", pt, 2, 6, {"Col1", "Col2"})
```

Save&Run … and, voila:



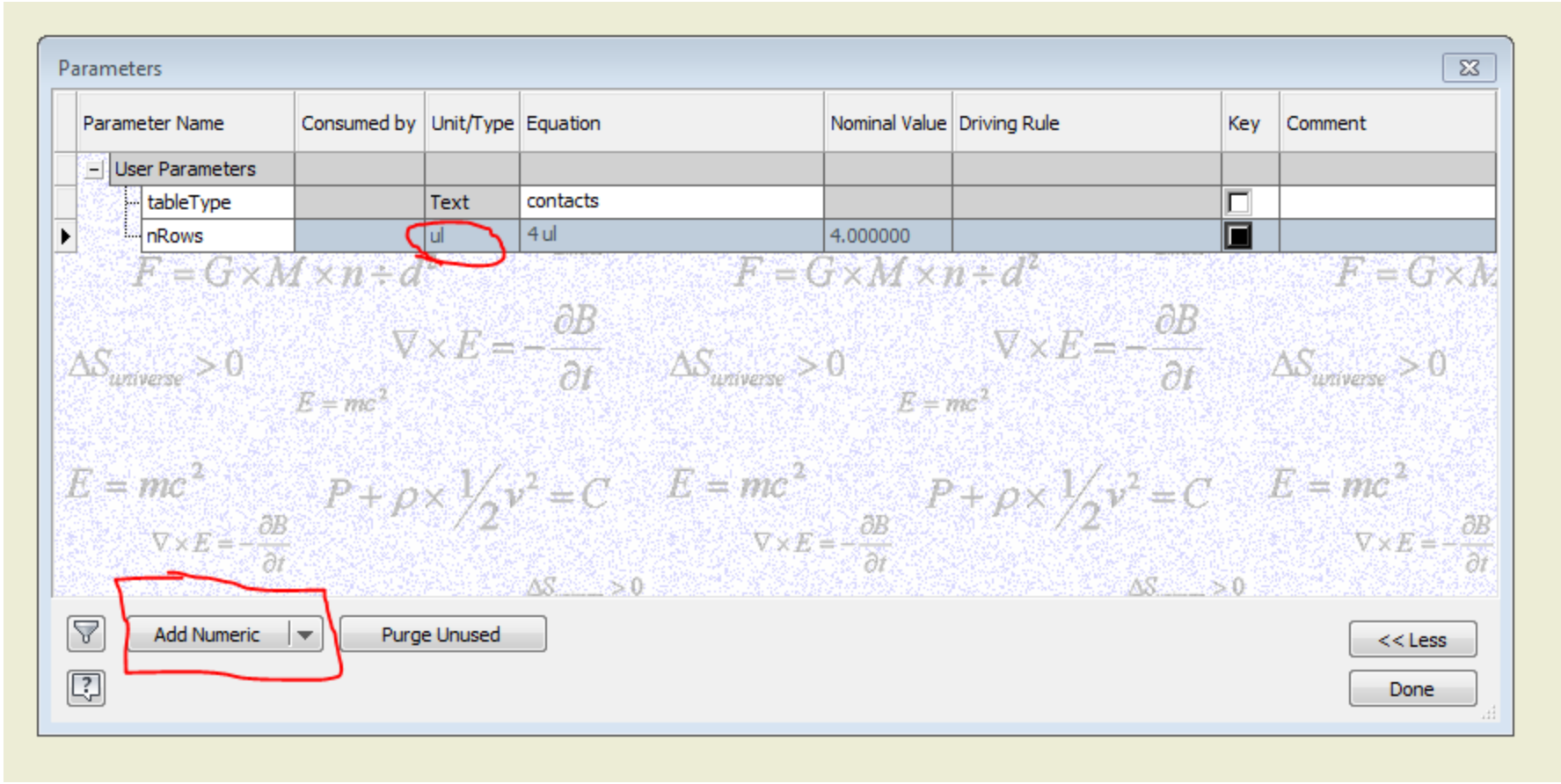
OK, cool, let's make a form to help:

Do "Add Form" and…



Huh. No parameters to use on the form. Duh.

OK, let's add some parameters:

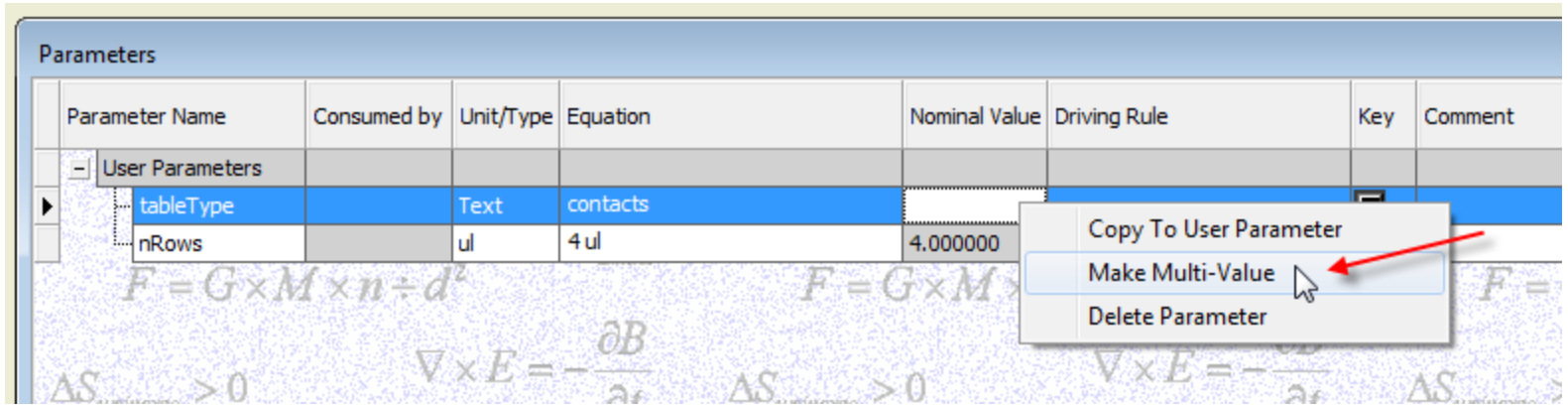


Note that the “tableType” parameter is a “Text” parameter. Under the “Add Numeric” there are options to “Add Text” and “Add True/False”.

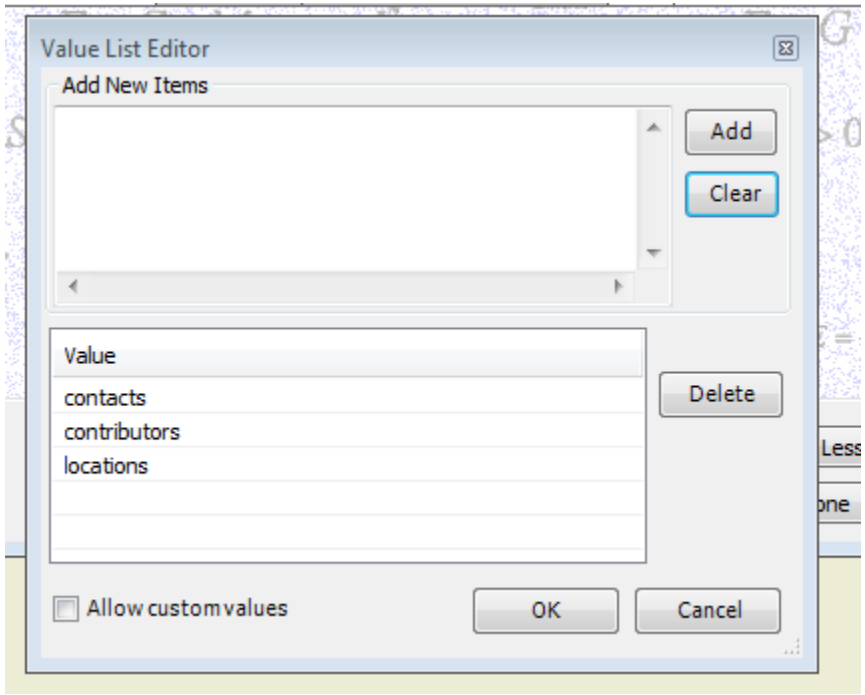
The nRows parameter is numeric, but set to be “ul” (unitless … integer).

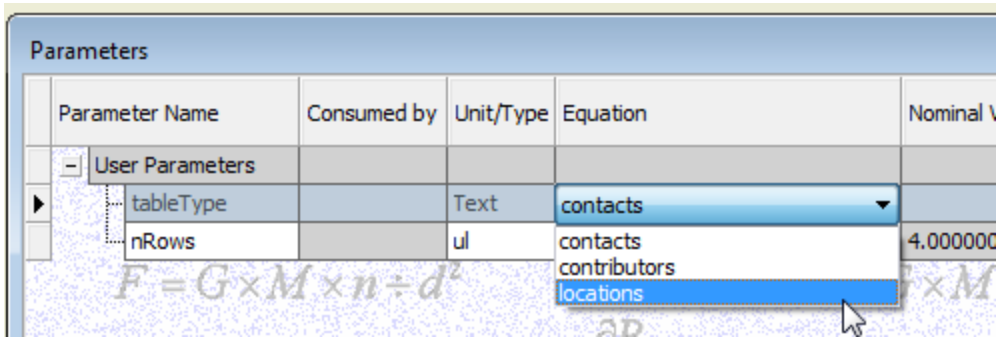
Note that the intention is for “tableType” to select from one of a handful of “standard” tables that I need to insert.

Well, I need to be able to enumerate the several possible table-types. So make it be a “Multi-Value” parameter, by right-clicking:

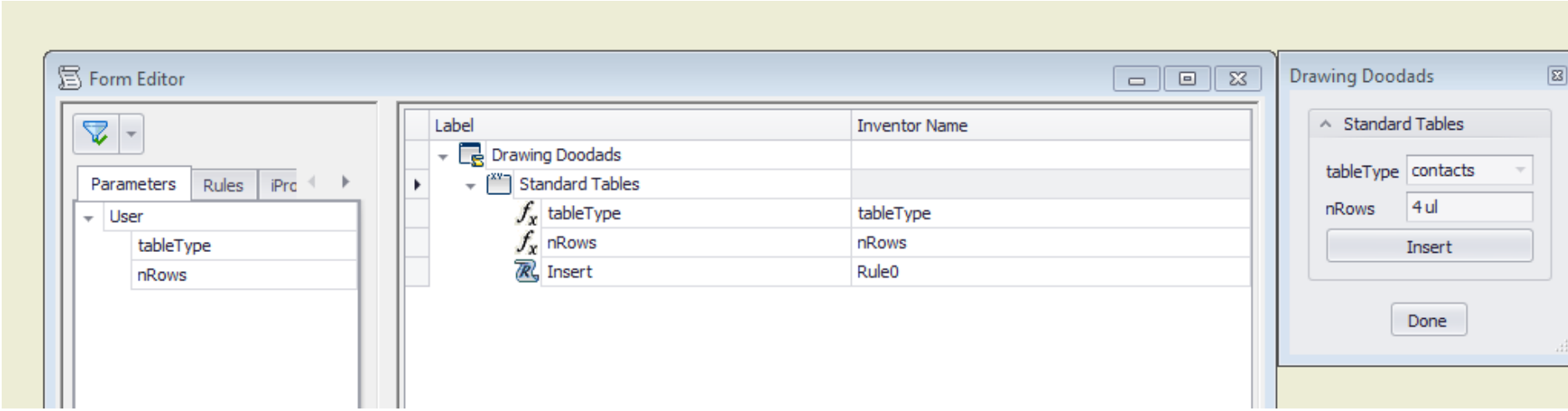


And provide the alternative values:





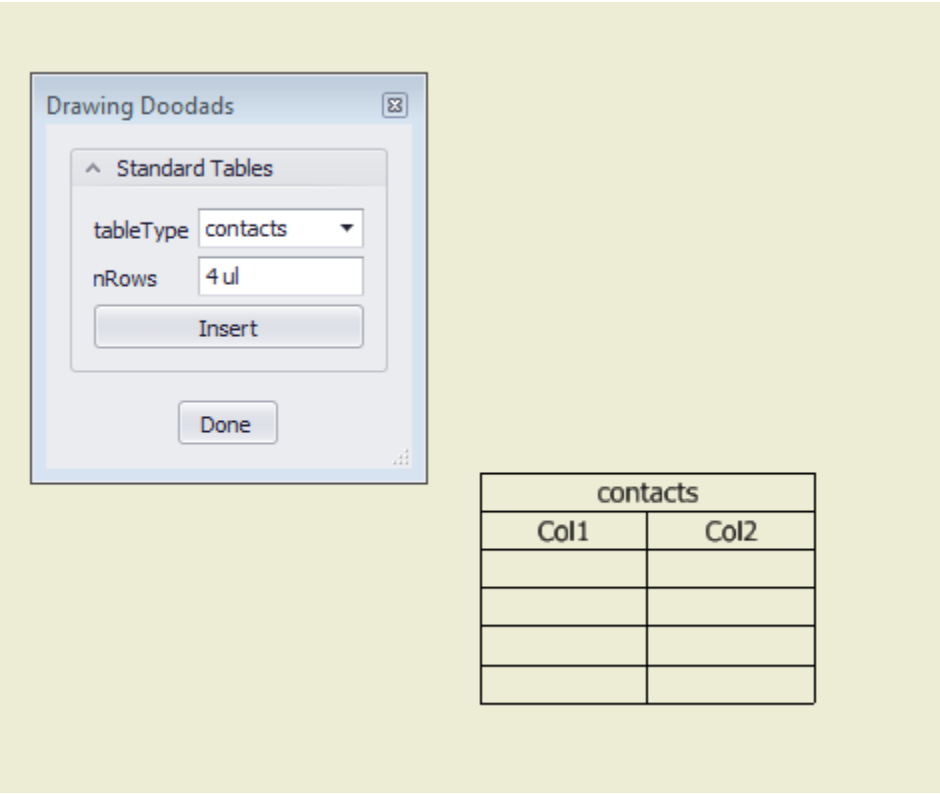
And now we can make the form:



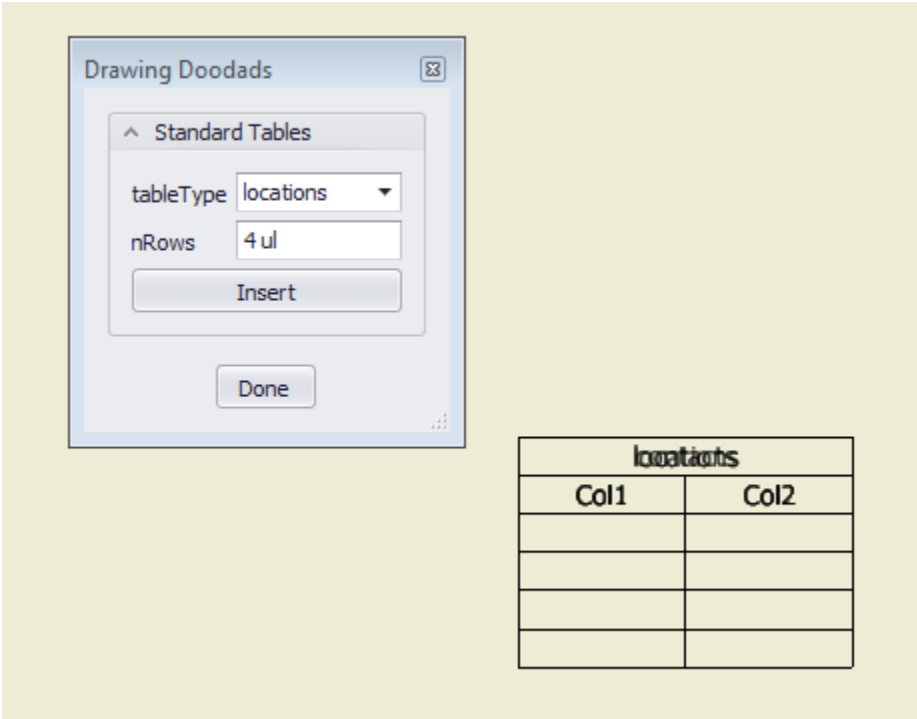
And update the rule to use the parameters:

```
Dim sh = ActiveSheet.Sheet

Dim pt = ThisApplication.TransientGeometry.CreatePoint2d(sh.Width/2, sh.Height/2)

Dim myTab = sh.CustomTables.Add(tableType, pt, 2, nRows, {"Col1", "Col2"})
```

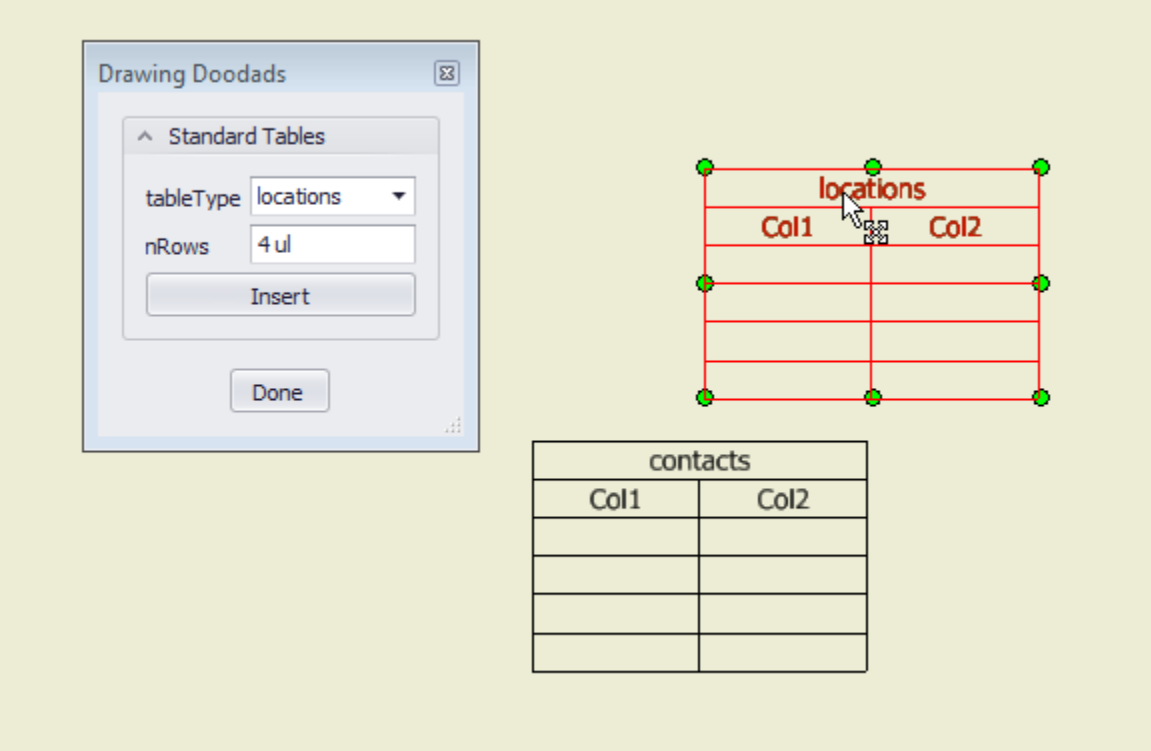
Seems to work!!!



But … when I change the tableType … something weird happens:



Huh! There's two tables!



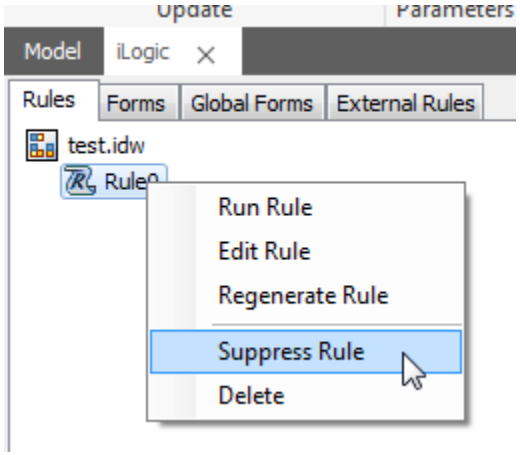
And then I remember … I didn't even click the "Insert" or "Done" buttons… hmm... must mean that my rule is firing, so I look at it:

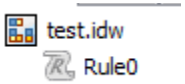
Here's the rule:

```
Dim sh = ActiveSheet.Sheet

Dim pt = ThisApplication.TransientGeometry.CreatePoint2d(sh.Width/2, sh.Height/2)

Dim myTab = sh.CustomTables.Add(tableType, pt, 2, nRows, {"Col1", "Col2"})
```

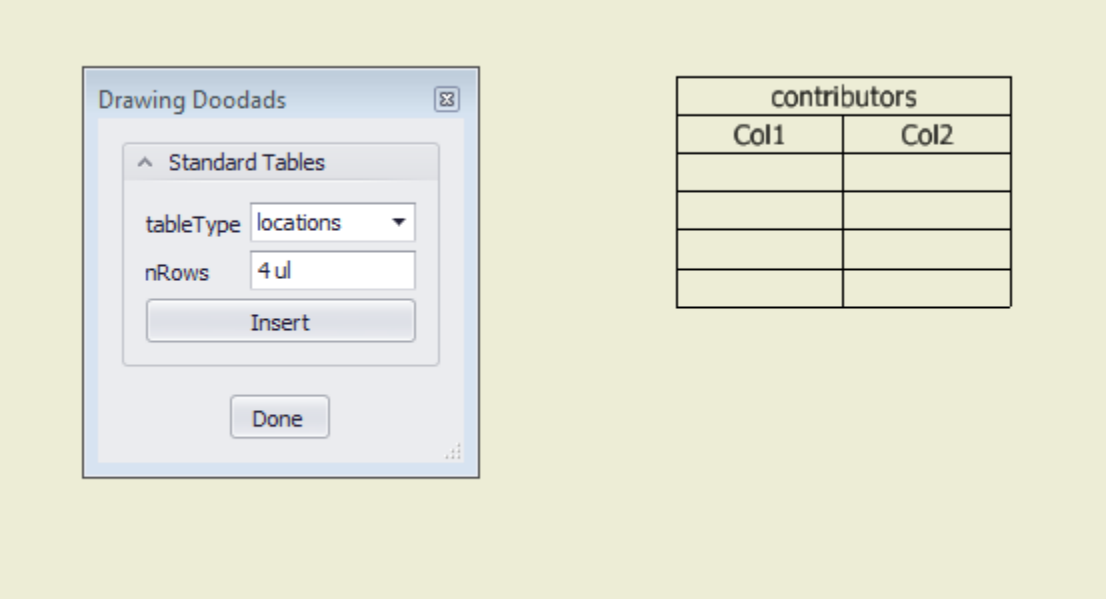
Oh yeah! Blue means "parameter", and iLogic fires the rule if any referenced parameter is changed! So every time I change one of these parameters, *I'm getting another table!!!*

So … easy solution. Since I only want this rule to run when I click the "Insert" button, I can just "suppress" the rule:





Now I test the form again:

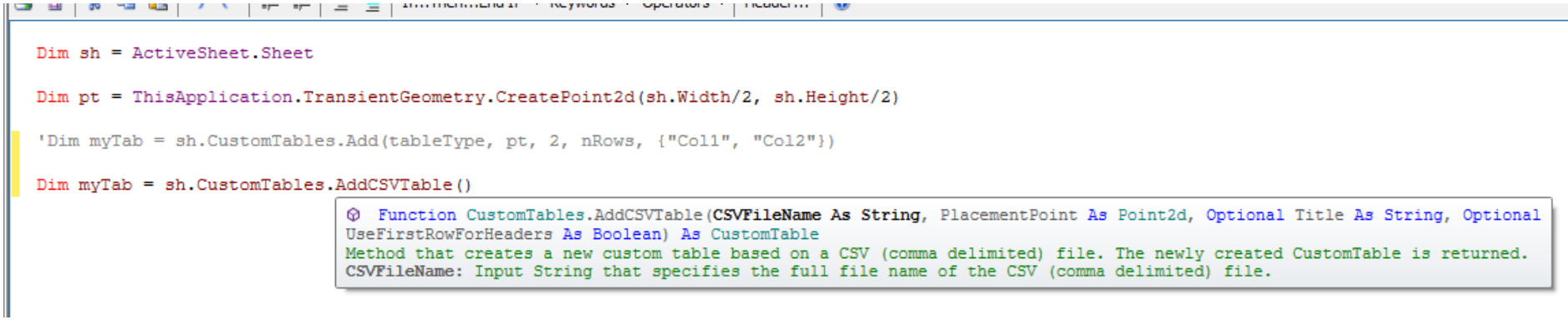


…and now it only inserts the new table when I click the “Insert” button. Merely changing the tableType does not trigger the rule. Yay!

Let’s save the file now (this is a key contributor to a later problem).

But really, I want it to insert the contents of the CSV file, so eventually I find AddCSVTable.

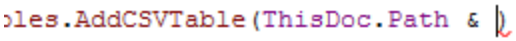
OK, it needs some different arguments:



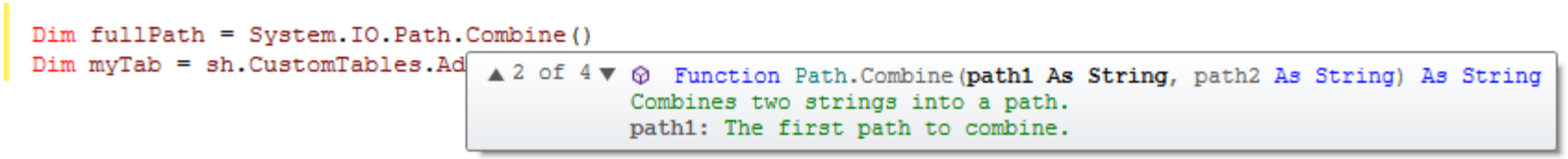
…and the first one is the filename. So where do I get the file from?

- Let's start by assuming the CSV files are in a subfolder under the current drawing's location.
- …And that they have the same names as the "tableTypes" parameter-value possibilities.

We can use "ThisDoc.Path" plus use the "&" string-concatenation operator:



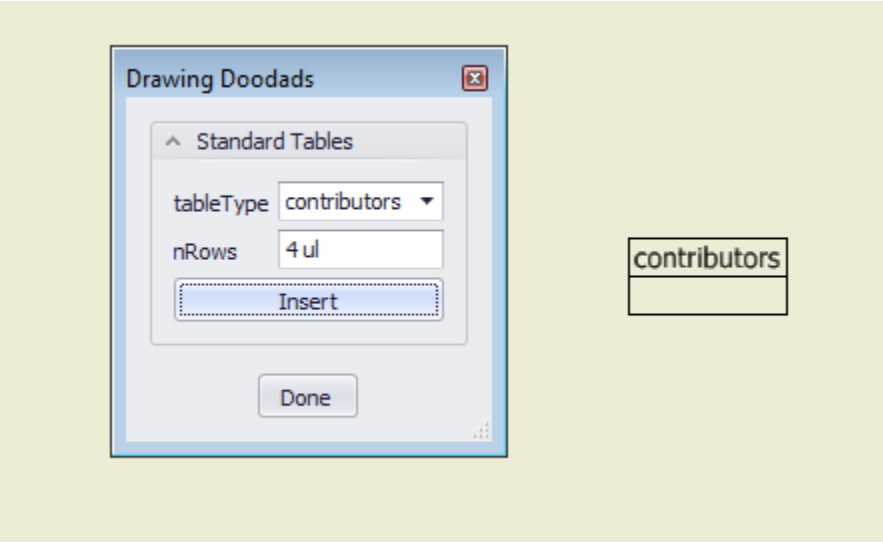
But, shoot, does ThisDoc.Path end in a "\" or not? I can never remember… So let's use the .Net method designed for this purpose:



OK, that's better:

```
Dim sh = ActiveSheet.Sheet

Dim pt = ThisApplication.TransientGeometry.CreatePoint2d(sh.Width/2, sh.Height/2)

'Dim myTab = sh.CustomTables.Add(tableType, pt, 2, nRows, {"Col1", "Col2"})

Dim fullPath = System.IO.Path.Combine(ThisDoc.Path, tableType & ".csv")
Dim myTab = sh.CustomTables.AddCSVTable(fullPath, pt, tableType, True)
```

Except it doesn't quite work:



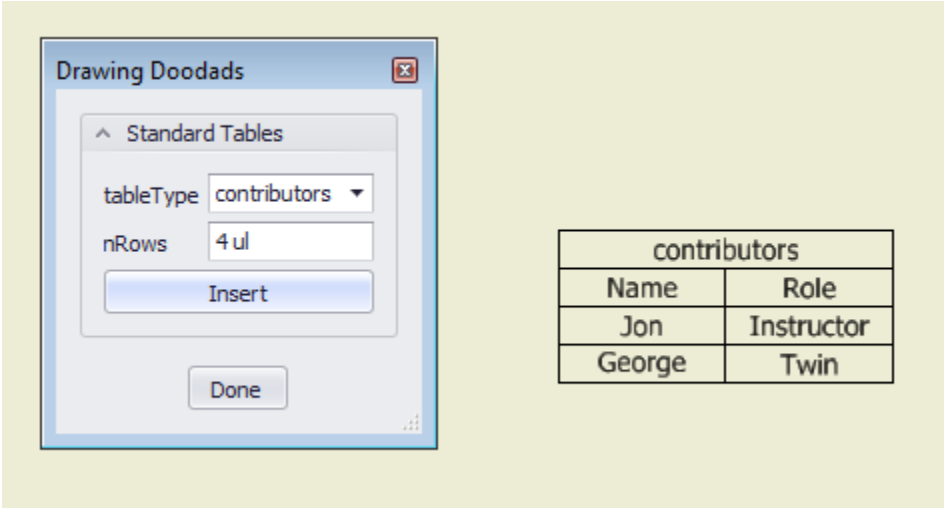
OK, it's not getting the content of the file, so … maybe I got the path wrong? Oops, yes, forgot the sub-folder in the path. So fix that stupid mistake:

```
Dim fullPath = System.IO.Path.Combine(ThisDoc.Path, "standardTables\" & tableType & ".csv")
Dim myTab = sh.CustomTables.AddCSVTable(fullPath, pt, tableType, True)
```

Or use a different variation on "Combine":

```
Dim fullPath = System.IO.Path.Combine(ThisDoc.Path, "standardTables\", tableType & ".csv")
```

So now it kinda works (does seem slow, though):

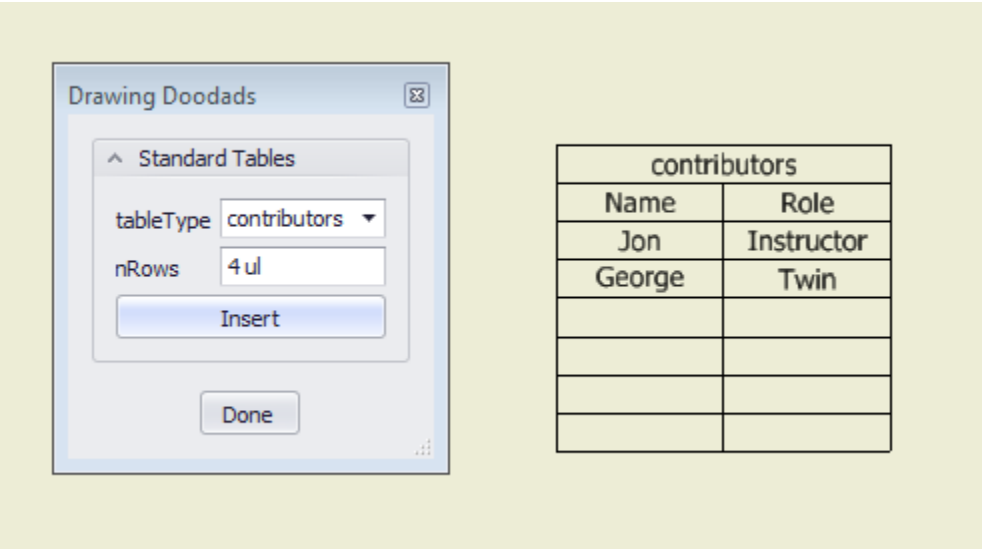


contributors	
Name	Role
Jon	Instructor
George	Twin

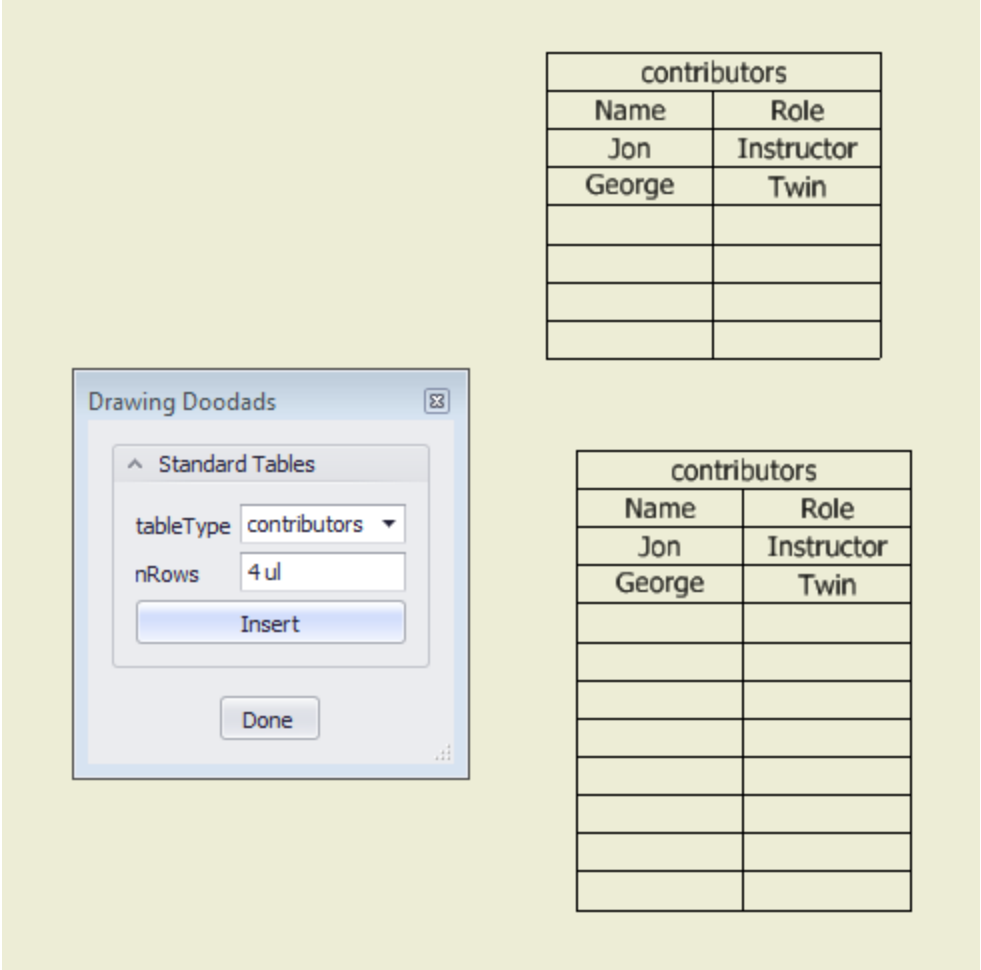
But it didn't add the extra rows. So I guess I have to add them manually:

```
Dim fullPath = System.IO.Path.Combine(
Dim myTab = sh.CustomTables.AddCSVTabl

For i = 1 To nRows
    myTab.Rows.Add()
Next i
```

So I delete the old table, and re-run it, and that seems to work! Yay!



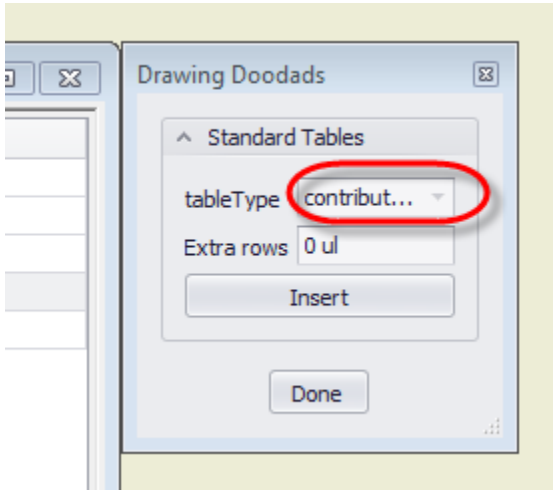
But then I move the first table out of the way, and try it again, and something weird happens:



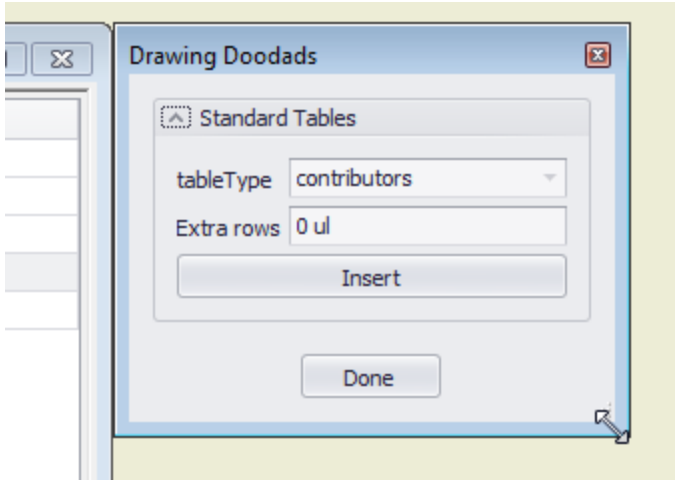
Well, after some investigation, it turns out that there should only be one table per CSV file. The table contains a link to the CSV file, and will stay up-to-date. And really, after some thought, that's what I want: (1) there should only be one table, at most, for any CSV file, and (2) the "rows" parameter really just controls some "extra blank" rows after data from the file, so that should usually just be zero.

It's easy to fix the second problem, just change the default value of the "nRows" parameter to be zero, and change the label in the form.

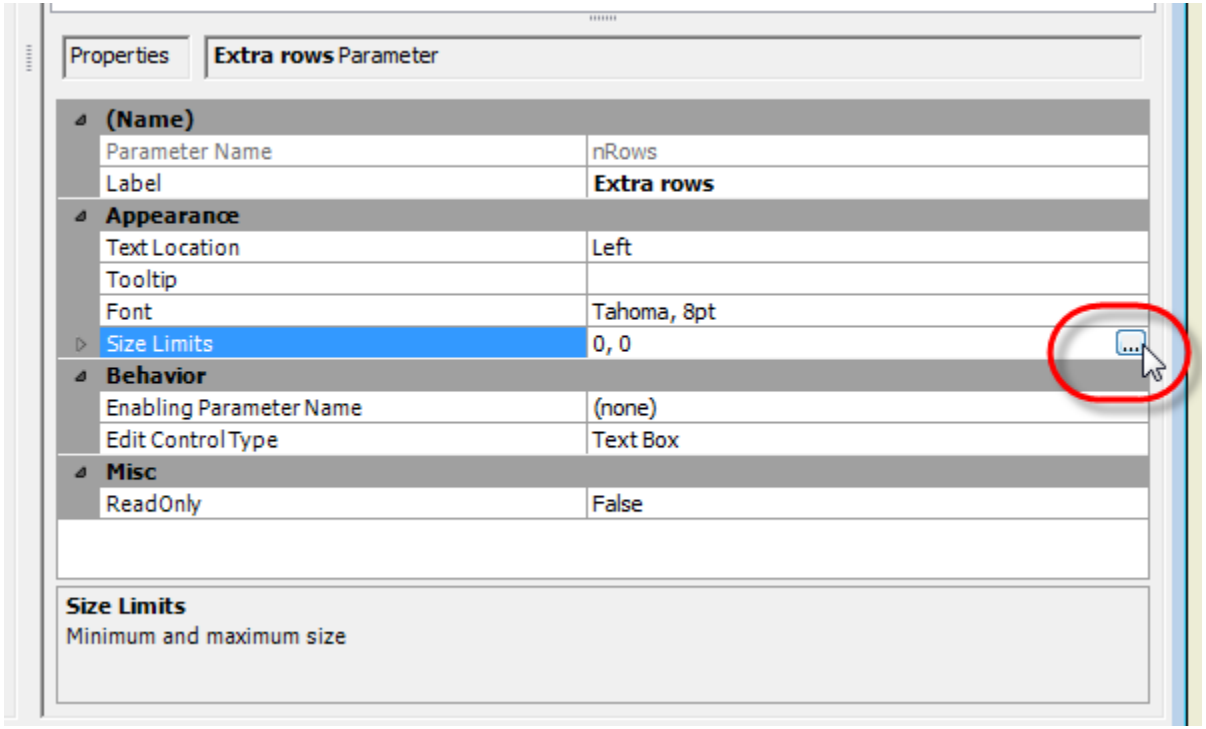
Well, turns out not to be so easy to change the label, because the longer text now truncates the tableType values:



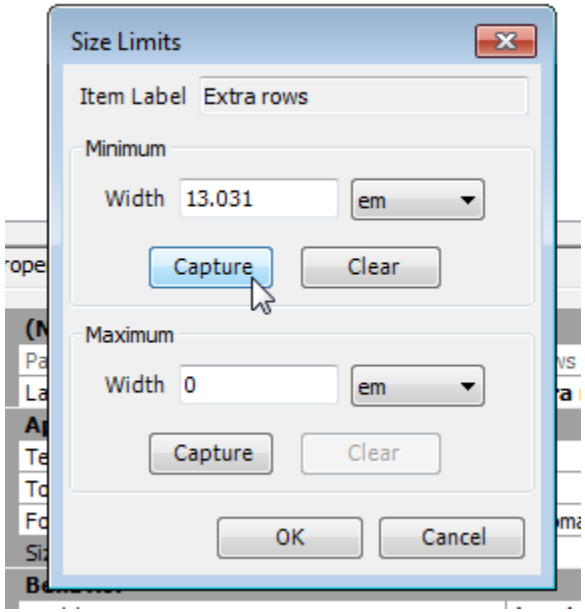
So we can use the form-editor to add some width. First, adjust the size of the form (the preview):



Then click on the nRows parameter, and on the “Size Limits” option, and the little popup button on the right:



Then click on “Capture” on the minimum width:

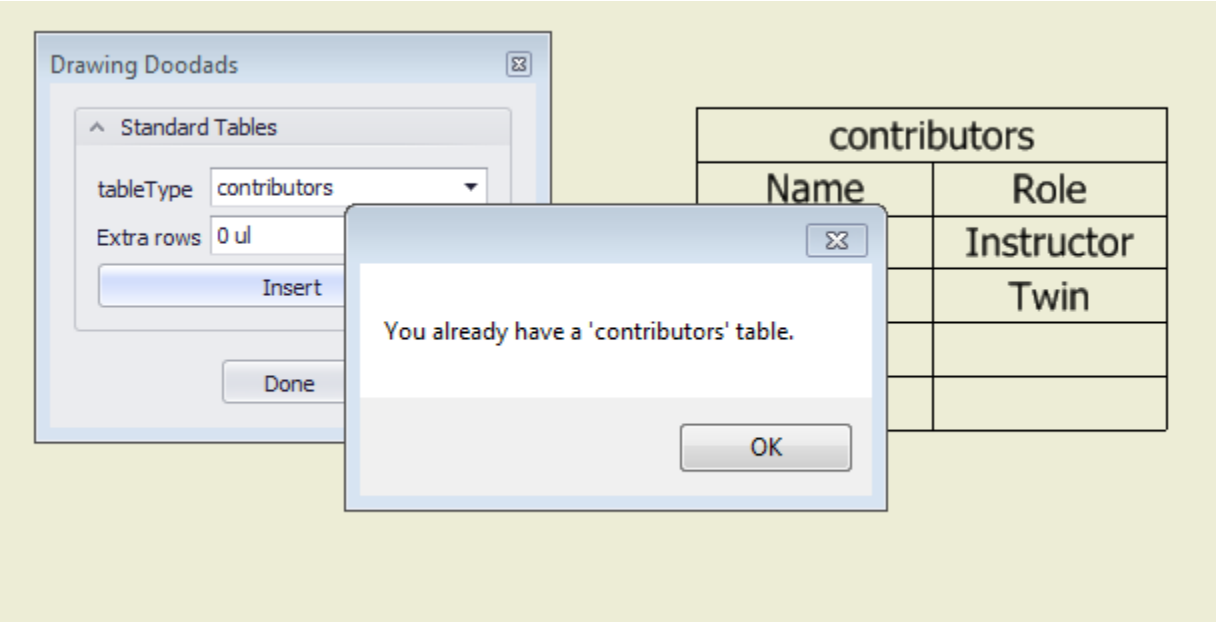


OK, that was the easy part. Now the hard part is … when we (attempt to) insert one of these tables, we have to check to see if we already have “it” inserted.

So we need to iterate through the existing tables. That part is easy. Here's the basic idea:

```
Dim sh = ActiveSheet.Sheet

Dim pt = ThisApplication.TransientGeometry.CreatePoint2d(sh.Width/2, sh.Height/2)

'Dim myTab = sh.CustomTables.Add(tableType, pt, 2, nRows, {"Col1", "Col2"})

Dim fullPath = System.IO.Path.Combine(ThisDoc.Path, "standardTables\" & tableType & ".csv")

For Each cusTab In sh.CustomTables
    Dim something As Boolean = True
    If (something) Then
        MessageBox.Show("You already have a '" & tableType & "' table.")
        Exit Sub
    End If
Next cusTab

Dim myTab = sh.CustomTables.AddCSVTable(fullPath, pt, tableType, True)

For i = 1 To nRows
    myTab.Rows.Add()
Next i
```

And we can test that (note that logic as-is requires that there be at least one custom table already, for the loop to run at least once, for testing):

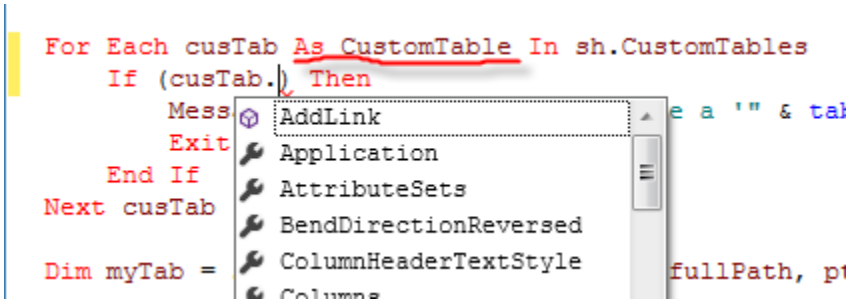


OK, so one easy way to check is by the title of the tables. So let's change the code a little:

```
For Each cusTab In sh.CustomTables
    If (cusTab.) Then
        Mess ToString You already
        Exit Sub
    End If
Next cusTab
```

But hmm, the auto-complete isn't working?!?!?

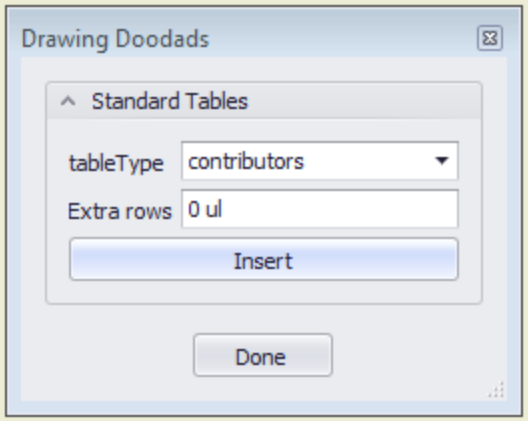
That's because it doesn't know the datatype of the variable. We can fix that:



Then it's easy to write it like this:

```
Dim sh = ActiveSheet.Sheet

Dim pt = ThisApplication.TransientGeometry.CreatePoint2d(sh.Width/2, sh.Height/2)

'Dim myTab = sh.CustomTables.Add(tableType, pt, 2, nRows, {"Col1", "Col2"})

Dim fullPath = System.IO.Path.Combine(ThisDoc.Path, "standardTables\" & tableType & ".csv")

For Each cusTab As CustomTable In sh.CustomTables
    If (cusTab.Title = tableType) Then
        MessageBox.Show("You already have a '" & tableType & "' table.")
        Exit Sub
    End If
Next cusTab

Dim myTab = sh.CustomTables.AddCSVTable(fullPath, pt, tableType, True)

For i = 1 To nRows
    myTab.Rows.Add()
Next i
```

And it works … except … the user can change the title. If the user changes the title even slightly, then it doesn't catch that.

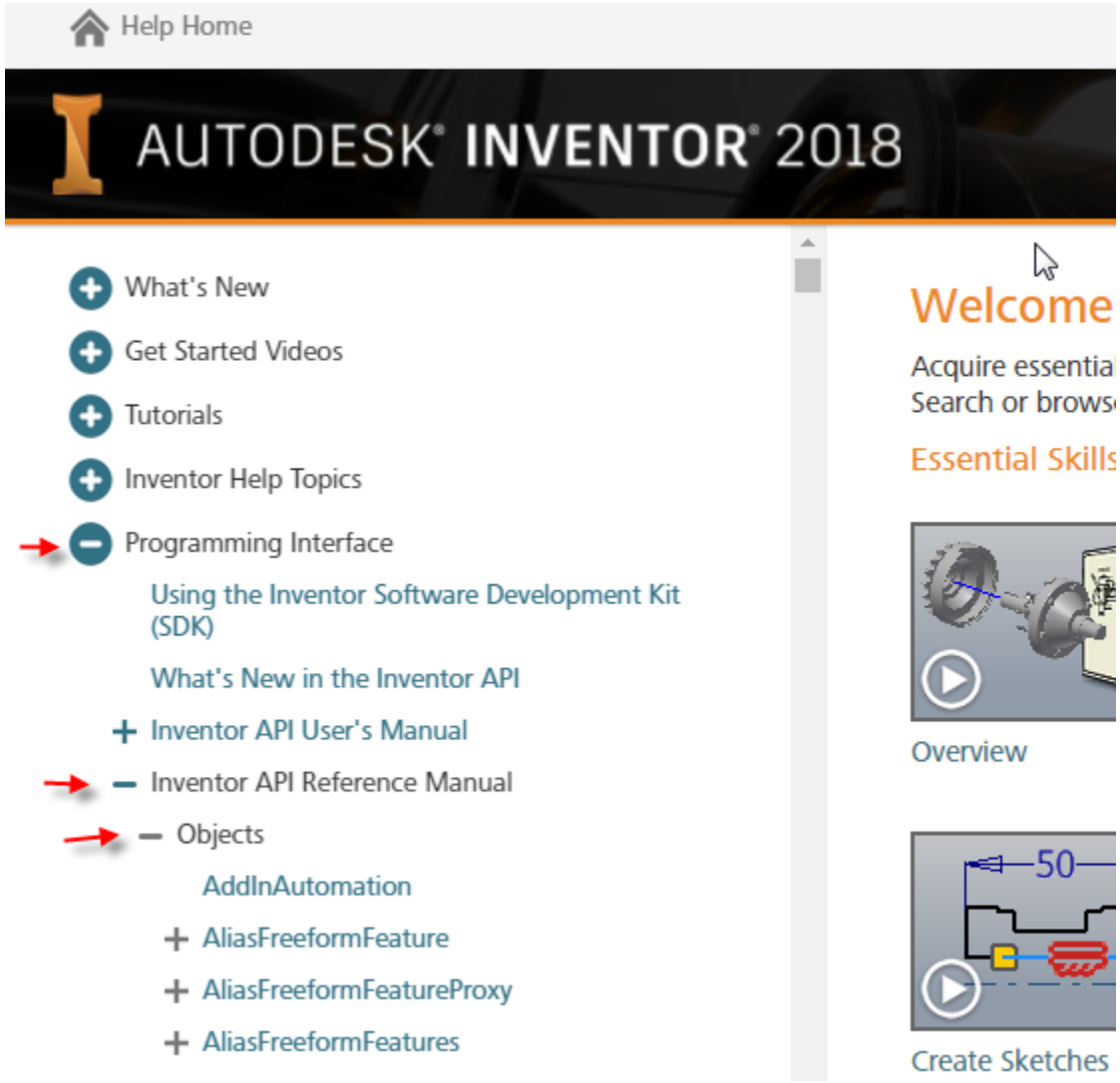


Contributors	
Name	Role
Jon	Instructor
George	Twin

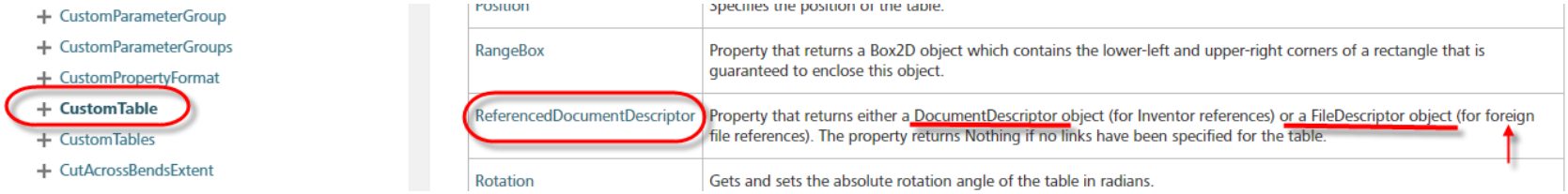
contributors	
Name	Role
Jon	Instructor
George	Twin

So let's try a different strategy. Each table has a reference to the underlying data file, so let's find that and use that for comparison.

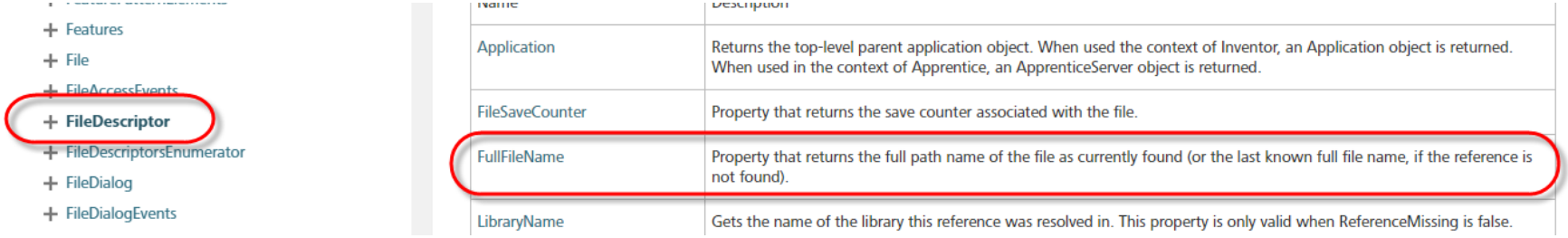
Let's try the Inventor help:



Search for “CustomTable” … it’s easy because it’s in alphabetical order. Then I read through the properties and find this:

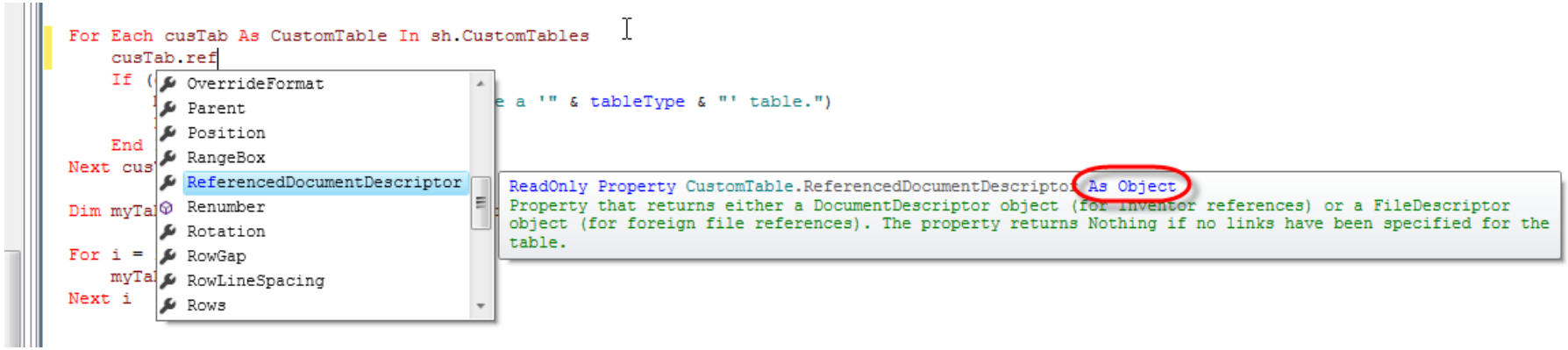


That sounds promising … let’s look at FileDescriptor:



OK, that looks good!

But ReferencedFileDescriptor property might return one of two kinds of object. How does it do that? Let's look in the Rule Editor:



Aha, it returns it "As Object". So we have to "cast" it or otherwise make sure it's the right kind.

Here's the easy way to do that:

```
Dim sh = ActiveSheet.Sheet

Dim pt = ThisApplication.TransientGeometry.CreatePoint2d(sh.Width/2, sh.Height/2)

'Dim myTab = sh.CustomTables.Add(tableType, pt, 2, nRows, {"Col1", "Col2"})

Dim fullPath = System.IO.Path.Combine(ThisDoc.Path, "standardTables\" & tableType & ".csv")

For Each cusTab As CustomTable In sh.CustomTables
    Dim fd As FileDescriptor
    fd = TryCast(cusTab.ReferencedDocumentDescriptor, FileDescriptor)
    If (fd IsNot Nothing) Then
        If (fd.FullFileName = fullPath) Then
            MessageBox.Show("You already have a '" & tableType & "' table.")
            Exit Sub
        End If
    End If
Next cusTab

Dim myTab = sh.CustomTables.AddCSVTable(fullPath, pt, tableType, True)

For i = 1 To nRows
    myTab.Rows.Add()
Next i
```

TryCast takes the first argument, and tries to access it like the type in the second argument ("FileDescriptor" in this case). If it can be converted ("cast"), then it's OK. If it can't be cast to that type, then it returns Nothing.

Then, in this example, we get the "fullFileName" and compare that to what we're trying to use for the new table.

To test this thoroughly, you can insert a custom table and reference an iPart factory file. That will give you the other situation.

Just for training purposes, let's rewrite this using Try/Catch:

```
For Each cusTab As CustomTable In sh.CustomTables
    Dim fd As FileDescriptor
    Try
        fd = cusTab.ReferencedDocumentDescriptor
    Catch
        fd = Nothing
    End Try

    If (fd IsNot Nothing) Then
        If (fd.FullFileName = fullPath) Then
            MessageBox.Show("You already have a '" & tableType & "' table.")
            Exit Sub
        End If
    End If
Next cusTab
```

You should use Try/Catch when you are *expecting* an error, *and* you have a way of handling that. Don't use try/catch to "swallow" or ignore *unexpected* errors.

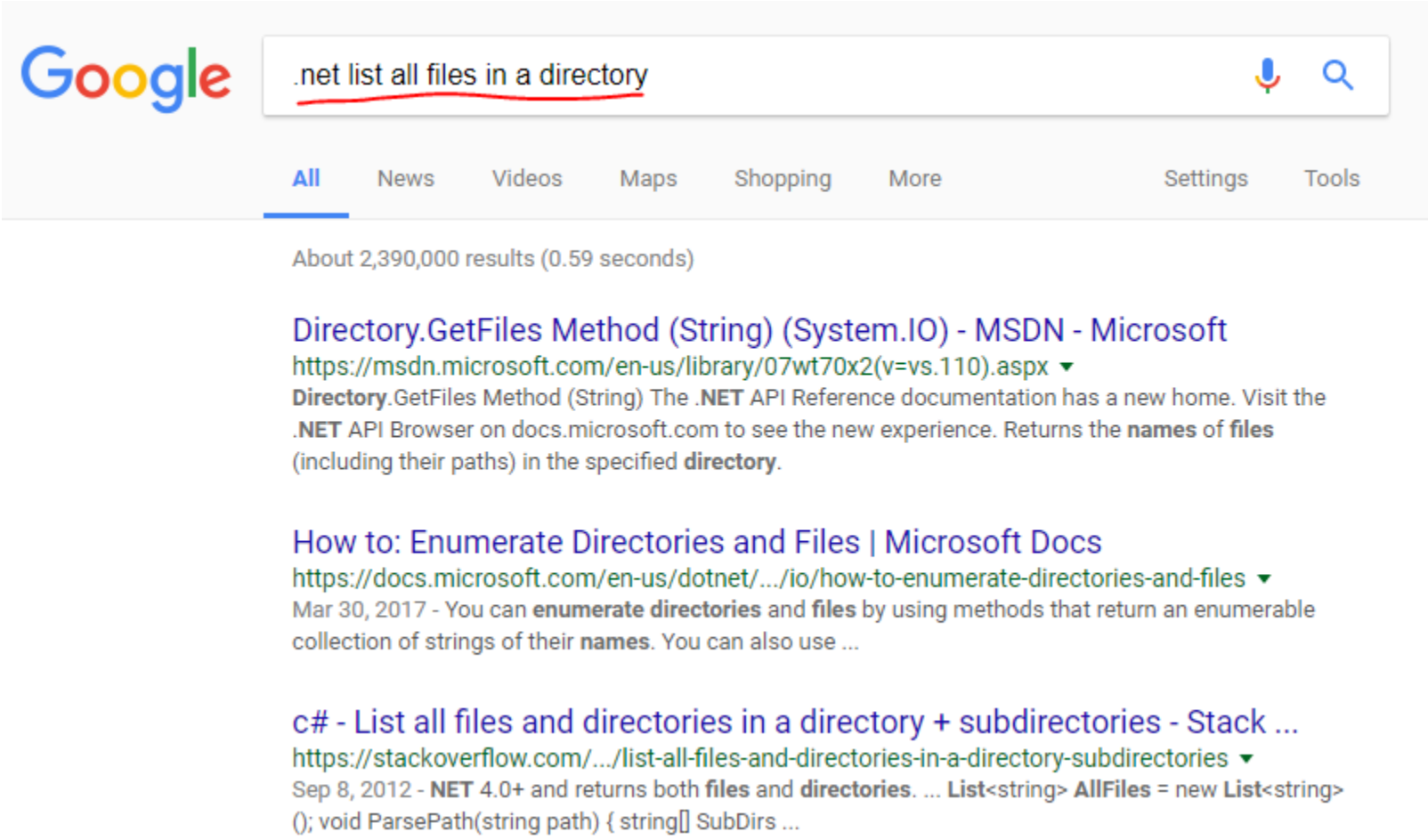
Try/Catch also has an optional "Finally" section, which is guaranteed to be executed, whether the main statements got an error or ran successfully.

OK!!! So that's working!

But… the options for the type of table to insert are hard-coded into the parameter. It would be better to grab them from the filesystem.

Let's write a *new rule* to do that.

We need to be able to find all the CSV files in the directory. It's easy to find utilities like this, just use Google, with ".net" in the search terms:



Some further searching leads to this function, which allows us to get all the files in a directory that match a certain pattern:

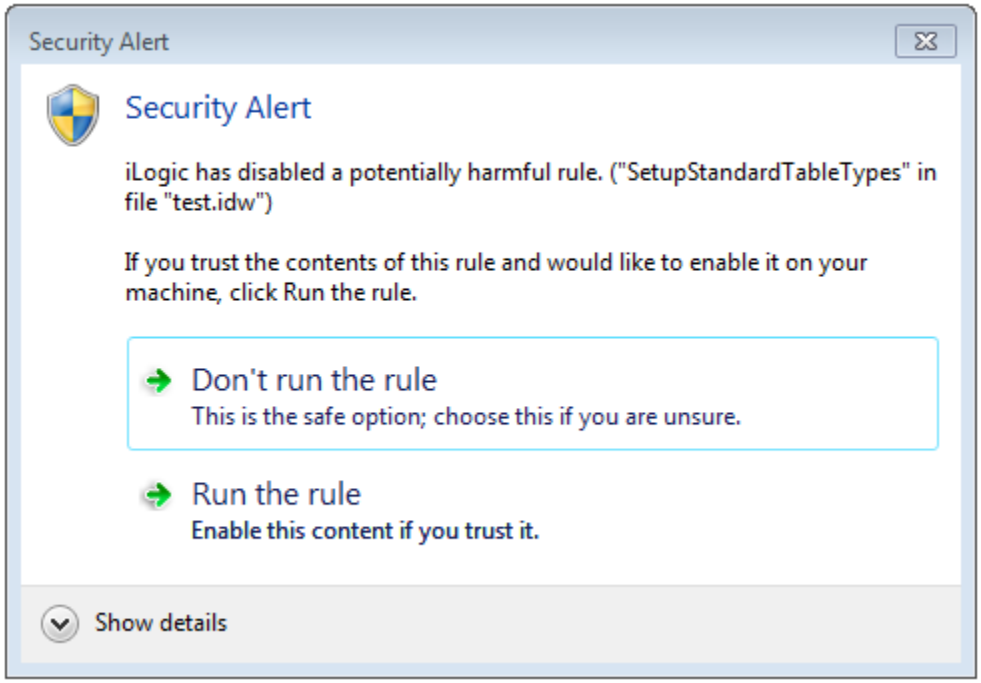
```
Dim files = System.IO.Directory.EnumerateFiles()
```

▲ 2 of 3 ▼ Function Directory.EnumerateFiles(**path As String**, searchPattern As String) As IEnumerable(Of String)
Returns an enumerable collection of file names that match a search pattern in a specified path.
path: The directory to search.

So we'll use it like this. Arrow points to a "test" statement.

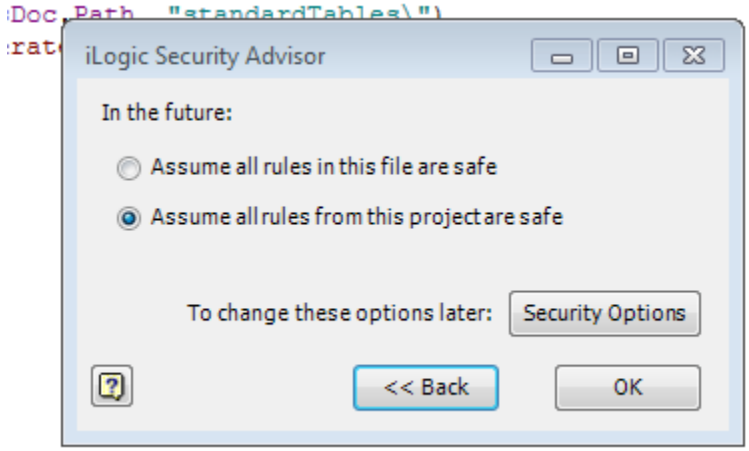
```
Dim Dir = System.IO.Path.Combine(ThisDoc.Path, "standardTables\")
Dim files = System.IO.Directory.EnumerateFiles(Dir, "*.csv")

For Each f In files
    MessageBox.Show(f)
Next f
```

Interestingly (or maybe annoyingly?), if you have iLogic Security turned on, this rule will trigger a warning:



If you were not expecting a rule to be searching your filesystem, this is your chance to stop it!

But we're OK with it, so I say "Run the rule". Then it gives me this dialog:



And I tell it to assume all rules from the project are safe (it's our own code, after all).

iLogic Security is OFF by default, so you may never see these messages.

Anyway, the for-each loop works OK.

Now we need to use iLogic functions to set the options. We need to use the "MultiValue" functions.

In the indicated code below, the first statement means, "in the future, when you set a multi-value list, also set the parameter's value to be the first (zero'th) value from the list (ensuring that the parameter's value is actually a member of the list).

The second statement is the assignment. Now we have to somehow construct the "array list".

```
Dim Dir = System.IO.Path.Combine(ThisDoc.Path, "standardTables\")
Dim files = System.IO.Directory.EnumerateFiles(Dir, "*.csv")

For Each f In files
    MessageBox.Show(f)
Next f

MultiValue.SetValueOptions(True, DefaultIndex := 0)
MultiValue.List("tableType") = MyArrayList
```

Here's how we construct the array list:

```
|

Dim Dir = System.IO.Path.Combine(ThisDoc.Path, "standardTables\")
Dim files = System.IO.Directory.EnumerateFiles(Dir, "*.csv")

Dim vals As New ArrayList()

For Each f In files
    vals.Add(System.IO.Path.GetFileNameWithoutExtension(f))
Next f

MultiValue.SetValueOptions(True, DefaultIndex := 0)
MultiValue.List("tableType") = vals
```

The first arrow points at a statement that creates a new empty ArrayList.

The second arrow points at the use of the Add() method, to add items to it (using the handy-dandy "GetFileNameWithoutExtension" method).

The third arrow just shows the renamed variable (renamed from what the "snippet" inserted)

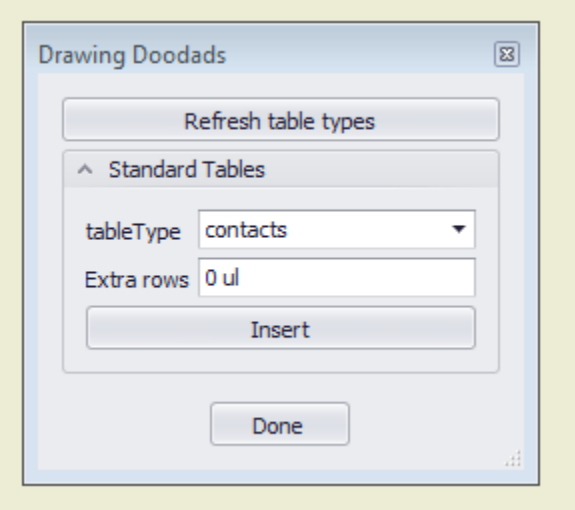
OK, run the rule … and … it works!

For testing purposes, if we rename one of the "standard" table CSV files to have a different extension, such as ".txt", and then re-run this rule, then that file no longer shows up in the list.

Here's the final version of that rule (including creating the parameters):

```
Sub Main

    Dim Dir = System.IO.Path.Combine(ThisDoc.WorkspacePath, "standardTables\")
    Dim files = System.IO.Directory.EnumerateFiles(Dir, "*.csv")

    Dim vals As New ArrayList()

    For Each f In files
        vals.Add(System.IO.Path.GetFileNameWithoutExtension(f))
    Next f

    CreateParameter("nRows", 0 ul, UnitsTypeEnum.kUnitlessUnits)
    CreateParameter("tableType", vals(1), UnitsTypeEnum.kTextUnits)

    MultiValue.SetValueOptions(True, DefaultIndex := 0)
    MultiValue.List("tableType") = vals

End Sub

Sub CreateParameter(pname As String, pval As Object, ptype As Inventor.UnitsTypeEnum)

    Try
        Dim dummy = Parameter(pname)
    Catch
        ThisDrawing.Document.Parameters.UserParameters.AddByExpression(pname, pval, ptype)
    End Try

End Sub
```

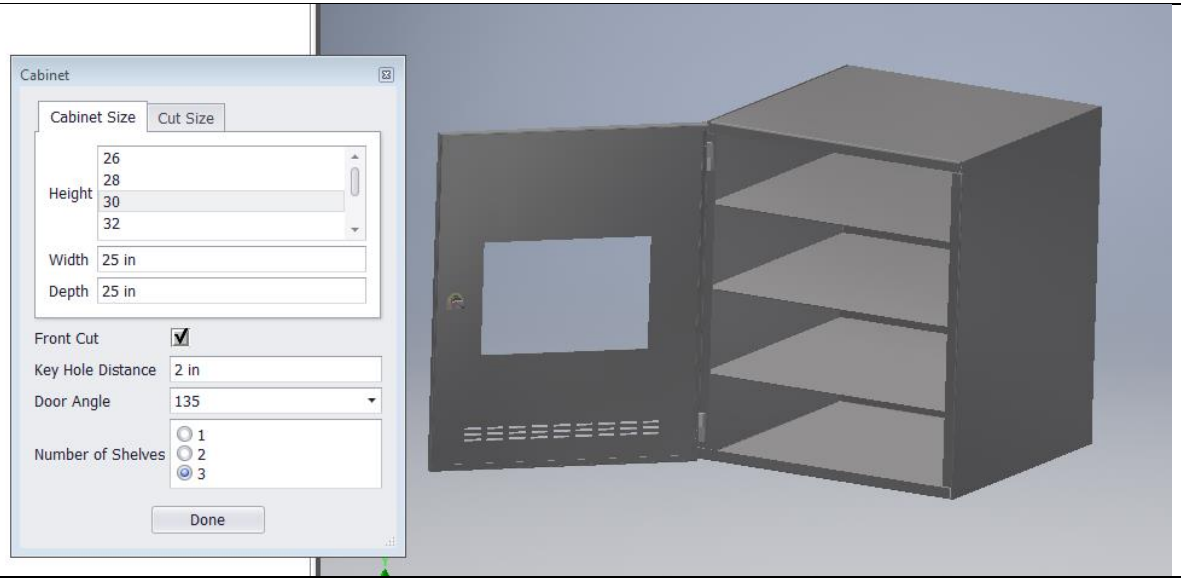
OK!!! But … when should we run this rule, “for real”? Well, one easy solution is to have a “setup” or “refresh” button on the form, and have a corresponding rule to do the setup. Let’s just do that:



The last step is to put the form and rules (and, if necessary, the relevant parameters) into the drawing template file. Use of template files is “ordinary Inventor stuff”; so we won’t discuss that here.

OK! That’s a “medium-sized” iLogic usage!

LARGE usage: “Configuration” application

“Cabinet” … pretty simple, actually. Options: • Size, height/width/depth • Number of shelves • Door options	

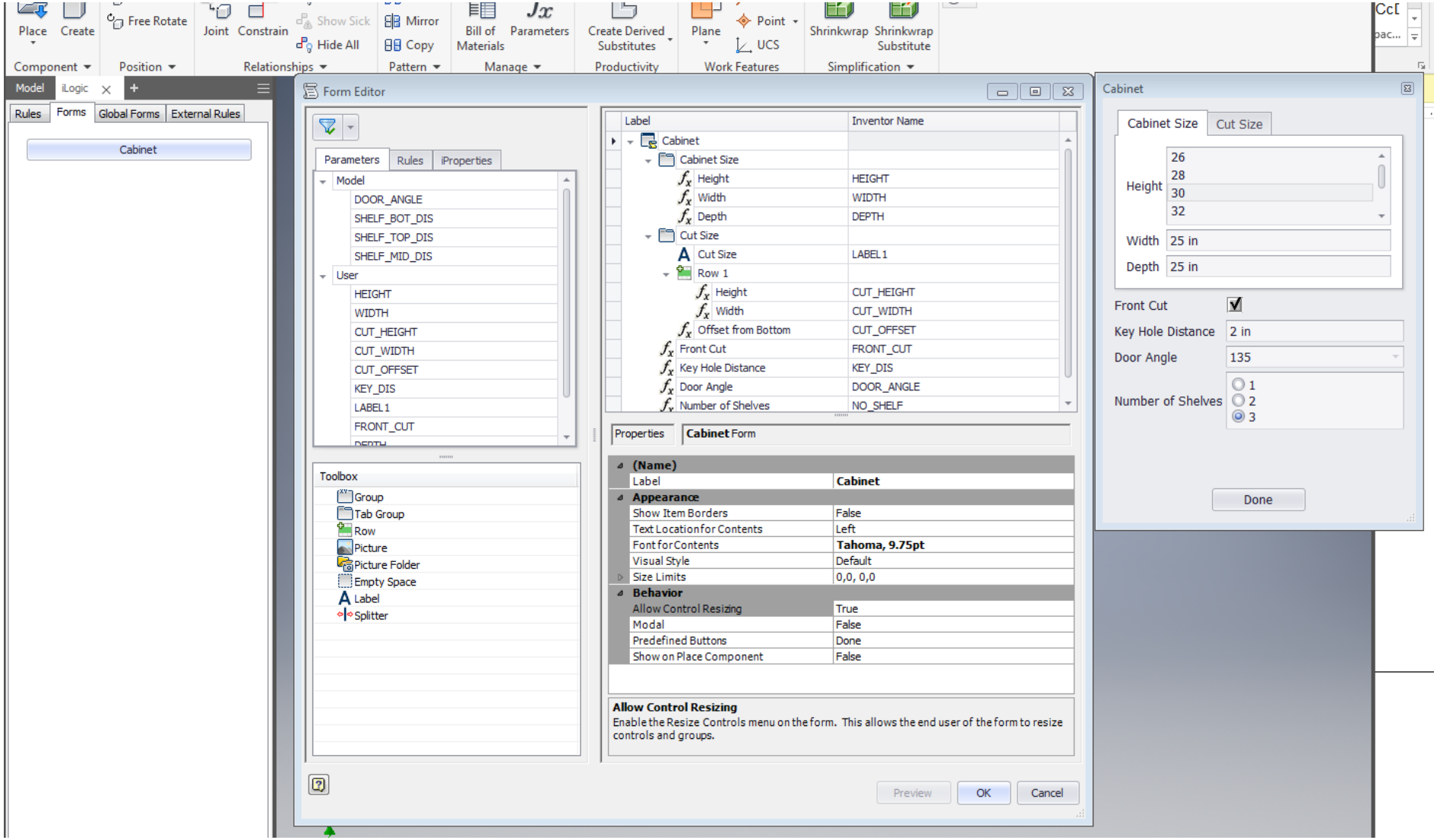
Why/when do we need this?

- It’s a “configure to order” kind of business
- Multiple orders, each one slightly different
- “in the sales process” (pre-order or post-order)

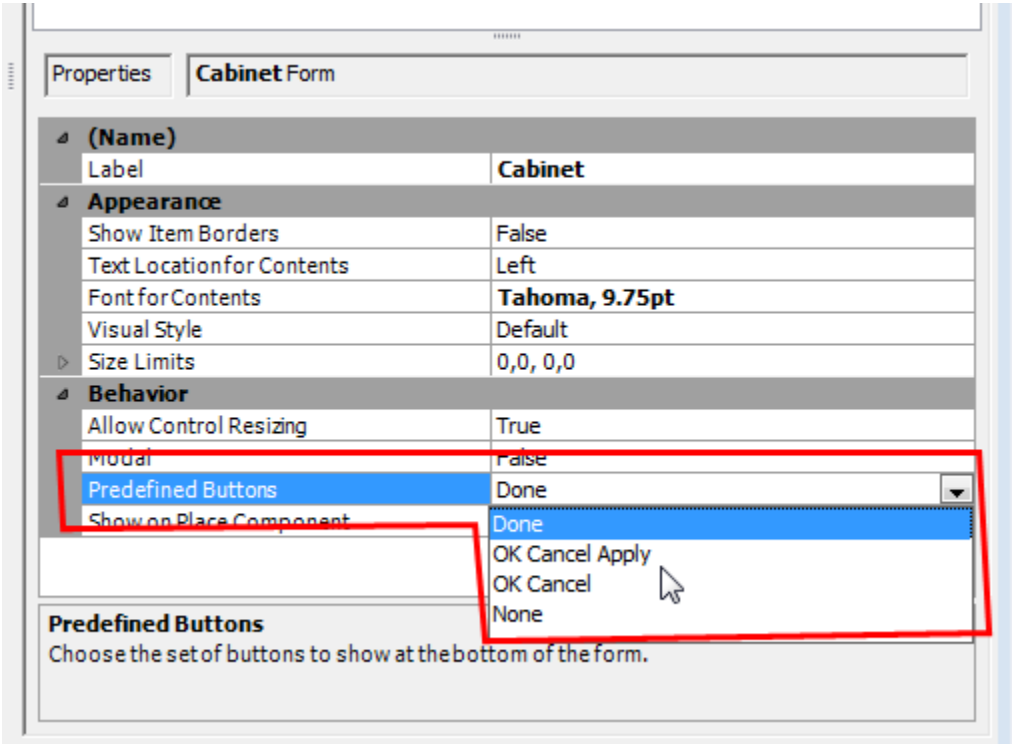
“Who” is going to use this, and are their skills/requirements? A “sales support” role.

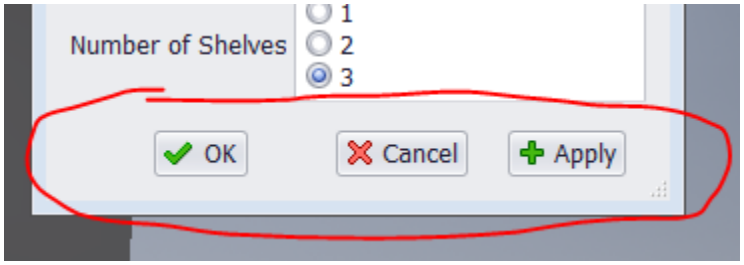
- Internal/external?
- Do they have access to Inventor?

"There's a Form"



Update immediately or not? (Done → OK/cancel/apply)



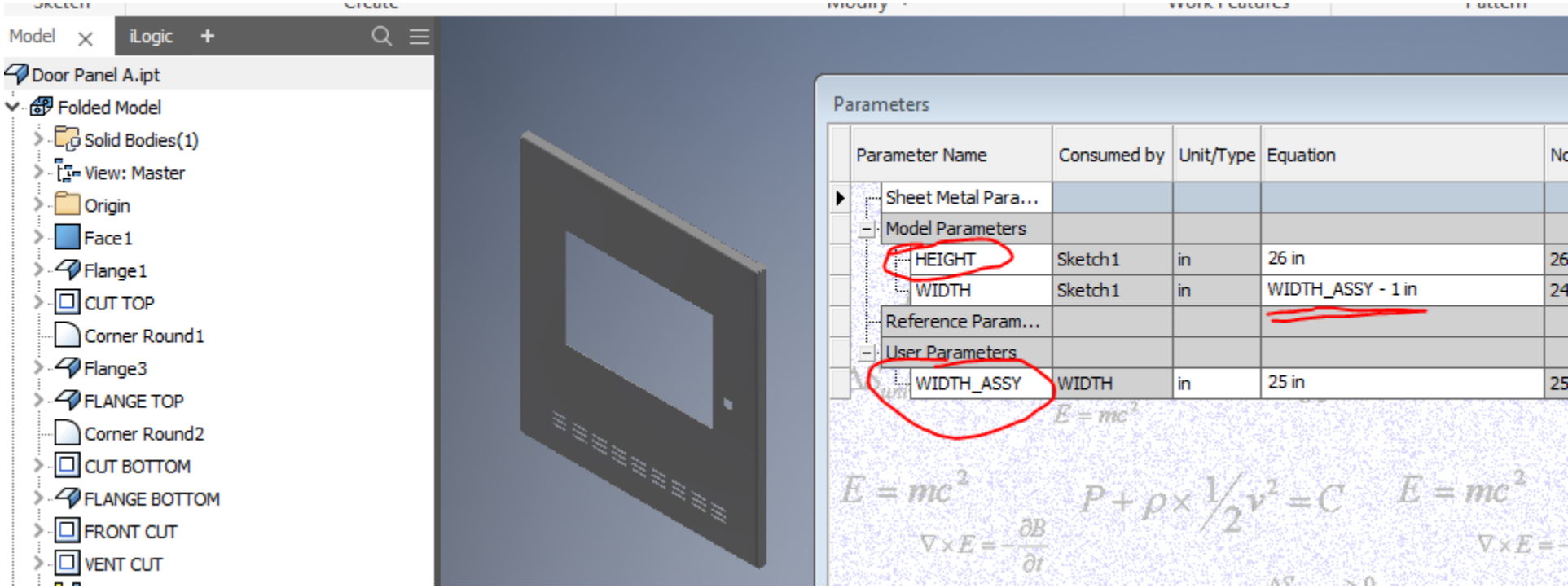


Allows changes to parameters in form, ***but doesn't do a full document-update until you click OK or apply***. Use for larger, more-complex models where updates take a long time.

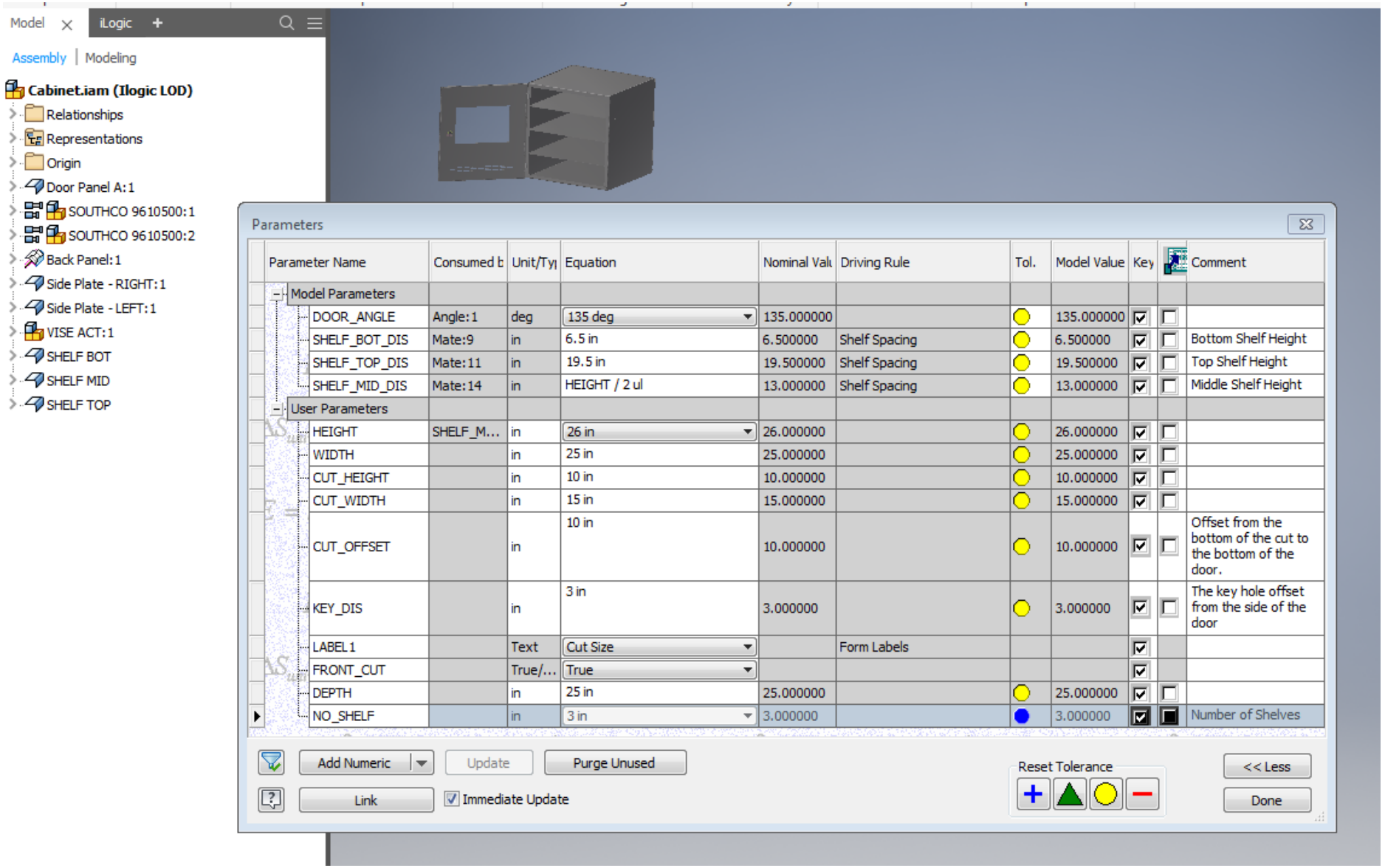
Concept: **Top-down data flow** (top-down control)

- Enter parameter values at assembly-level, push values down to components. In contrast to skeleton files. Be smart, use either/both when appropriate.

Here's the "door" part and its relevant parameters:



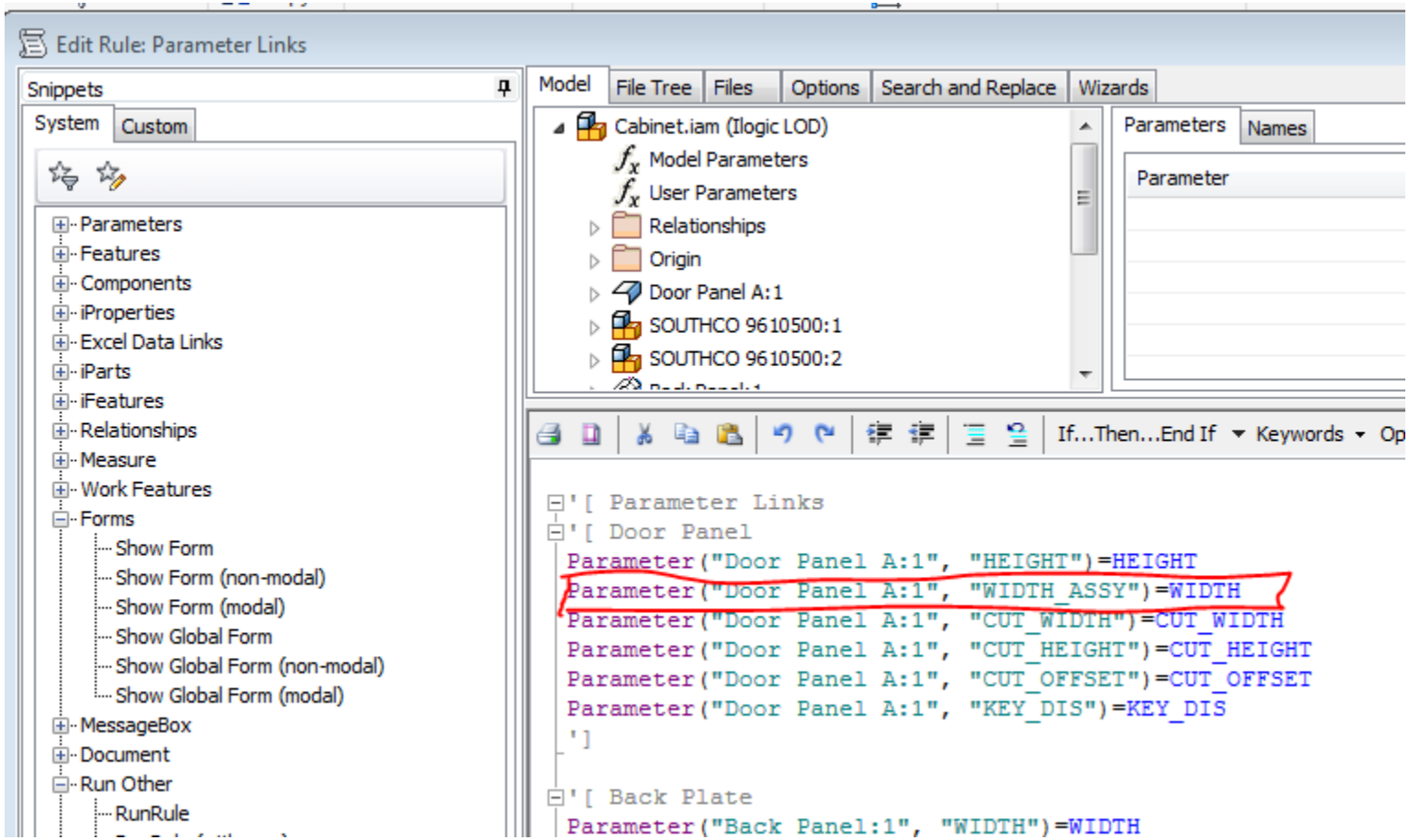
Here's the assembly and its relevant parameters:



Requires iLogic (top-down data-flow has no built-in Inventor support).

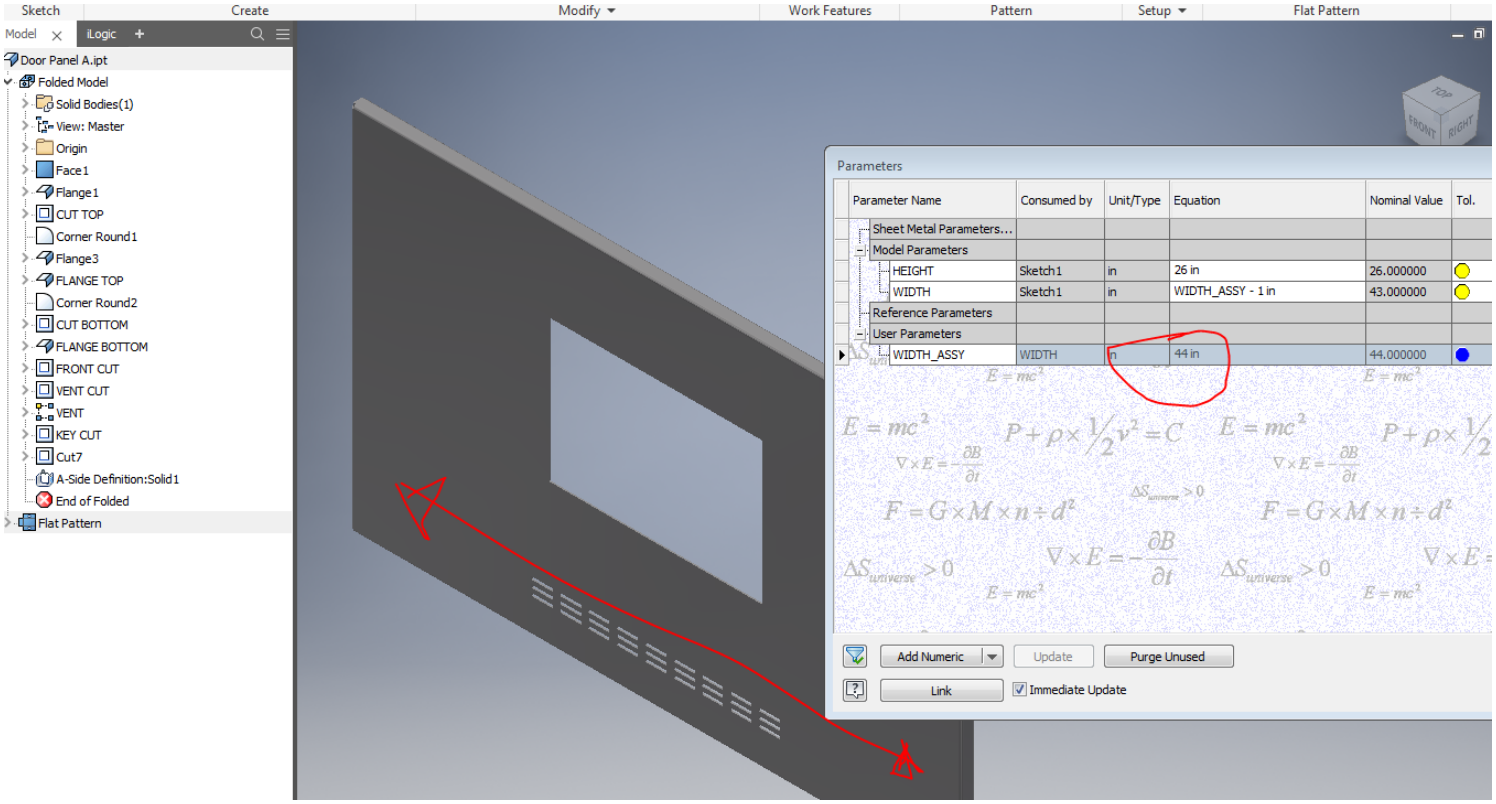
Implies that “some of the logic is in the rules, not the ‘model’” … but a mix may be best

Examine overall dimensions, e.g., width: This is a rule in the assembly. It “passes down” the width from the assembly level to the door (and other parts). Parameter names can match or not.

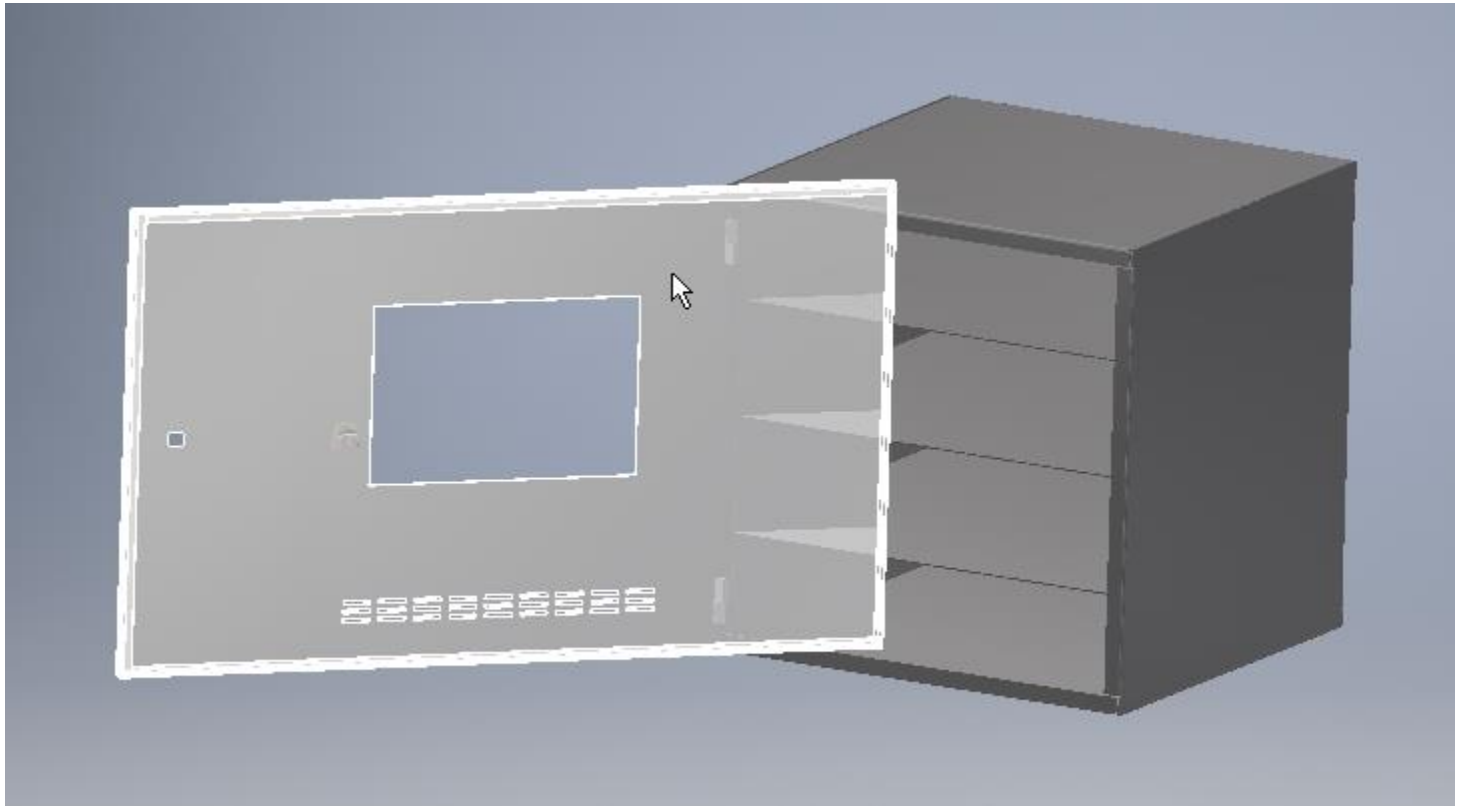


Parametric parts – change them independently and “things break”, as shown:

Open the “door” part and change the width to be 44 (large value, chosen at random):



Result in assembly:



“Don’t do that!” (old joke)

About the **"Parameter"** function

We are using it to "pass data down to next level".

Implication on files. At least three options:

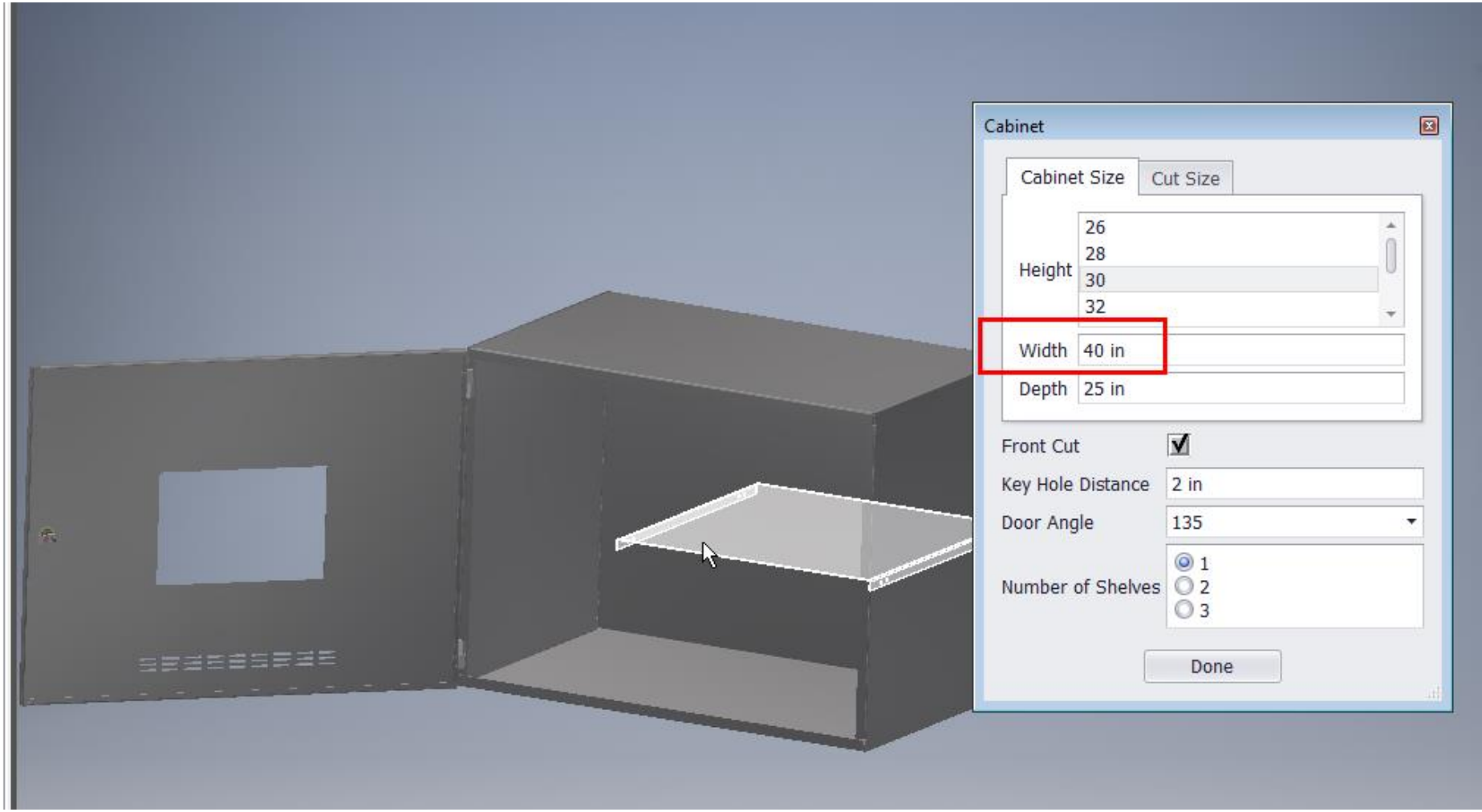
1. **Fully custom**. Need a copy for each "job" or "order". We're actually modifying "the" file. The implication is that "you" have to make the copy. That's part of the "job" or "order" setup process. *That's what we're doing in this example.* Common technique.
2. **iParts** when there are limited permutations, known in advance. Use iLogic's "iPart" FindRow or ChangeRow functions … need to ensure that the relevant member is found, either by only offering valid choices, or by checking for result. Would be implemented differently than shown here.
3. **Custom copies with naming convention**. Like "DIY iPart". E.g., "shelf_24x18.ipt". Find and reuse previously-created files based on parameter-requirements. Create new ones from "factory" (ordinary IPT) when needed. Need to address or ignore issue of changes to "factory" file vs existing "member" files. Use of true "custom iPart members" is also an option. More work for you to implement, and to manage the files.

Parameter function … argument is occurrence name or filename … currently uses “SHELF BOT” (occurrence name), but won’t work for width/depth when only one shelf (mid).

Doesn’t work if “SHELF BOT” is not present!

```
'[ Shelf
Parameter("SHELF BOT", "WIDTH")=WIDTH
Parameter("SHELF BOT", "DEPTH_ASSEM")=DEPTH
']
```

And when using filename, it has to be open and ref’d from this document (not just unrelated file)


Cabinet
Cabinet Size
Cut Size
Height
26
28
30
32
Width
40 in
Depth
25 in
Front Cut
Key Hole Distance
2 in
Door Angle
135
Number of Shelves
1
2
3
Done

So instead do this:

```
'[ Shelf
Parameter("Shelf.ipt", "WIDTH")=WIDTH
Parameter("Shelf.ipt", "DEPTH_ASSEM")=DEPTH
']
```

…which updates the file, regardless of which components are present in the assembly … but with a further restriction … at least one such component must be present in the assembly. That's a valid assumption in this model.

Also note – "Parameter( )" does not trigger the rule upon changes to the ref'd parameter, unlike direct references to parameter-names (i.e., in BLUE, without using the "Parameter( )" function).

Rule IS triggered if either DEPTH or HEIGHT is changed:

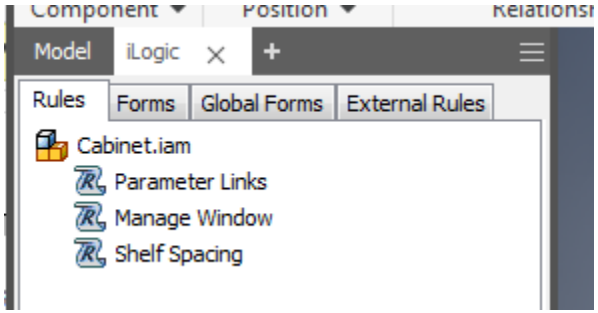
```
'[ Side Plate Left
Parameter("Side Plate - LEFT:1", "DEPTH_ASSY")=DEPTH
Parameter("Side Plate - LEFT:1", "HEIGHT")=HEIGHT
']
```

Rule IS NOT triggered if either DEPTH or HEIGHT is changed:

```
'[ Side Plate Left
Parameter("Side Plate - LEFT:1", "DEPTH_ASSY")=Parameter("DEPTH")
Parameter("Side Plate - LEFT:1", "HEIGHT")=Parameter("HEIGHT")
']
```

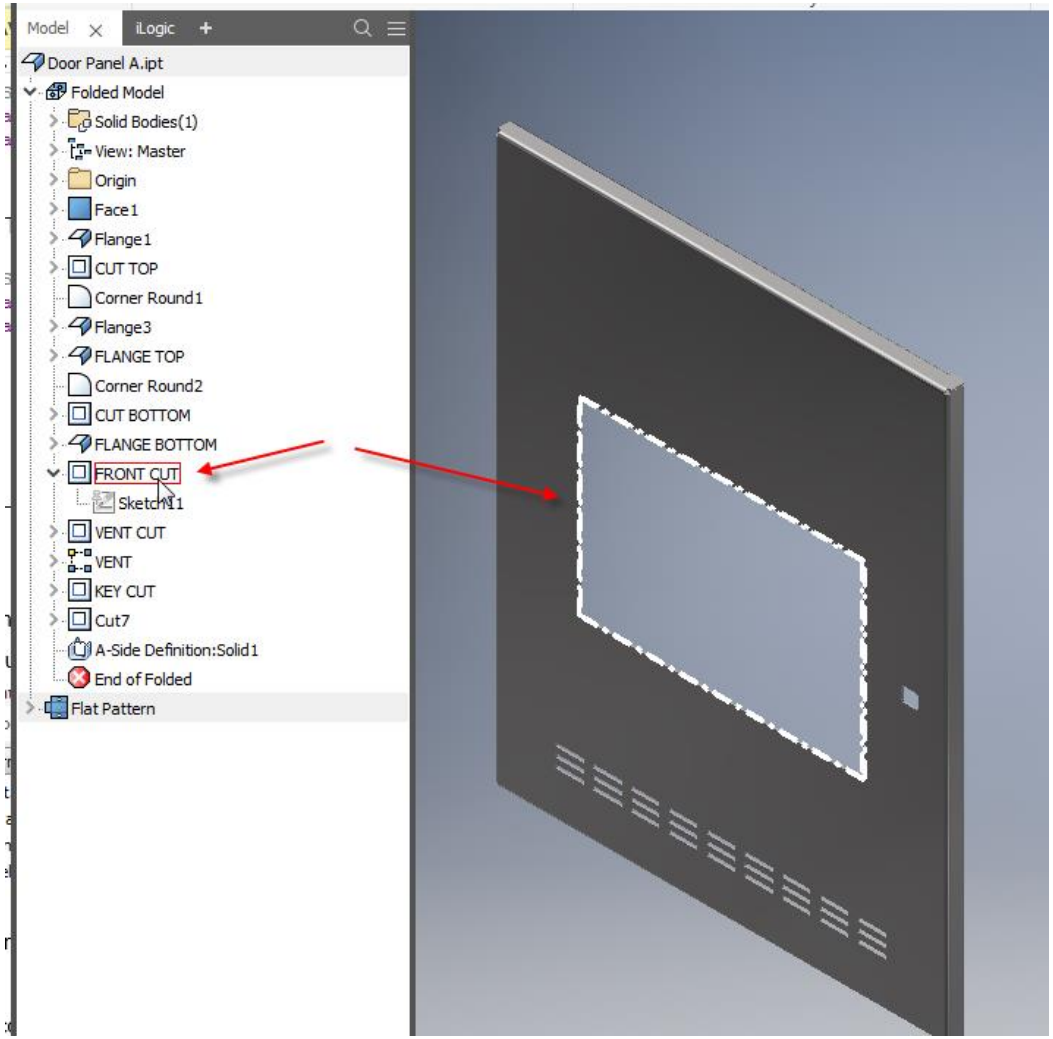
Door options: (for “front cut” (window))

Can use multiple independent rules, or not.

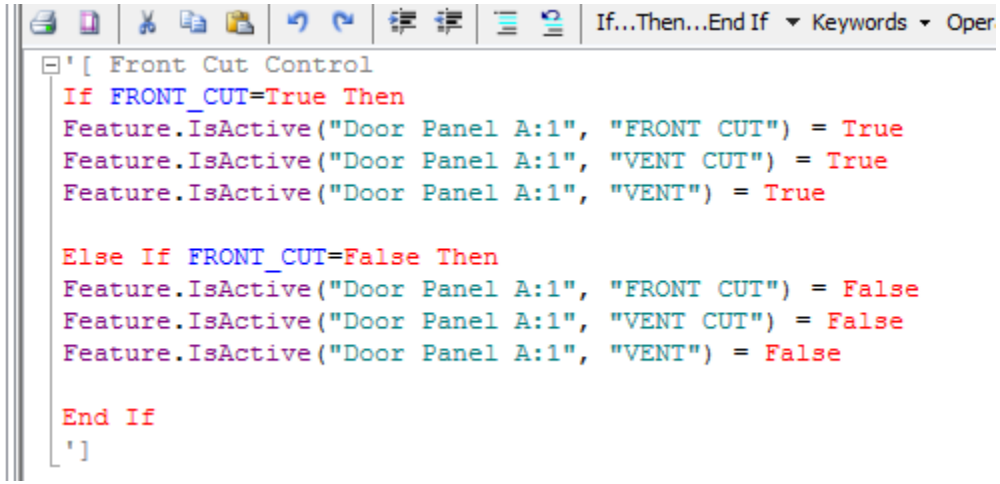


My preference is to keep rules to a “manageable” size. Here the “Manage Window” rule controls the presence/absence of the window in the door. Note that this still an assembly-rule.

Door part contains “cut” feature that is unconditional:

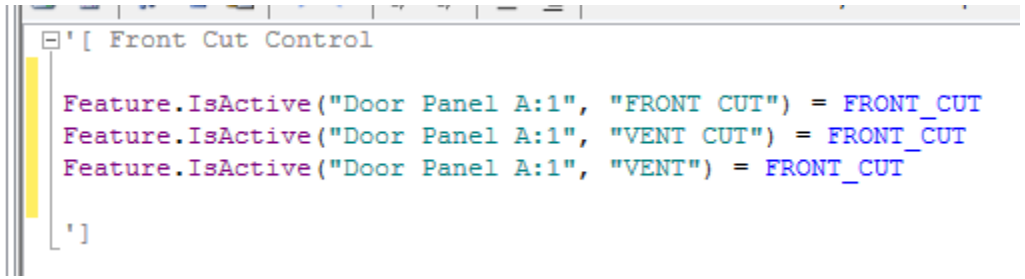


The assembly-rule simply suppresses it, or not, using Feature.IsActive:



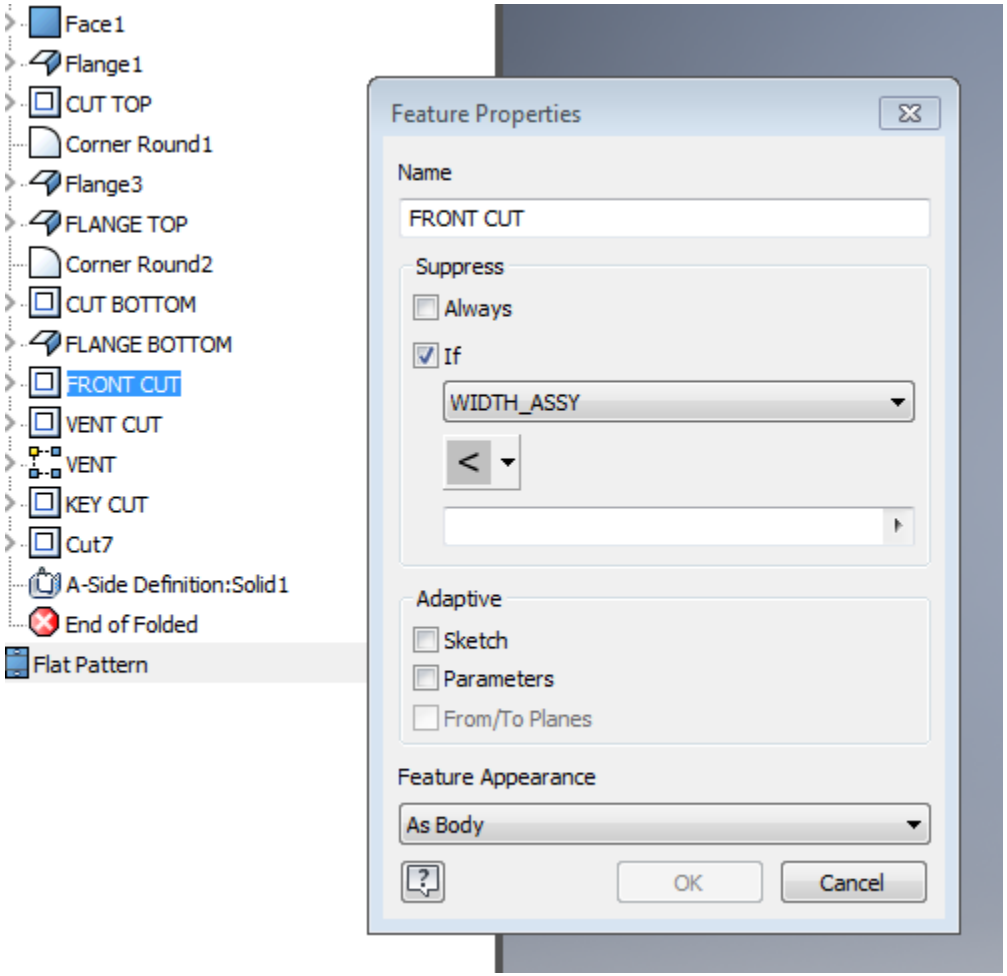

```
'[ Front Cut Control
If FRONT_CUT=True Then
Feature.IsActive("Door Panel A:1", "FRONT CUT") = True
Feature.IsActive("Door Panel A:1", "VENT CUT") = True
Feature.IsActive("Door Panel A:1", "VENT") = True

Else If FRONT_CUT=False Then
Feature.IsActive("Door Panel A:1", "FRONT CUT") = False
Feature.IsActive("Door Panel A:1", "VENT CUT") = False
Feature.IsActive("Door Panel A:1", "VENT") = False

End If
']
```

Note there is an easier/shorter way to write this:



```
'[ Front Cut Control

Feature.IsActive("Door Panel A:1", "FRONT CUT") = FRONT_CUT
Feature.IsActive("Door Panel A:1", "VENT CUT") = FRONT_CUT
Feature.IsActive("Door Panel A:1", "VENT") = FRONT_CUT

']
```

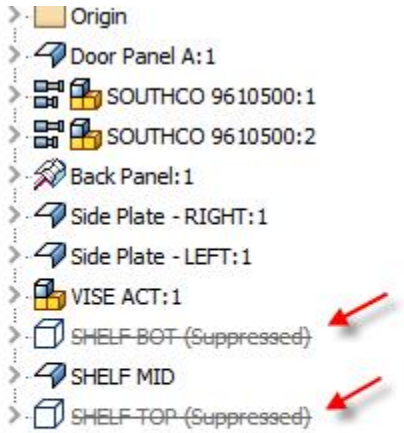
It is also possible to use conditional feature suppression. In that case, we'd set the controlling parameter, instead of using Feature.IsActive. But it has to be a numeric value, so use 1 or 0 "ul" (not shown here)

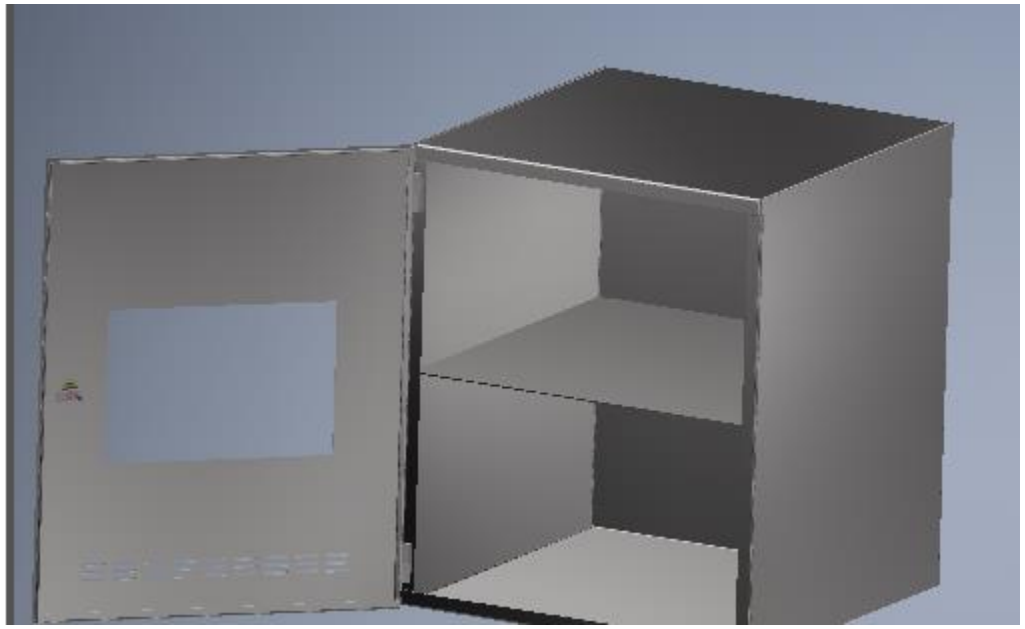


Note that all the "real" logic is in the rule in the assembly, not the door-part. Therefore, all the door-ui has to be in the assy-form.

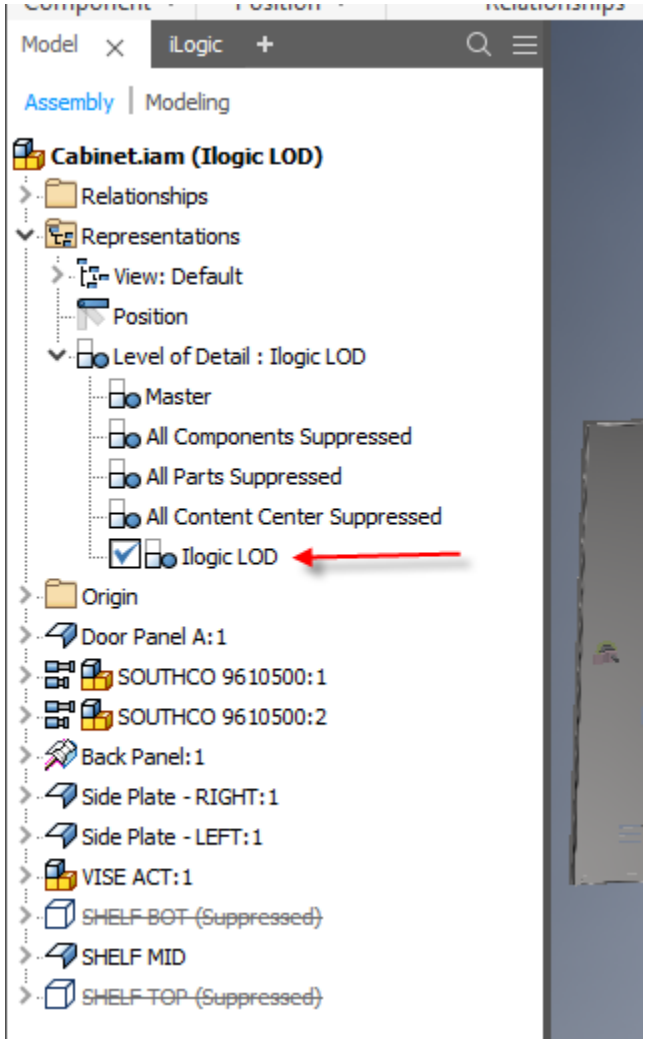
Shelves:

- Note: Could have been done with an “occurrence pattern”, *let’s assume there was a reason to do it this way*. E.g., each shelf might be different kind.
- Conditional components use “suppression” … implies the “master model” must contain all possible components.

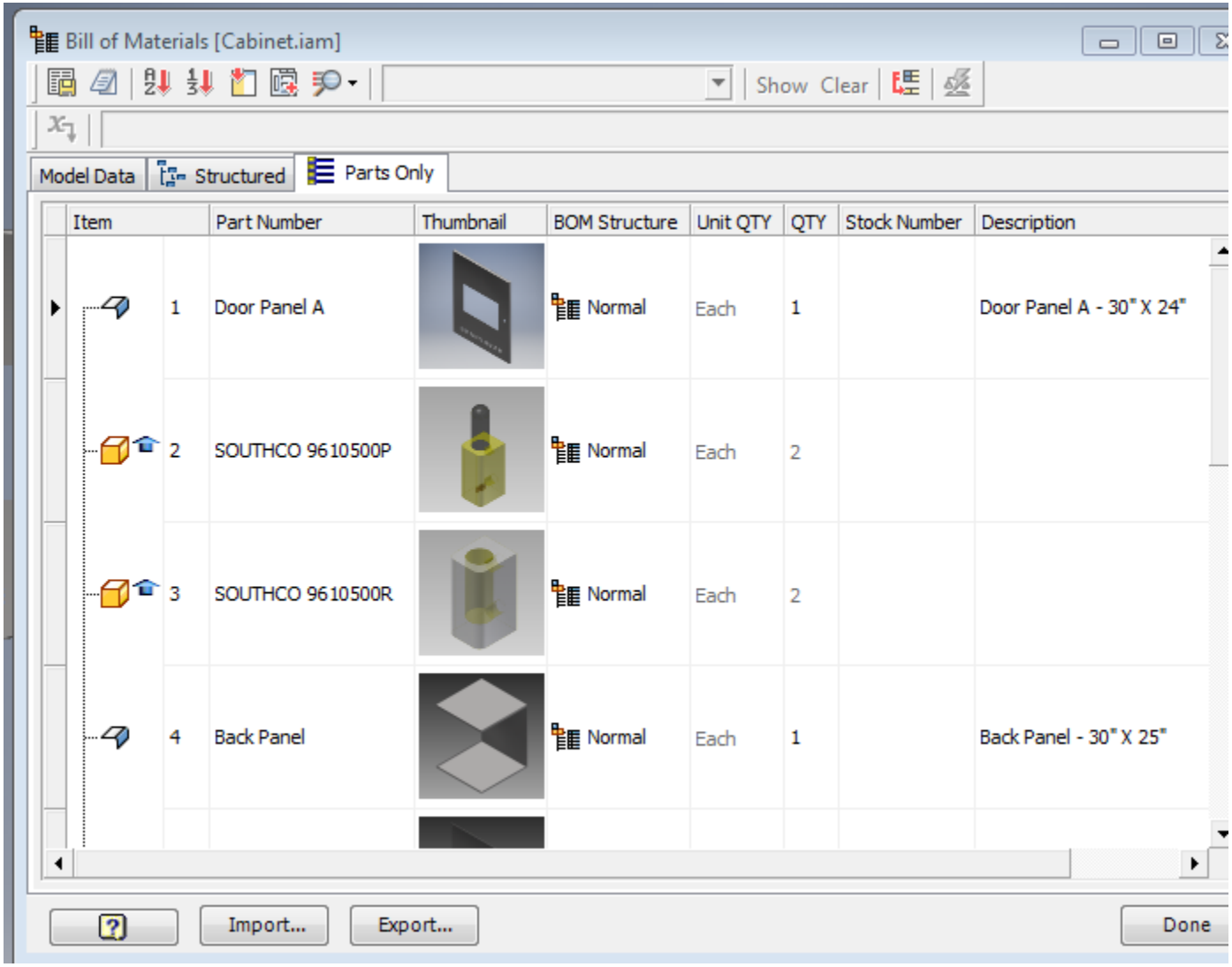




Suppression requires a custom “LOD” (can’t use master):



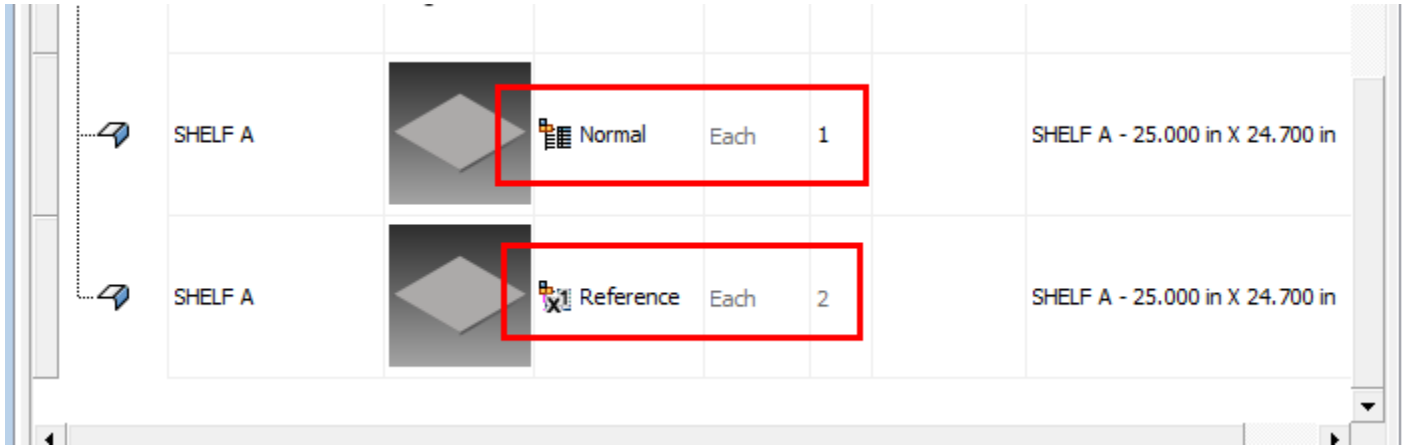
BOM, unfortunately, always uses Master (ignores suppression).



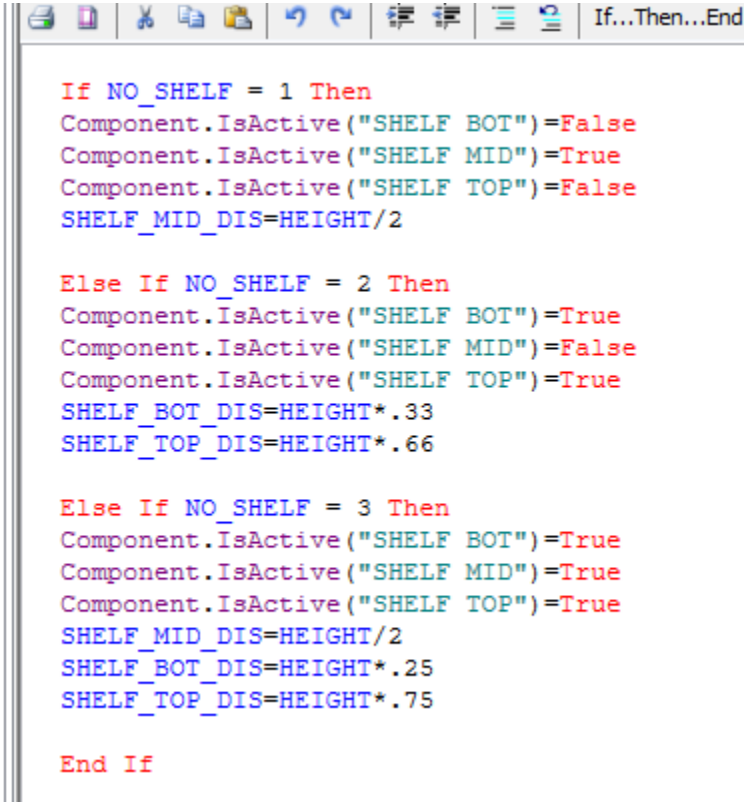
(shelves not shown in image)

So:

- LOD's really intended as a "memory management" technique for large assemblies, being "abused" by suppression-for-configuration. But … OK, everyone does it.
- "Master model" can be problematic if there are zillions of possible components.
- iLogic "Component.IsActive" does **two things**:
  - Suppress/unsuppress
  - Toggle BOM Structure between "Normal" and "Reference" ("reference" means "not included in BOM")



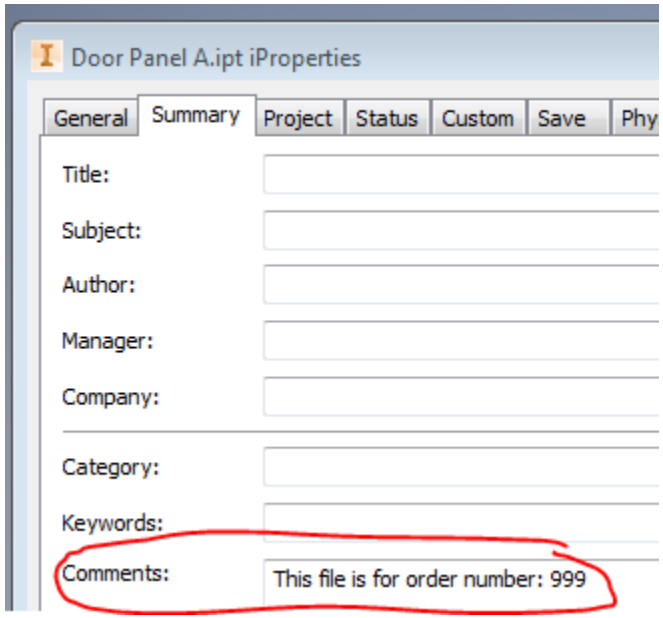
So rule is pretty simple:



OK! Pretty straightforward!

Using SharedVariables between files

Let's say the main assembly has an "order number", and we want the part files to be able to access that. Part files don't "know" that they are part of an assembly, so how can they get that information?



The files can work together, something like this.

You can create a shared variable just by using the SharedVariable function, with any name, and assign a value to it. Shared variables are not saved, and only persist to the end of the session.

In assembly:

```
SharedVariable("ORDER_ID") = ORDER_ID

Dim adoc As AssemblyDocument = ThisDoc.Document
Dim refs = adoc.AllReferencedDocuments
For Each ref As Document In refs
    Try
        iLogicVb.RunRule(ref.DisplayName, "ProcessOrderID")
    Catch
    End Try
Next ref
```

When you change the order-ID-number, this rule runs, the shared variable is updated, and all of the relevant files are “notified”.

In addition, the shared variable is available for the rest of the Inventor session. The shared variable itself is not saved (but in this case, the ORDER_ID parameter is saved in the assembly).

In a part file (in “ProcessOrderID” rule):

```
Dim Str As String

Try
    Str = SharedVariable("ORDER_ID")
Catch
    Str = "[not available]"
End Try

iProperties.Value("Summary", "Comments") = "This file is for order number: " & Str
```

Note that the try/catch in the assembly-rule handles the situation when a given part-file doesn’t have a “ProcessOrderID” rule.

External rules

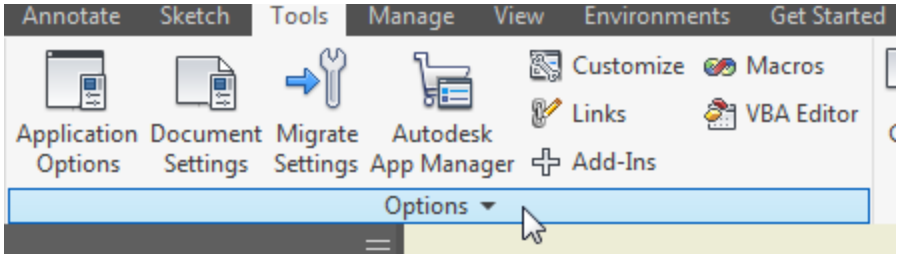
The “call a rule in every file” capability seems like it could be more-generally useful, i.e., perhaps used by other rules/files, and so is a good candidate for an External Rule. It is a “utility”.

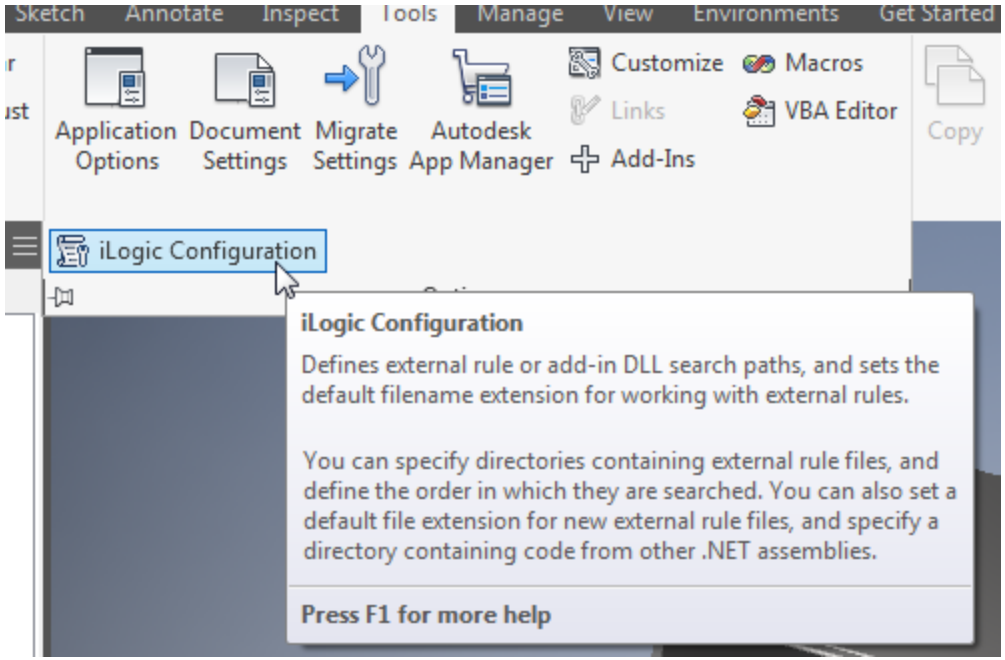
About external rules:

- Each external rule lives in its own file.
- File extension is “iLogicVb” but there are some alternatives too
- External rules don’t know what Inventor document they will run in, so are not triggered by parameter changes. So they are only run by explicitly running them, either from a form or another rule, or by being attached to an event.
    - Note as of Inventor 2018.1 there are now “Global Events” … only External Rules can be attached to Global Events.
- External rules do *run* in the context of the active document, so they can access parameters using the Parameter function, or any other aspect of the document. But you, the author, cannot ensure that the document has these aspects, unless you check for them yourself in the rule. Or just assume it’s right…
- iLogic only uses external rules from known folders:

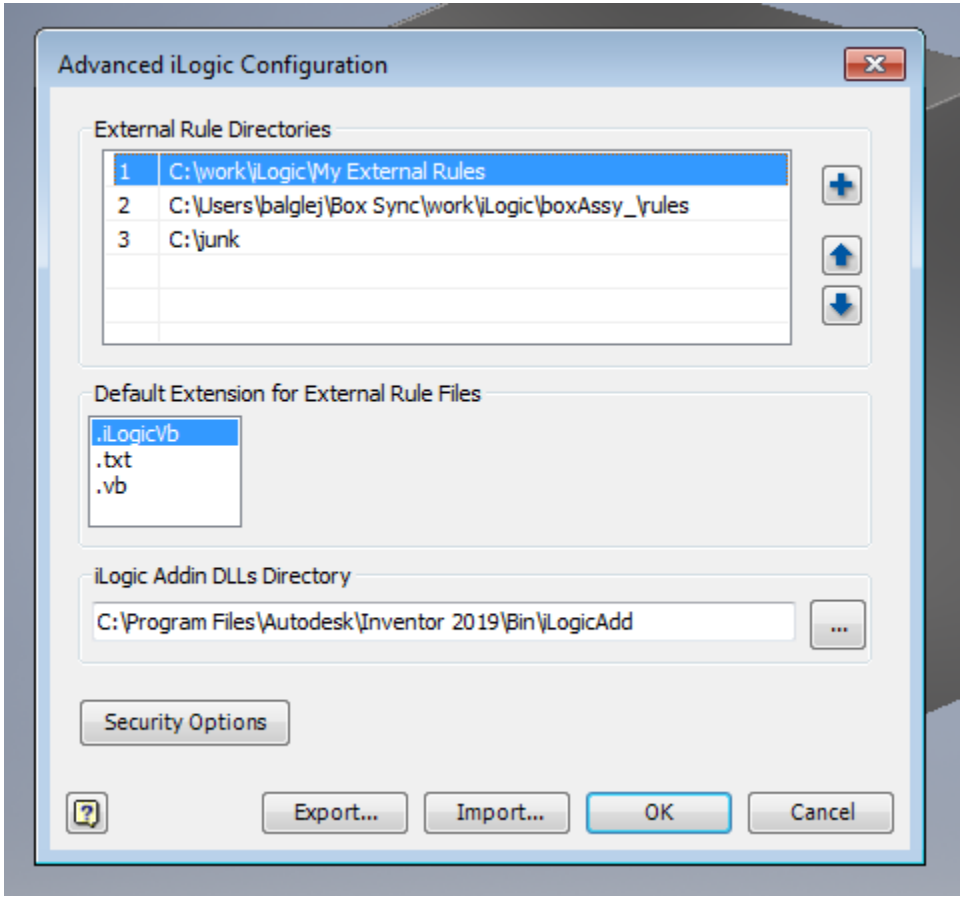
First you need a folder to store the external rules. Create it somewhere.

Then tell iLogic where it is. Use the “iLogic Configuration” command, in (under) the Tools ribbon:

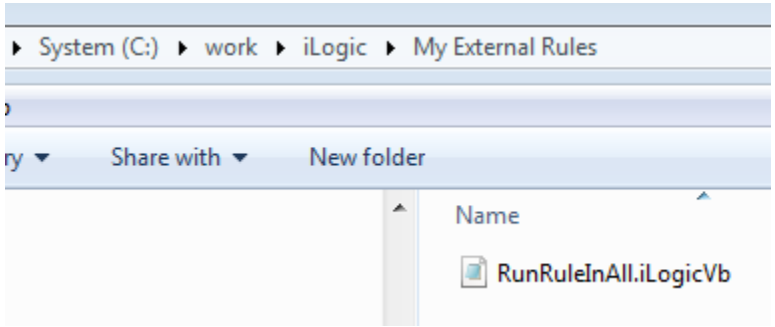



Sketch
Annotate
Inspect
Tools
Manage
View
Environments
Get Started
Application Options
Document Settings
Migrate Settings
Autodesk App Manager
Customize
Links
Add-Ins
Macros
VBA Editor
Copy
iLogic Configuration
iLogic Configuration
Defines external rule or add-in DLL search paths, and sets the default filename extension for working with external rules.
You can specify directories containing external rule files, and define the order in which they are searched. You can also set a default file extension for new external rule files, and specify a directory containing code from other .NET assemblies.
Press F1 for more help

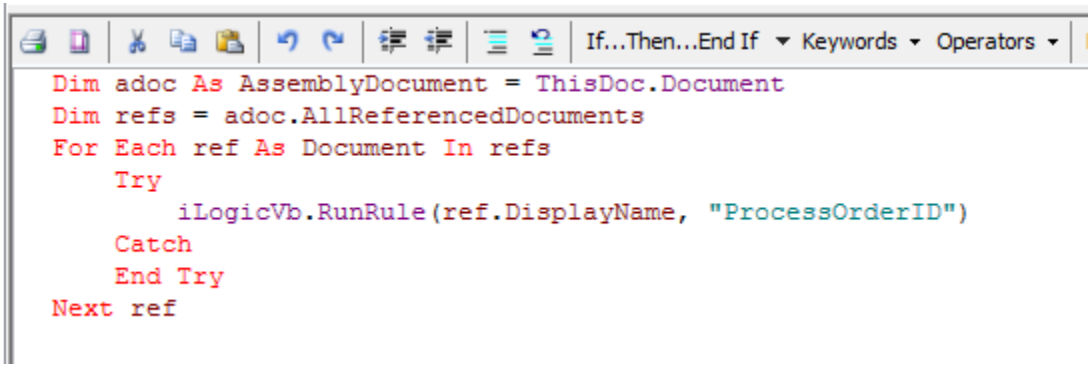
Add your folder to the “External Rule Directories” list:



In the External Rules tab, create a new rule. Note that each rule is its own file, so you are creating a file. I prefer the “iLogicVB” extension.

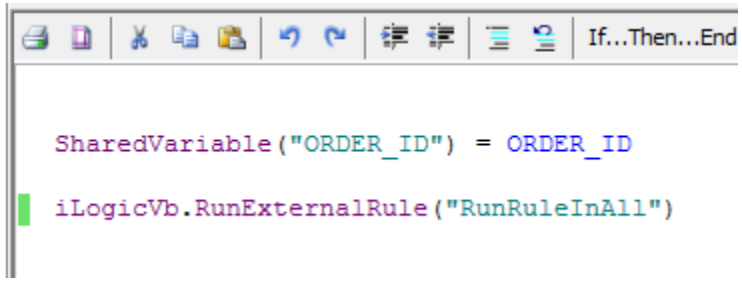


In this case, we just copy and paste the relevant code from the “internal” rule to the new External rule.



The advantage of External rules is they are outside of any file, so are available to any file. The disadvantage is … they are outside of any file, so are not aware of the parameters in the file. If you need to access any parameters, you have to use the "long-hand" version to access the parameters (the "Parameter( )" function). But you have to *know* that the parameter is present. And further, iLogic will never trigger an External rule because of a parameter change.

In this case, though, it doesn't matter, because we're going to explicitly call the external rule from an internal rule:



But note that this "utility" always runs the same internal rule (ProcessOrderID). That's great in this case, but if we want it to run different rules in other cases … then we need to tell it which rule to run.

RuleArguments

Allows the caller of the rule to “pass in” additional information. Construct a “name value map”, add the data to that, and invoke the external rule with it:

Internal rule in assembly:

```
SharedVariable("ORDER_ID") = ORDER_ID


Dim arguments As Inventor.NameValueMap = ThisApplication.TransientObjects.CreateNameValueMap()
arguments.Add("internalRule", "ProcessOrderID")


iLogicVb.RunExternalRule("RunRuleInAll", arguments)
```

External rule:

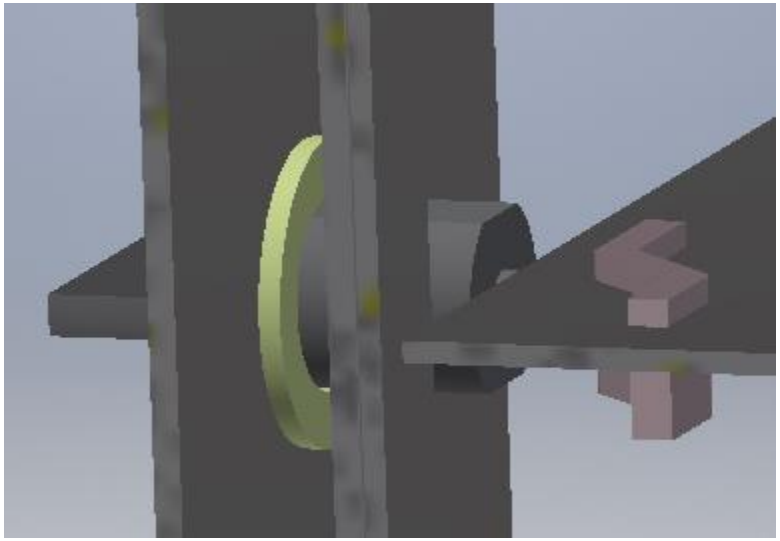
In the external rule, we can get the supplied data using the “RuleArguments” function.

```
Dim intRule As String = RuleArguments("internalRule")

Dim adoc As AssemblyDocument = ThisDoc.Document
Dim refs = adoc.AllReferencedDocuments
For Each ref As Document In refs
    Try
        iLogicVb.RunRule(ref.DisplayName, intRule)
    Catch
    End Try
Next ref
```

This functionality (RuleArguments) is also applicable to internal rules. You can also “pass back” data using the same “map”.

OK, that was all programmer-speak. Let's finish with a modeling "success":

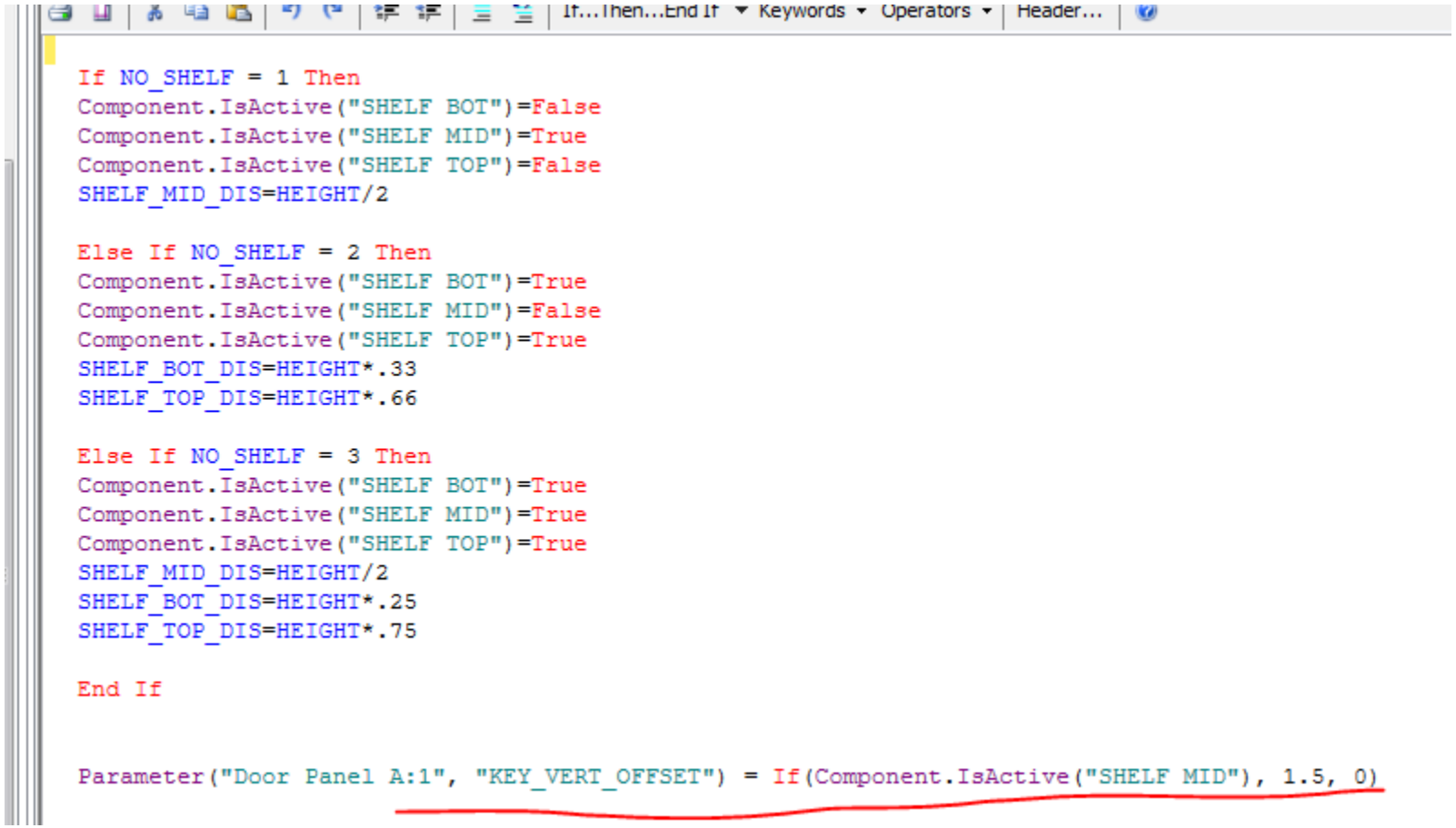
Even simple assemblies have quirks. Suppose that a common mistake (by the "salespeople", when "spec'ing" a cabinet) is to leave the handle at the vertical center, interfering with a middle shelf, as shown here:



So when there is a middle shelf, we'd like to automatically adjust the handle-position.

Here's the "key":

Add some "logic" into the model, an extra parameter that controls the vertical offset of the handle. It can be zero in the "normal case". Then use iLogic to control that parameter:



A great little example of how an automated solution can prevent expensive mistakes!

Things We've Seen

- Small, medium, large usage

Small:
- Simple: Clamp parameters into range ("range-checking")
- Warn upon bad interaction among parameter values
- Round off user-input
- Keep iProperties in sync with parameters

Medium:
- Look through snippets for available starting-points
- Jump into Inventor API (seamless!)
- Use auto-complete (enable filter if necessary)
- CustomTables collection on Sheet
- Look at "arguments" help (e.g., for CustomTables.Add)
- Add/AddCSVTable
- TransientGeometry
- Making a form (internal to document)
- New parameters – use "ul" type for integer
- New parameters – "Text" parameter
- New parameters – multi-value
- Form – rule-button
- Rule editor – blue text means Inventor parameter
- Parameters are triggers
- Triggers are occasionally not desirable – suppress rule to stop
- System.IO.Path.Combine
- Set width of fields in forms
- Using "As *type*" in for-loop variable
- Use "reference manual" for API doc (online as of 2017)
- ReferencedDocumentDescriptor
- TryCast
- Try/Catch
- Using ".Net" as Google search keyword
- System.IO.Directory.EnumerateFiles
- iLogic Security
- MultiValue snippets
- ArrayList
- "Rule" button on Form

Large:
- Configuration – who's going to use it?
- Needs a form
- Form – immediate vs. explicit update
- "top-down data-flow"
- iLogic "Parameter" function (occurrence vs. filename)
- Need a copy of all the files "per job/order" (data wrangling)
- One big rule or multiple rules
- Feature.IsActive to suppress.
- Conditional suppression.
- Patterns or individual components?
- Component.IsActive (vs. LOD and BOM and "master model")
- Shared variables
- External rules
- Rule arguments
- Preventing mistakes using iLogic rules! (adjust handle-position to avoid middle shelf)

Some other things, not discussed, that you might be interested in:

- The “Place iLogic Component”. Makes a copy of the component, inserts it in the assembly. Option to pop up a specific form upon insertion, for customizing.
- Order of rules. Can make a difference in some circumstances. Drag them in the Rules tab.

More questions?

Search/Post on Inventor Customization forum:
https://forums.autodesk.com/t5/inventor-customization/bd-p/120