Tips and Tricks for Creating a Configurator with Autodesk® Inventor®

Brian Ekins – Autodesk, Inc.

MA316-5

Class Description:
Automatically constructing parts and assemblies based on their description is the holy grail of CAD for many. In this class, we will look at some of the options for doing just that. We will also look at some of the realities and pitfalls you need to be aware of when designing a configuration system. Topics to be discussed include what to consider when creating libraries, using iLogic™, building assemblies, creating the front-end user-interface, web-based options, creating drawings, choosing output formats, and more. Automating creation of parts and assemblies can be for internal use to help automate the design process or for external uses, such as sales tools to allow customers to visualize and use your products during their design phase.

About the Speaker:
Brian is the designer of the Autodesk Inventor programming interface. He began working in the CAD industry over 25 years ago in various positions, including CAD administrator, applications engineer, CAD API designer, and consultant. Brian is the original designer of the Inventor API and has presented at conferences and taught classes throughout the world to thousands of users and programmers.

brian.ekins@autodesk.com

Introduction

What is a configurator? The term "configurator" can mean a lot of different things. For this class I'm going to limit the definition to creating a specific version of a part or assembly. Typically, the creation is an automated process that takes a description (part numbers, sizes, etc.) as input and creates the specified part or assembly.

This paper and associated material isn't meant to fully cover the creation of a configurator, but is instead a sampling of various issues I've run into when creating configurators and the solutions I found. Please don't consider any of these as the only or even the best solution to these problems either; they're just the solution I came up with. You may be able to figure out better answers than I did.

I believe there are two basic types of configurators; sales and engineering. A sales configurator is typically used by a salesman or the customer to configure a product and get a representative model that can be used in a design to check for interferences and create between other components in the assembly. A sales model is typically a simplified version of the engineering model.

An engineering configurator is used in-house to generate variations of a product, or sub-systems of a larger product. The intent in this case is to create a model that can be used for the engineering and manufacture of the product. Complete models, and possibly drawings, are required in this case.

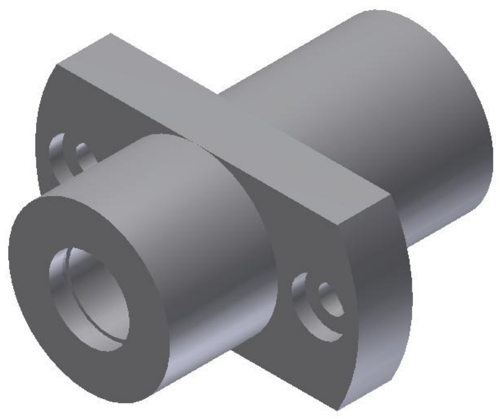
I want to thank Parker Hannifin Corp. for allowing me to use some of their parts as examples.

Part Configuration

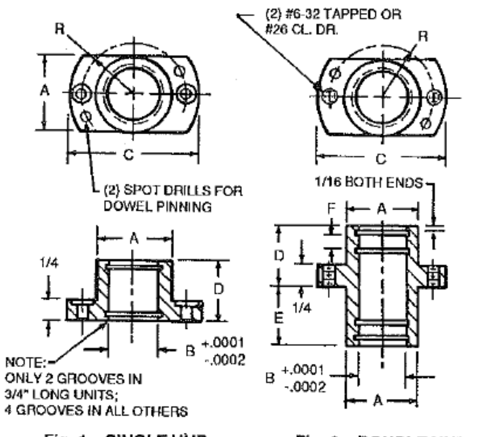
Below, several approaches to creating a part configurator are discussed. The important thing to note is that all of them depend heavily on the parametric nature of Inventor. The ability to parametrically modify a model is extremely powerful and something to take advantage of when possible. The alternative to using parameters is to build a model from scratch. This is possible and the API has the functionality to support it, but it definitely is not as easy as modifying an existing model and will also take more time to process.

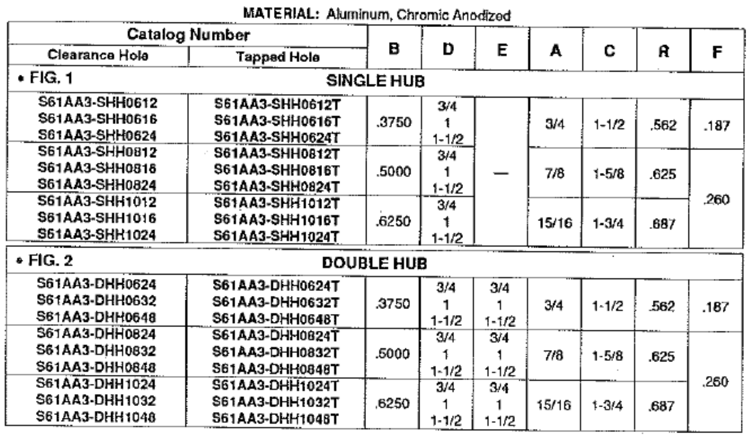
Parameters

Inventor itself can be considered a configurator. Because of its parametric capabilities it's relatively easy to modify a part to represent various configurations of that part by editing parameter values and allowing the part to recompute. To the right, is a simple part I'll use to illustrate various techniques to change part parameters.



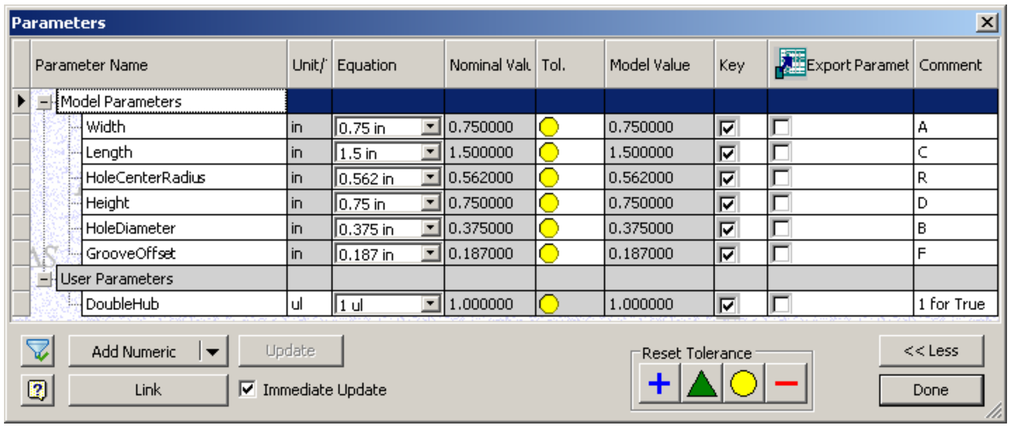
The part and its valid sizes are shown in the table below. There are only specific values that can be used to create a valid variation of this part.


R
A
C
(2) SPOT DRILLS FOR DOWEL PINNING
A
1/4
D
B +.0001 -.0002
NOTE: ONLY 2 GROOVES IN 3/4" LONG UNITS; 4 GROOVES IN ALL OTHERS
(2) #6-32 TAPPED OR #26 CL. DR.
R
C
1/16 BOTH ENDS
F
A
D
E
1/4
B +.0001 -.0002
A
Fig. 1   SINGLE HUB   Fig. 2   DOUBLE HUB


MATERIAL: Aluminum, Chromic Anodized

Catalog Number		B	D	E	A	C	R	F
Clearance Hole	Tapped Hole							
• FIG. 1	SINGLE HUB							
S61AA3-SHH0612 S61AA3-SHH0616 S61AA3-SHH0624	S61AA3-SHH0612T S61AA3-SHH0616T S61AA3-SHH0624T	.3750	3/4 1 1-1/2	—	3/4	1-1/2	.562	.187
S61AA3-SHH0812 S61AA3-SHH0816 S61AA3-SHH0824	S61AA3-SHH0812T S61AA3-SHH0816T S61AA3-SHH0824T	.5000	3/4 1 1-1/2		7/8	1-5/8	.625	.260
S61AA3-SHH1012 S61AA3-SHH1016 S61AA3-SHH1024	S61AA3-SHH1012T S61AA3-SHH1016T S61AA3-SHH1024T	.6250	3/4 1 1-1/2		15/16	1-3/4	.687	
• FIG. 2	DOUBLE HUB							
S61AA3-DHH0624 S61AA3-DHH0632 S61AA3-DHH0648	S61AA3-DHH0624T S61AA3-DHH0632T S61AA3-DHH0648T	.3750	3/4 1 1-1/2	3/4 1 1-1/2	3/4	1-1/2	.562	.187
S61AA3-DHH0824 S61AA3-DHH0832 S61AA3-DHH0848	S61AA3-DHH0824T S61AA3-DHH0832T S61AA3-DHH0848T	.5000	3/4 1 1-1/2	3/4 1 1-1/2	7/8	1-5/8	.625	.260
S61AA3-DHH1024 S61AA3-DHH1032 S61AA3-DHH1048	S61AA3-DHH1024T S61AA3-DHH1032T S61AA3-DHH1048T	.6250	3/4 1 1-1/2	3/4 1 1-1/2	15/16	1-3/4	.687	

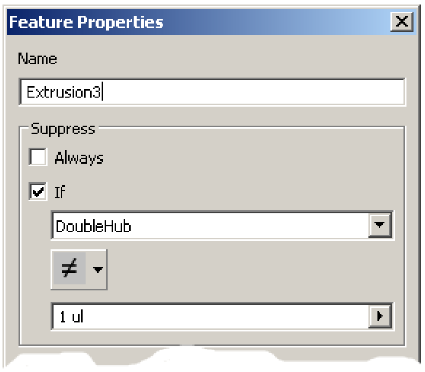
One approach to configuring this part is to use Inventor parameters as shown below. The end-user can use the Parameters dialog to edit the value of any of the parameters and then let the model recompute to get the new configuration. Using multi-value lists for a parameter limits the choices the user has when selecting values, but it's still possible to pick invalid combinations of values that aren't represented in the table.



Parameter Name	Unit/	Equation	Nominal Valu	Tol.	Model Value	Key	Export Paramet	Comment
Model Parameters								
Width	in	0.75 in	0.750000	●	0.750000	☑	☐	A
Length	in	1.5 in	1.500000	●	1.500000	☑	☐	C
HoleCenterRadius	in	0.562 in	0.562000	●	0.562000	☑	☐	R
Height	in	0.75 in	0.750000	●	0.750000	☑	☐	D
HoleDiameter	in	0.375 in	0.375000	●	0.375000	☑	☐	B
GrooveOffset	in	0.187 in	0.187000	●	0.187000	☑	☐	F
User Parameters								
DoubleHub	ul	1 ul	1.000000	●	1.000000	☑	☐	1 for True

☑ Immediate Update

It's also possible to control the suppression state of a feature based on the value of a parameter, which is what's happening with the DoubleHub parameter above. This is controlled through the feature's properties, as shown to the right. When the parameter DoubleHub has a value not equal to 1, the feature will be suppressed. Even though paramaters can be used to configure a part it's not the ideal solution.

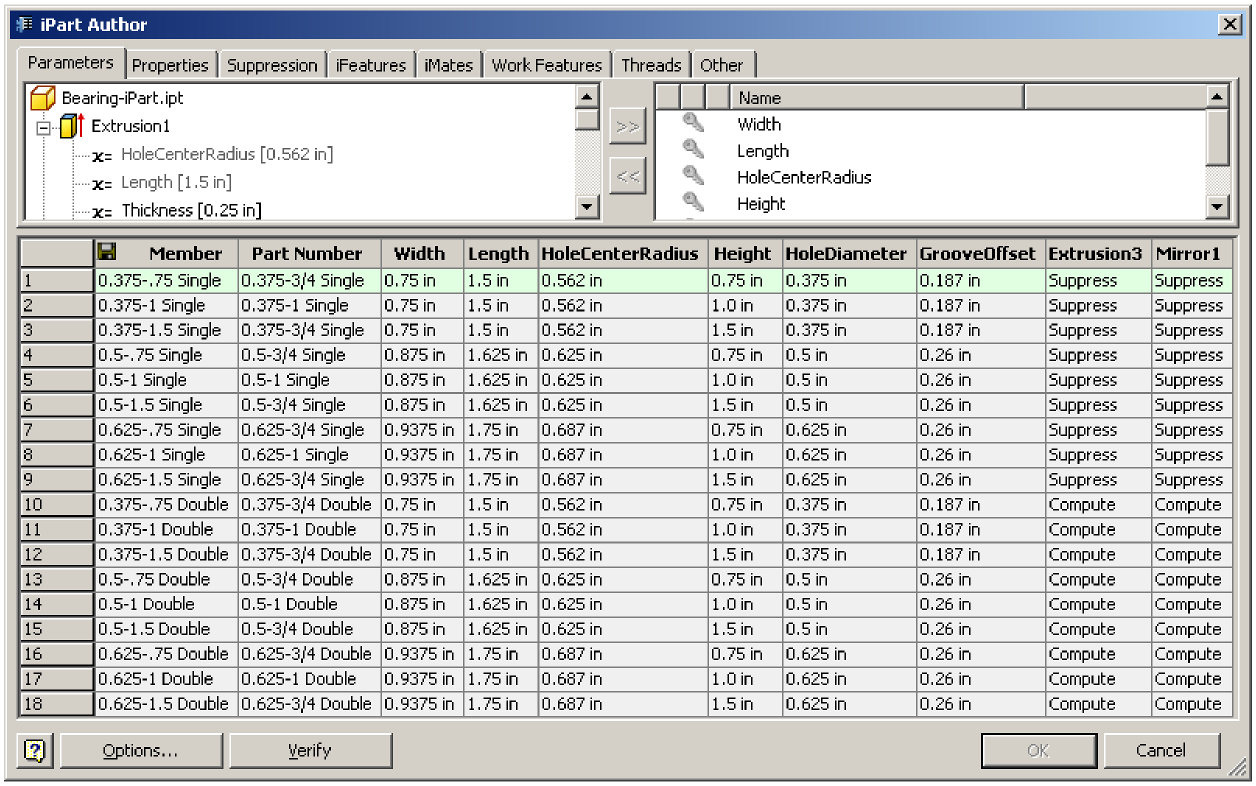


Name: Extrusion3

Suppress
☐ Always
☑ If
DoubleHub
≠
1 ul

iParts

Another approach to configuring parts is to use iParts. With an iPart you can define all of the valid combinations of values in the iPart table, as shown below. It's also possible to have some variable values using a custom iPart.

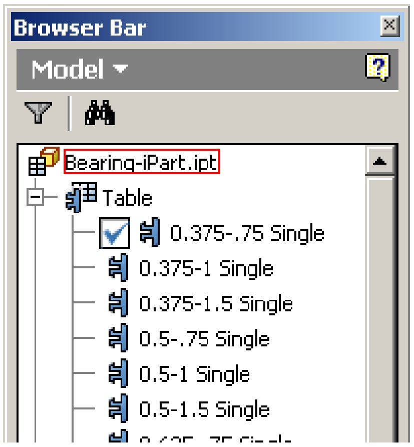


Bearing-iPart.ipt
Extrusion1
HoleCenterRadius [0.562 in]
Length [1.5 in]
Thickness [0.25 in]

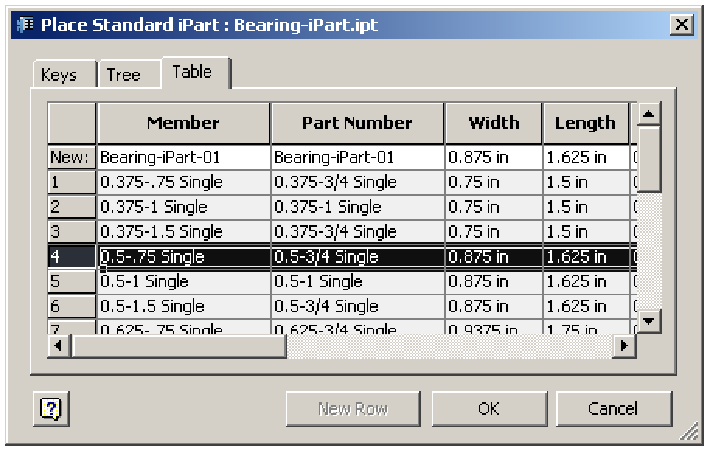
Name
Width
Length
HoleCenterRadius
Height

	Member	Part Number	Width	Length	HoleCenterRadius	Height	HoleDiameter	GrooveOffset	Extrusion3	Mirror1
1	0.375-.75 Single	0.375-3/4 Single	0.75 in	1.5 in	0.562 in	0.75 in	0.375 in	0.187 in	Suppress	Suppress
2	0.375-1 Single	0.375-1 Single	0.75 in	1.5 in	0.562 in	1.0 in	0.375 in	0.187 in	Suppress	Suppress
3	0.375-1.5 Single	0.375-3/4 Single	0.75 in	1.5 in	0.562 in	1.5 in	0.375 in	0.187 in	Suppress	Suppress
4	0.5-.75 Single	0.5-3/4 Single	0.875 in	1.625 in	0.625 in	0.75 in	0.5 in	0.26 in	Suppress	Suppress
5	0.5-1 Single	0.5-1 Single	0.875 in	1.625 in	0.625 in	1.0 in	0.5 in	0.26 in	Suppress	Suppress
6	0.5-1.5 Single	0.5-3/4 Single	0.875 in	1.625 in	0.625 in	1.5 in	0.5 in	0.26 in	Suppress	Suppress
7	0.625-.75 Single	0.625-3/4 Single	0.9375 in	1.75 in	0.687 in	0.75 in	0.625 in	0.26 in	Suppress	Suppress
8	0.625-1 Single	0.625-1 Single	0.9375 in	1.75 in	0.687 in	1.0 in	0.625 in	0.26 in	Suppress	Suppress
9	0.625-1.5 Single	0.625-3/4 Single	0.9375 in	1.75 in	0.687 in	1.5 in	0.625 in	0.26 in	Suppress	Suppress
10	0.375-.75 Double	0.375-3/4 Double	0.75 in	1.5 in	0.562 in	0.75 in	0.375 in	0.187 in	Compute	Compute
11	0.375-1 Double	0.375-1 Double	0.75 in	1.5 in	0.562 in	1.0 in	0.375 in	0.187 in	Compute	Compute
12	0.375-1.5 Double	0.375-3/4 Double	0.75 in	1.5 in	0.562 in	1.5 in	0.375 in	0.187 in	Compute	Compute
13	0.5-.75 Double	0.5-3/4 Double	0.875 in	1.625 in	0.625 in	0.75 in	0.5 in	0.26 in	Compute	Compute
14	0.5-1 Double	0.5-1 Double	0.875 in	1.625 in	0.625 in	1.0 in	0.5 in	0.26 in	Compute	Compute
15	0.5-1.5 Double	0.5-3/4 Double	0.875 in	1.625 in	0.625 in	1.5 in	0.5 in	0.26 in	Compute	Compute
16	0.625-.75 Double	0.625-3/4 Double	0.9375 in	1.75 in	0.687 in	0.75 in	0.625 in	0.26 in	Compute	Compute
17	0.625-1 Double	0.625-1 Double	0.9375 in	1.75 in	0.687 in	1.0 in	0.625 in	0.26 in	Compute	Compute
18	0.625-1.5 Double	0.625-3/4 Double	0.9375 in	1.75 in	0.687 in	1.5 in	0.625 in	0.26 in	Compute	Compute

The end-user is limited to selecting only valid combinations of values by selecting the desired member row from the browser, or the "Place Standard iPart" dialog when inserting the part into an assembly, as shown below.



Bearing-iPart.ipt
Table
0.375-.75 Single
0.375-1 Single
0.375-1.5 Single
0.5-.75 Single
0.5-1 Single
0.5-1.5 Single



	Member	Part Number	Width	Length
New:	Bearing-iPart-01	Bearing-iPart-01	0.875 in	1.625 in
1	0.375-.75 Single	0.375-3/4 Single	0.75 in	1.5 in
2	0.375-1 Single	0.375-1 Single	0.75 in	1.5 in
3	0.375-1.5 Single	0.375-3/4 Single	0.75 in	1.5 in
4	0.5-.75 Single	0.5-3/4 Single	0.875 in	1.625 in
5	0.5-1 Single	0.5-1 Single	0.875 in	1.625 in
6	0.5-1.5 Single	0.5-3/4 Single	0.875 in	1.625 in

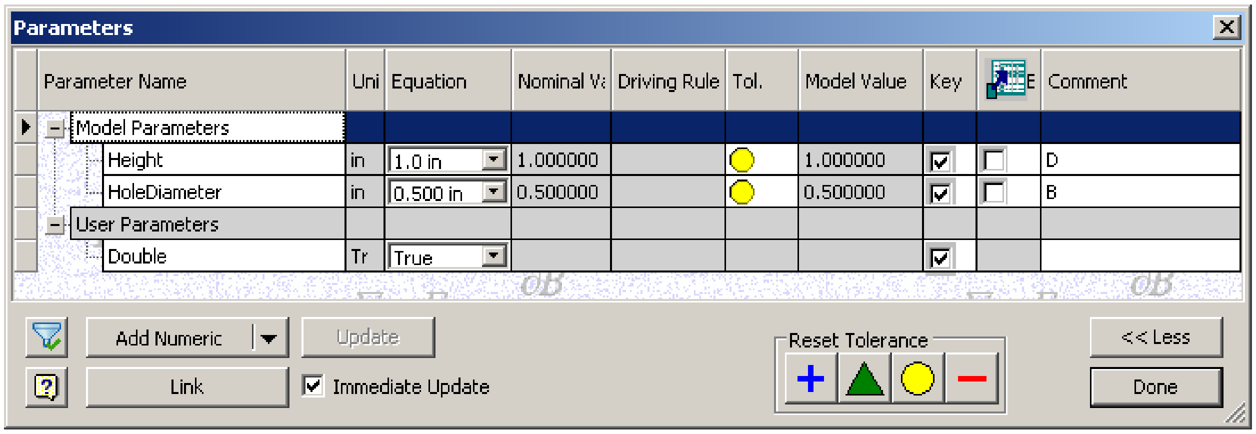
iParts provides a better user-interface and integrates nicely with assemblies but there are some things to be aware of when using iParts that can cause problems with some workflows.

- The table is embedded within the part. This makes maintenance of the part a little more difficult when edits need to be made to the table.
- A new file is created when a part member is created. This is good and bad. It's good because since there is a new file for each member, you can place different members of the same factory into an assembly. If you were using the factory part directly in an assembly, the factory could only represent one member at a time. It's bad because you have additional files to manage.
- When the member file is created, Inventor creates the model for the member by deriving the factory into the member file. Because of this, some information on the original factory model is lost. For example, there aren't any features in the member model. Also any API attributes assigned to the model in the factory will not be copied over to the member model.
- The member can be out of date. If the factory has been changed by modifying the model or editing the values for a specific member, that member will be out of date until it's opened and recomputed.

My personal preference has been to avoid iParts for configurators but you need to look at your specific situation before making a decision to use iParts or not.

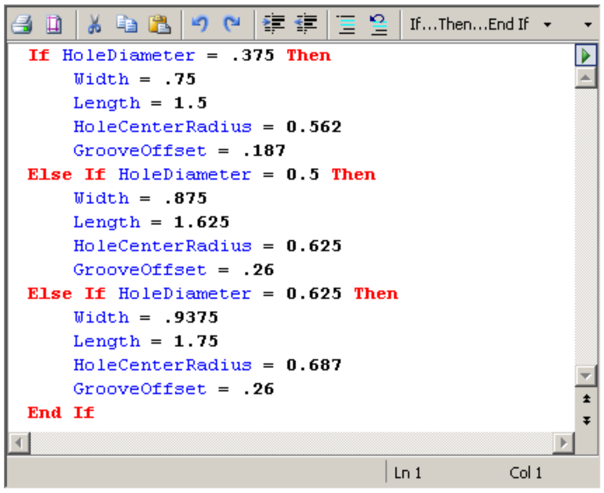
iLogic

Another Inventor tool that can be used to configure parts is iLogic. For the example bearing we've been looking at there are two approaches. One is to embed the table as logic within a rule and the other is to use a table in Excel. Here are the parameters in the part with the filter set to only show key parameters. This particular part has two values that can be changed independent of each other and a setting defining whether it is a single or double bearing. For this particular part, this limits the user to only selecting valid configurations but most cases will be more complex than this so this approach isn't feasible.

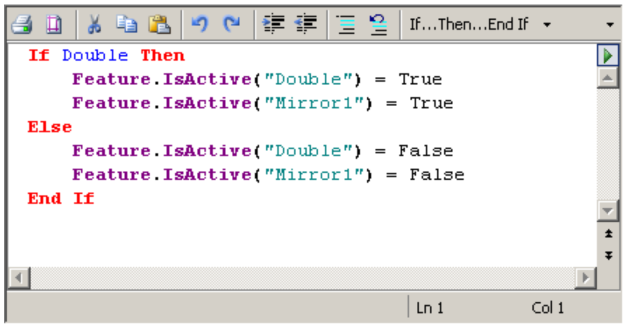


Parameter Name	Uni	Equation	Nominal V	Driving Rule	Tol.	Model Value	Key	E	Comment
Model Parameters									
Height	in	1.0 in	1.000000		●	1.000000	✓		D
HoleDiameter	in	0.500 in	0.500000		●	0.500000	✓		B
User Parameters									
Double	Tr	True					✓		

The picture below shows the two rules that are used to drive this part. The on the left is entirely dependent on the "HoleDiameter" parameter and changes the values of four other parameters based on the current value of "HoleDiameter". The values that were in the iPart table in the previous example are now embedded within the rule. The rule on the right suppresses and unsuppresses two features based on the value of the parameter "Double". I wouldn't recommend defining your table as code. Having the table as code is not very easy to maintain and will be very difficult for someone not comfortable with programming or iLogic to edit.

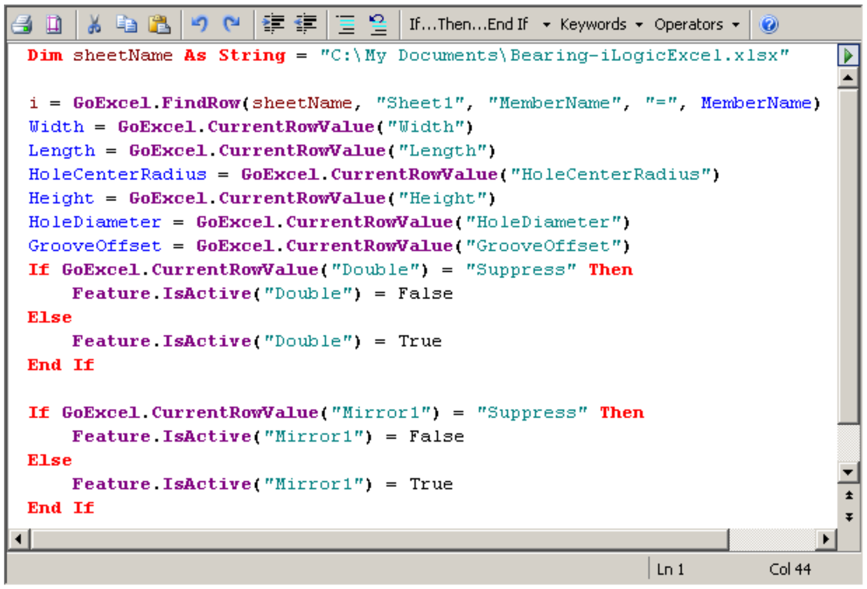


```
If HoleDiameter = .375 Then
    Width = .75
    Length = 1.5
    HoleCenterRadius = 0.562
    GrooveOffset = .187
Else If HoleDiameter = 0.5 Then
    Width = .875
    Length = 1.625
    HoleCenterRadius = 0.625
    GrooveOffset = .26
Else If HoleDiameter = 0.625 Then
    Width = .9375
    Length = 1.75
    HoleCenterRadius = 0.687
    GrooveOffset = .26
End If
```



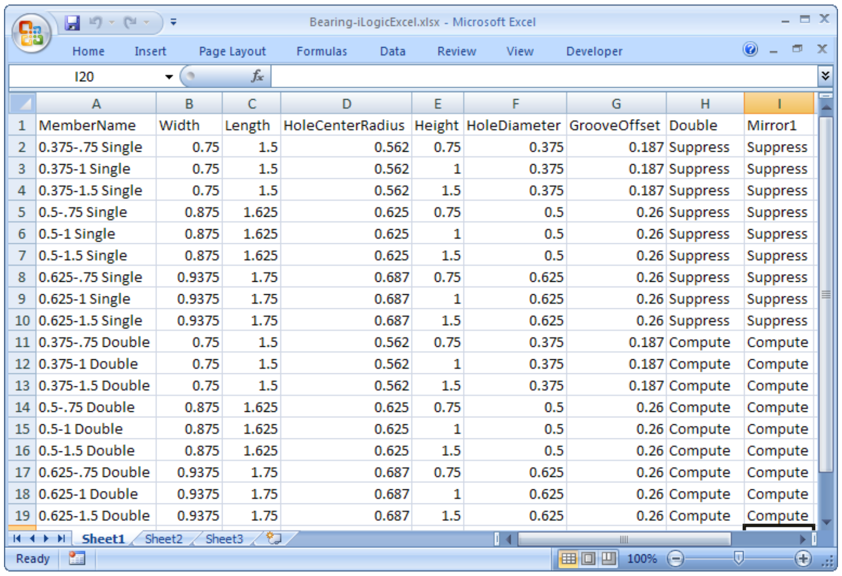
```
If Double Then
    Feature.IsActive("Double") = True
    Feature.IsActive("Mirror1") = True
Else
    Feature.IsActive("Double") = False
    Feature.IsActive("Mirror1") = False
End If
```

Another approach is to combine iLogic with an Excel worksheet. The table data is stored in Excel and used by iLogic to perform the update. The iLogic rule is shown below.

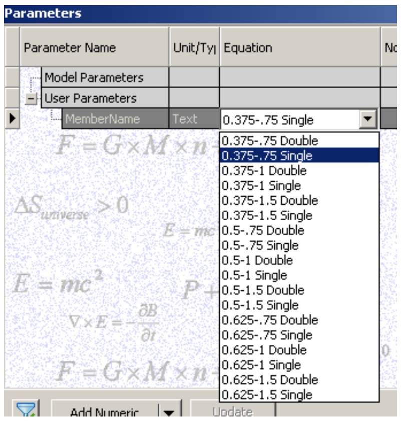


```
Dim sheetName As String = "C:\My Documents\Bearing-iLogicExcel.xlsx"

i = GoExcel.FindRow(sheetName, "Sheet1", "MemberName", "=", MemberName)
Width = GoExcel.CurrentRowValue("Width")
Length = GoExcel.CurrentRowValue("Length")
HoleCenterRadius = GoExcel.CurrentRowValue("HoleCenterRadius")
Height = GoExcel.CurrentRowValue("Height")
HoleDiameter = GoExcel.CurrentRowValue("HoleDiameter")
GrooveOffset = GoExcel.CurrentRowValue("GrooveOffset")
If GoExcel.CurrentRowValue("Double") = "Suppress" Then
    Feature.IsActive("Double") = False
Else
    Feature.IsActive("Double") = True
End If

If GoExcel.CurrentRowValue("Mirror1") = "Suppress" Then
    Feature.IsActive("Mirror1") = False
Else
    Feature.IsActive("Mirror1") = True
End If
```

And here is the table data accessed by the rule and the multi-value list parameter that the rule is dependent on and which provides the user-interface.



	A	B	C	D	E	F	G	H	I
1	MemberName	Width	Length	HoleCenterRadius	Height	HoleDiameter	GrooveOffset	Double	Mirror1
2	0.375-.75 Single	0.75	1.5	0.562	0.75	0.375	0.187	Suppress	Suppress
3	0.375-1 Single	0.75	1.5	0.562	1	0.375	0.187	Suppress	Suppress
4	0.375-1.5 Single	0.75	1.5	0.562	1.5	0.375	0.187	Suppress	Suppress
5	0.5-.75 Single	0.875	1.625	0.625	0.75	0.5	0.26	Suppress	Suppress
6	0.5-1 Single	0.875	1.625	0.625	1	0.5	0.26	Suppress	Suppress
7	0.5-1.5 Single	0.875	1.625	0.625	1.5	0.5	0.26	Suppress	Suppress
8	0.625-.75 Single	0.9375	1.75	0.687	0.75	0.625	0.26	Suppress	Suppress
9	0.625-1 Single	0.9375	1.75	0.687	1	0.625	0.26	Suppress	Suppress
10	0.625-1.5 Single	0.9375	1.75	0.687	1.5	0.625	0.26	Suppress	Suppress
11	0.375-.75 Double	0.75	1.5	0.562	0.75	0.375	0.187	Compute	Compute
12	0.375-1 Double	0.75	1.5	0.562	1	0.375	0.187	Compute	Compute
13	0.375-1.5 Double	0.75	1.5	0.562	1.5	0.375	0.187	Compute	Compute
14	0.5-.75 Double	0.875	1.625	0.625	0.75	0.5	0.26	Compute	Compute
15	0.5-1 Double	0.875	1.625	0.625	1	0.5	0.26	Compute	Compute
16	0.5-1.5 Double	0.875	1.625	0.625	1.5	0.5	0.26	Compute	Compute
17	0.625-.75 Double	0.9375	1.75	0.687	0.75	0.625	0.26	Compute	Compute
18	0.625-1 Double	0.9375	1.75	0.687	1	0.625	0.26	Compute	Compute
19	0.625-1.5 Double	0.9375	1.75	0.687	1.5	0.625	0.26	Compute	Compute

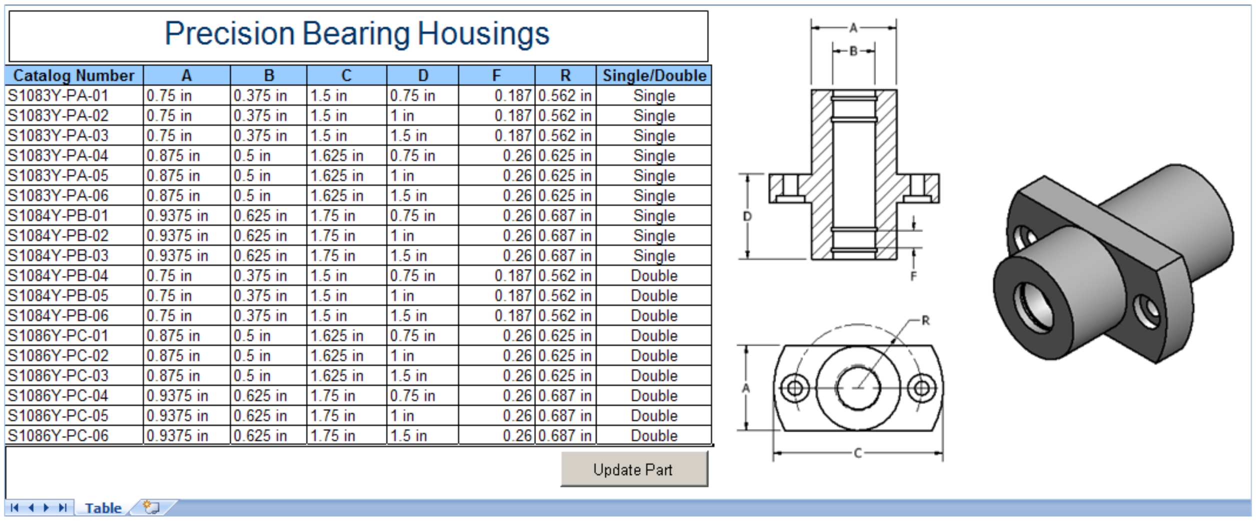


Parameter Name	Unit/Ty	Equation	No
Model Parameters			
User Parameters			
MemberName	Text	0.375-.75 Single	

0.375-.75 Double
0.375-.75 Single
0.375-1 Double
0.375-1 Single
0.375-1.5 Double
0.375-1.5 Single
0.5-.75 Double
0.5-.75 Single
0.5-1 Double
0.5-1 Single
0.5-1.5 Double
0.5-1.5 Single
0.625-.75 Double
0.625-.75 Single
0.625-1 Double
0.625-1 Single
0.625-1.5 Double
0.625-1.5 Single

Excel

Using Excel in combination with Inventor's API is another technique that can be used to drive a table-driven part. In this example, when the button on the form is clicked a VBA macro is executed that reads the values from the select row and uses Inventor's API to edit the associated parameter values and update the part. Excel acts as the user-interface in this case.



Precision Bearing Housings

Catalog Number	A	B	C	D	F	R	Single/Double
S1083Y-PA-01	0.75 in	0.375 in	1.5 in	0.75 in	0.187	0.562 in	Single
S1083Y-PA-02	0.75 in	0.375 in	1.5 in	1 in	0.187	0.562 in	Single
S1083Y-PA-03	0.75 in	0.375 in	1.5 in	1.5 in	0.187	0.562 in	Single
S1083Y-PA-04	0.875 in	0.5 in	1.625 in	0.75 in	0.26	0.625 in	Single
S1083Y-PA-05	0.875 in	0.5 in	1.625 in	1 in	0.26	0.625 in	Single
S1083Y-PA-06	0.875 in	0.5 in	1.625 in	1.5 in	0.26	0.625 in	Single
S1084Y-PB-01	0.9375 in	0.625 in	1.75 in	0.75 in	0.26	0.687 in	Single
S1084Y-PB-02	0.9375 in	0.625 in	1.75 in	1 in	0.26	0.687 in	Single
S1084Y-PB-03	0.9375 in	0.625 in	1.75 in	1.5 in	0.26	0.687 in	Single
S1084Y-PB-04	0.75 in	0.375 in	1.5 in	0.75 in	0.187	0.562 in	Double
S1084Y-PB-05	0.75 in	0.375 in	1.5 in	1 in	0.187	0.562 in	Double
S1084Y-PB-06	0.75 in	0.375 in	1.5 in	1.5 in	0.187	0.562 in	Double
S1086Y-PC-01	0.875 in	0.5 in	1.625 in	0.75 in	0.26	0.625 in	Double
S1086Y-PC-02	0.875 in	0.5 in	1.625 in	1 in	0.26	0.625 in	Double
S1086Y-PC-03	0.875 in	0.5 in	1.625 in	1.5 in	0.26	0.625 in	Double
S1086Y-PC-04	0.9375 in	0.625 in	1.75 in	0.75 in	0.26	0.687 in	Double
S1086Y-PC-05	0.9375 in	0.625 in	1.75 in	1 in	0.26	0.687 in	Double
S1086Y-PC-06	0.9375 in	0.625 in	1.75 in	1.5 in	0.26	0.687 in	Double

API

Writing an external application that uses the API to configure the desired part is also possible. The Excel example above is a variation of this. By writing a fully custom application you can create any type of user-interface you want including a command line interface, windows dialog, or even a web based interface. You're free to choose whatever works the best for your specific case. We'll look more at API specific solutions when we look at assembly configurators.

As mentioned above, another option when using the API is to dynamically create or edit a model rather than editing parameters. It is possible, but not suggested because of the complexity and performance issues. But in a few cases it still might be the best approach and is an option.

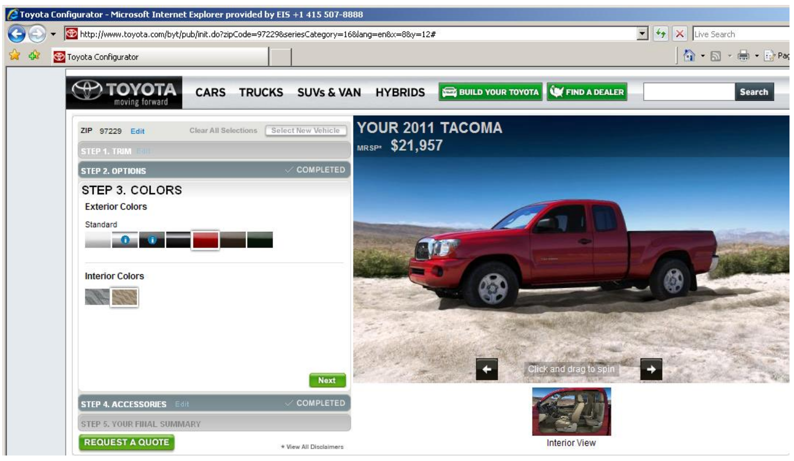
The User-Interface

There are two distinct parts of a configurator; the user-interface that collects the input from the user and the back-end that creates the model. The user interface is a critical component and typically contains much of the overall logic. It's the user interface that has to know what's valid and enforce the rules as the user interacts with the user interface to specify the component they want. The result after the user has finished with the user interface is a description of the desired component. The back-end then takes this description as input and creates the model.

The user-interface and the back-end can be part of the same program or they can be separate programs running on different machines and commuting over a network or internet. Here are some examples of some configurator user interfaces.

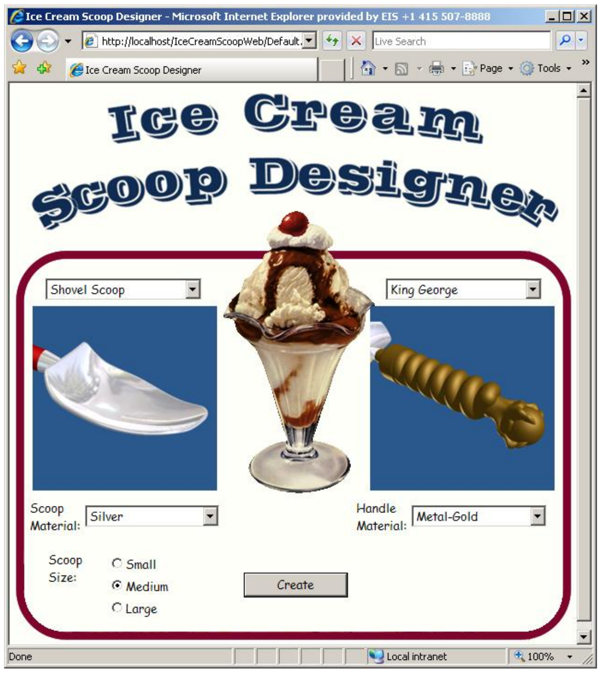
Inventor

The simplest user-interface to implement is to just use Inventor itself. This is the approach used in most of the previous examples and makes it easy to implement but is typically not very user-friendly, especially in the case where someone without Inventor experience or even access to Inventor needs to use it.



This next example doesn't have anything to do with Inventor but illustrates a web-based user interface for a configurator. Even though this configurator doesn't result in the creation of an Inventor model, it shares many of the same concepts as an Inventor based configurator; a user is guided to specify a valid configuration and returns the configured result. To help the user understand what he's configuring, the display of the truck updates dynamically as different choices are selected. The final result is a picture of the configured truck and a description of the vehicle that can be sent to your nearest Toyota dealer.

Below is a simple web-based configurator that works with Inventor to create a model of an ice cream scoop. The user sees images representing the various components they're selecting. Once they've configured the model they want, a description is generated (as xml) and passed to the back-end that reads the description and creates the specified model, which is then passed back to the user.


Ice Cream Scoop Designer - Microsoft Internet Explorer provided by EIS +1 415 507-8888
http://localhost/IceCreamScoopWeb/Default.
Live Search
Ice Cream Scoop Designer
Page
Tools
Ice Cream Scoop Designer
Create
Done
Local intranet
100%

Shovel Scoop

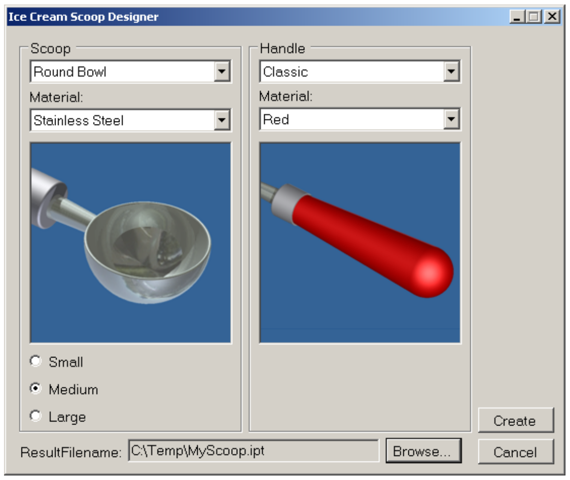
King George

Scoop Material: Silver

Handle Material: Metal-Gold

Scoop Size: ( ) Small [✓ Medium] ( ) Large

Below is an example of another user-interface to configure an ice cream scoop. This one is a Windows form. Configuring a scoop results in the same xml data as the web-based configurator so the back-end that processes the model can be the same.



Scoop: Round Bowl

Material: Stainless Steel

Handle: Classic

Material: Red

( ) Small [✓ Medium] ( ) Large

ResultFilename: C:\Temp\MyScoop.ipt

Even though the user interface is a critical part of any configurator, I'm not going to spend a lot of time discussing it here. Designing and creating the user interface is typically very independent of Inventor and will typically require some programming. It's the use of Inventor as part of the configurator's back-end that I want to focus on. Creating a good user interface will require reasonable design and programming skills. I don't have any tips or tricks to building your front-end, except to look at all of the existing conifigurators that you can find and decide what you like and don't like about each one and incorporate what you do like into yours.

Associative or Not?

Typically, when designing with Inventor a primary goal is for the entire model including any associated drawings to be fully associative. This means you should be able to edit a part and have the assembly and any related drawings correctly update. When creating a configurator you need to carefully consider whether associativity is needed in the final result. In most cases it's not needed and by not worrying about associativity you can often simplify the problem. For example, instead of using assembly constraints to position parts in an assembly there are some cases where it's easier to compute the desired position of the part and place it directly. With drawings it's also possible to create dimensions to a location in space without worrying about how it attaches to the model.

Typically for a sales automation configurator, associativity is not needed because the final model and drawing are considered static and the user will not be making changes. For an engineering configurator you have to consider whether the result of the configurator is the final design or not. If there will be continued design work done using the result then you will likely want full associativity. If the result is the final design to be used for manufacturing then associativity probably doesn't matter.

Even in the cases where you don't care about the final result being associative, you will still take advantage of Inventor's associativity. As we've already seen, the parametric nature of Inventor can make it easy to create different variations of a part by just editing parameters and suppressing features. In assemblies you can also edit parameters and suppress occurrences to create different versions of the assembly.

The primary question when considering associativity is can the different variations of the model you're configuring be obtained by editing parameters, suppressing features, and disabling parts in assemblies? If you can answer yes to this then it may be possible to use the parameter editing techniques described above, along with iAssemblies, iLogic, and possibly some API routines to modify an existing model to obtain the desired configuration.

In other cases it's not practical to create the final result by editing an existing part or assembly because there are so many possible combinations. In this case the solution is to construct the assembly from scratch. You still need to determine whether the result needs to be associative or not, but creating a non-associative result is simpler.

Parametric Parts or Not?

The previous section on configuring parts relied on the fact that Inventor is a parametric modeler and is a powerful tool that can be used to create many variations of a part. However, in many cases you don't need to be able to create different parts on the fly but can use a library of static parts that the configurator assembles in different ways.

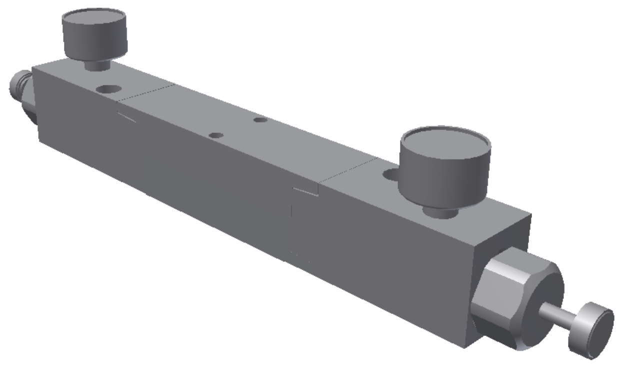
Even though the parts might be parametric it doesn't mean you need to take advantage of that. For example, there might be a part that has four different variations. Rather than use a single part and modify it each time to create the version you need, you can create four different part files, one for each variation. You'll have more files on disk, but the configurator will be simpler and will also run faster because you can skip the part configuration step.

Having static parts also simplifies the problem where you might need instances of different variations of a part. For example, if you have a part that is available in 4 different lengths and you need 3 of those lengths in the assembly you're building, you'll need to create copies of the original parametric part, one for each variation so you have a unique part for each variation. Creating these files will take time and increases the complexity of the configurator by having to manage the files.

The primary issues to consider when deciding whether to use parametric parts are:

- How many variations of the part will there be? If there's a small number then I would suggest having multiple static parts.
- Will different variations of the part be used in the same final assembly? If there are a reasonable number of variations of parts, then this would push the decision further towards having multiple static parts.
- Is your source part parametric? Maybe you're getting some of your parts from another CAD system and they're not parametric to begin with. If they're not going to change, there's no reason to rebuild them in Inventor so they are parametric but just translate them into Inventor as use them as static parts.

Pre-build Subassemblies

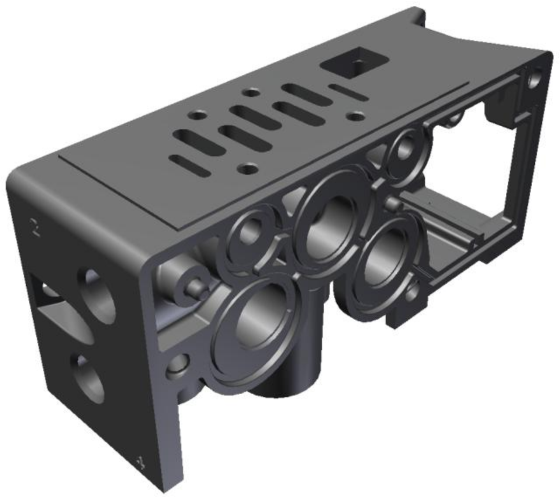


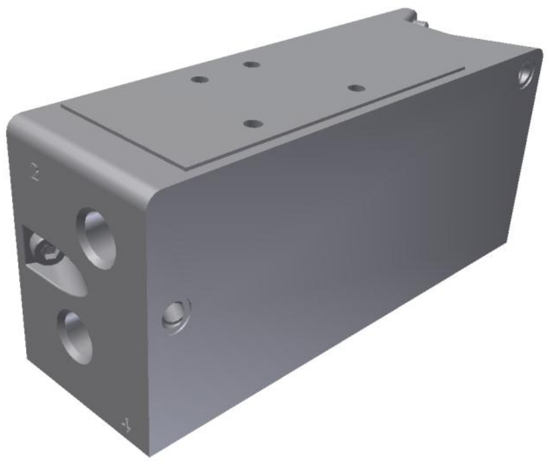
If there are portions of an assembly that are always built up the same way you can pre-build those and have them available as finished assemblies to place into the final top-level assembly. Even better than using a pre-built assembly is to create a derived part from the subassembly so that the subassembly is represented by a single part. This will eliminate some files and speed up processing. The part to the right is a good example of combining several parts into a single these parts are always assembled in the same way. Using this approach will affect the BOM, but for most sales automation configurators that doesn't matter since you don't need to generate a full engineering BOM.

Simplify Your Parts and Assemblies

This is a shortcut reserved exclusively for sales automation configurators since you'll want to maintain full part detail in an engineering configurator. There are two advantages that come from part simplification. First is that you hide some of the details and possible intellectual property of the part and second is that the resulting model will be smaller and simpler to display, resulting in better performance.

A great example of part simplification is a manifold. For a sales configurator, it's only the outside of the manifold that's important as far as the customer being able to correctly position the part within their assembly, connect it to other equipment, and check for interference. The interior cavities aren't needed and can be represented by simple holes to show where the connection points are. The pictures below show the full manifold model on the left and the simplified model on the right. When this part is used in an assembly, it is always sandwiched in between other parts, so the fact that some internals are missing can't even be seen. The front of the manifold is what the customer sees and where they will connect to so it so the details are left there but all of the other detail has been removed. The holes on the top and side are there as handles to use in positioning this part and other parts relative to this part.

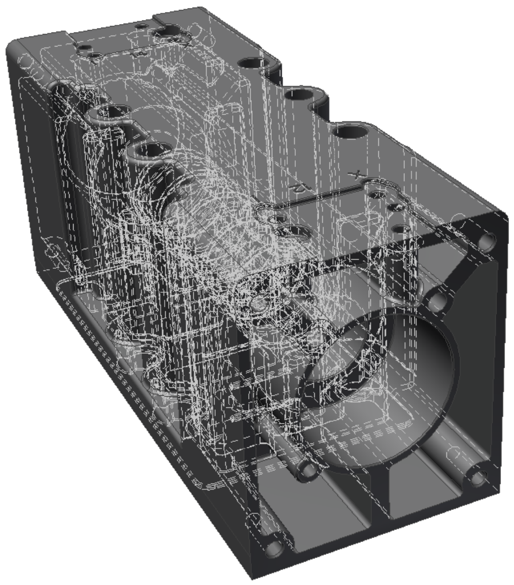


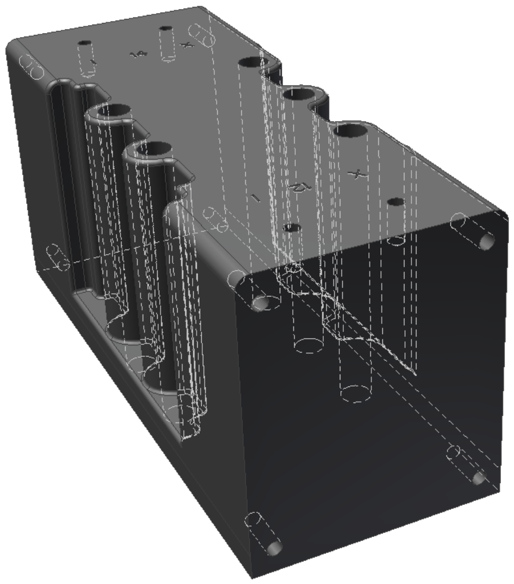


The part on the right is created by deriving an assembly. You might notice a bolt in the hole on front of the part on the right. It is used to fasten this manifold to the adjacent manifold. Since the bolt is always needed, it's been included into this part. Simplifying by removing geometry and combining parts into a single part reduces the number of parts and the geometric complexity. The original part on the left is about 5.2 MB in size. The part on the right is about 0.6 MB.

Using the Derive command to create the final part is important for two reasons. First it lets you create a single part from an assembly and second it creates a simpler model because the derived part only contains the geometry and not any of the other data used to create it, like sketch and feature information.

Below, on the left, is a full featured engineering model that has a lot of detail internally that is not needed in a sales configurator. The model to the right is the simplified version. The goal in creating the simplified version is to have the model look the same on the outside but remove all other geometry. Obviously, this eliminates the internal geometry but can also simplify other areas too. For example, the end of the part facing you in the picture below will always be covered with another part. The only reason to even have the holes that are seen is for attaching other parts to this one. All of the detail on the outside of the part has been left unchanged so that the part will be recognizable to the user.

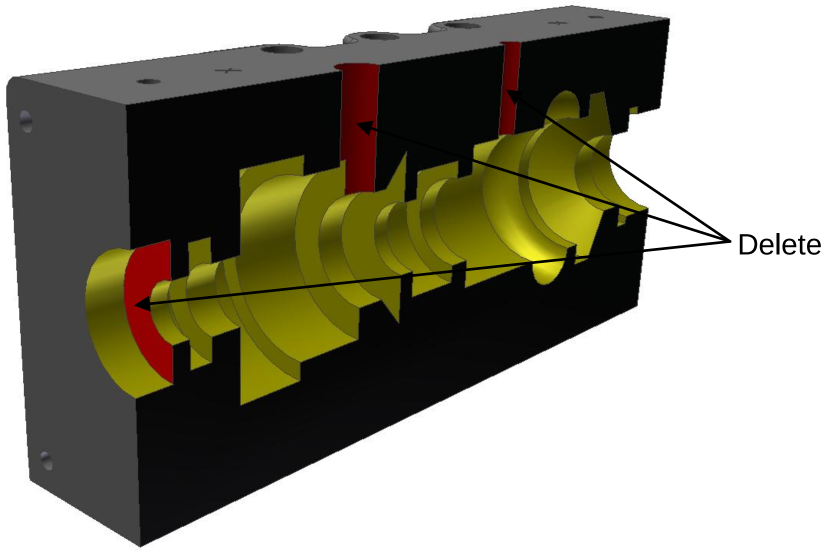




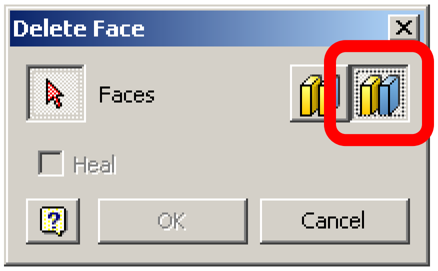
Here are a couple of tips for simplifying geometry. If the part was originally created in Inventor you can likely simplify a lot of the geometry by just suppressing some of the features. Once you've suppressed the as many features as possible you can, or if the part wasn't created in Inventor, the next step is to use the **Delete Face**, **Boundary Patch,** and **Stitch** commands to further simplify the part.

The first step is to look at the part and determine what geometry you want to remove. There are two things to consider in this step. First, what can be removed but still allow the part to look correct? Second, what geometry needs to remain so that you can use it to attach other parts? As discussed below, I always use holes (circular edges) as the attachment points when assembling parts. In many cases, rather than try and preserve the original holes during the simplification process, it's just easier to recreate them as a final step. To make the creation of the holes easy, I create a sketch on the face that contains the hole before I start simplifying the model. This creates geometry that I can use later to re-create the hole in the correct location.

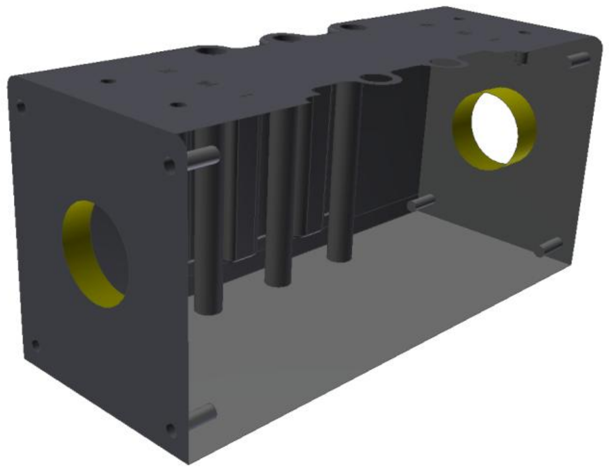
The next step is to use the **Delete Face** command to delete all of the faces that connect the geometry you want to delete to the geometry you want to keep. The simple model below illustrates this concept. (It's been sectioned to easily view the internals.) Even though there's quite a bit of internal geometry in this part, there are only four areas where the internals connect to the outer faces. If I use the **Delete Face** command to delete the three red faces shown below and a fourth that's not visible in this view, it will disconnect the inside from the outside. The "Heal" option on the Delete Face dialog can also be handy for automatically filling in holes when you delete faces. For example, deleting the two red faces that connect to the top face with the heal option checked will automatically fill the holes.

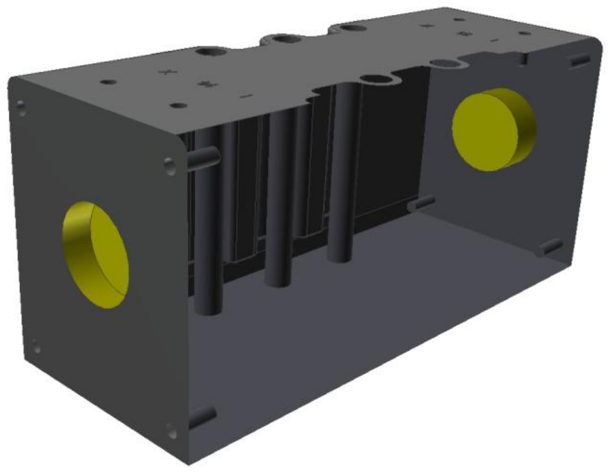


Now that the internals are disconnected from the outside you can delete the internals in one step using the **Delete Face** command. Choose the "Select lump or void" button on the right-hand side of the dialog to select connected sets of faces as one, as shown to the right. This setting in the **Delete Face** command is also a useful tool in determining if you have completely disconnected the internals or not. If the entire solid highlights as one, then there's still something connecting the inside to the outside.



The picture on the left shows the model (with the front removed for clarity) with the insides removed. The next step is to seal up the holes. In some cases, like here, I still want to show the location of the holes, but don't need all the internal detail. That's why I left a portion of the cylinder that's connected to the outside. To cap these you can use the **Boundary Patch** command and then the **Stitch** command to combine it all back into a solid.

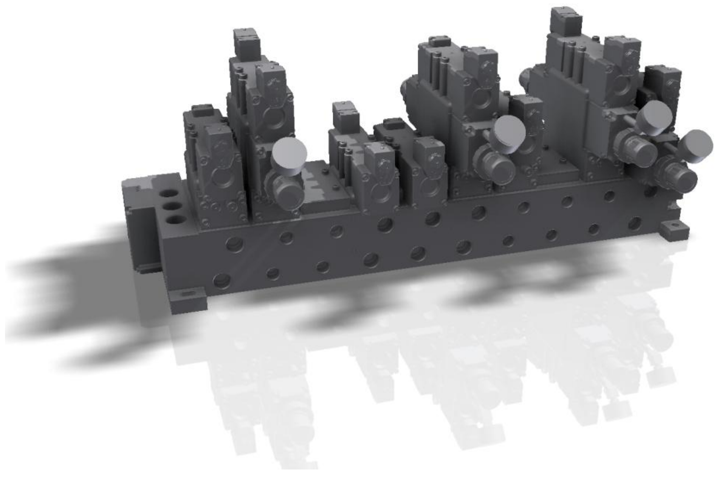




For some holes you may need to recreate them once you've finished simplified the model.

Constructing your Assembly

To configure an assembly, you can have a pre-built assembly that you modify by editing the parts and subassemblies and then let Inventor update it. That approach doesn't work when the assembly can radically change. The assembly below is an example of this. The user can choose from 1 to 24 stations with each station having many options. For this assemblies like this it makes more sense to build it up from scratch.



Constructing any assembly is done by placing one part at a time. The construction process simplifies down to two easy steps; determining which part to place and then positioning it correctly. There are at least three approaches to positioning a part within an assembly. The first is to calculate its position, construct a matrix that defines that position, and place the part using the matrix. This has the advantage of placing and positioning the part in a single step and doesn't require Inventor to compute location using constraints. However, this isn't very practical for most configurators because it would take quite a bit of coding and logic to be able to compute the position of all of the parts in the assembly and this logic could be difficult to maintain if small changes are made to the parts.

Another approach is to use iMates and let Inventor automatically assemble them by finding matching iMates. This may work in some cases, but in my experience this isn't a good solution for a configurator.

The solution that I've used the most is to name geometry in each part and then after the part has been placed into the assembly, find the named geometry and use it to create assembly constraints. This is going to seem to contradict the previous paragraph, but I've typically used iMates as the naming mechanism. However, I didn't use iMates the way they were intended; to automatically create constraints, but only used them as a way to associate a name with geometry. I chose iMates to do this because they are a standard part of Inventor and the user interface to create and edit them already exists. I could have chosen to write a custom program that added an attribute to selected geometry, but then I would have had to write a program to add, view, and edit these names.

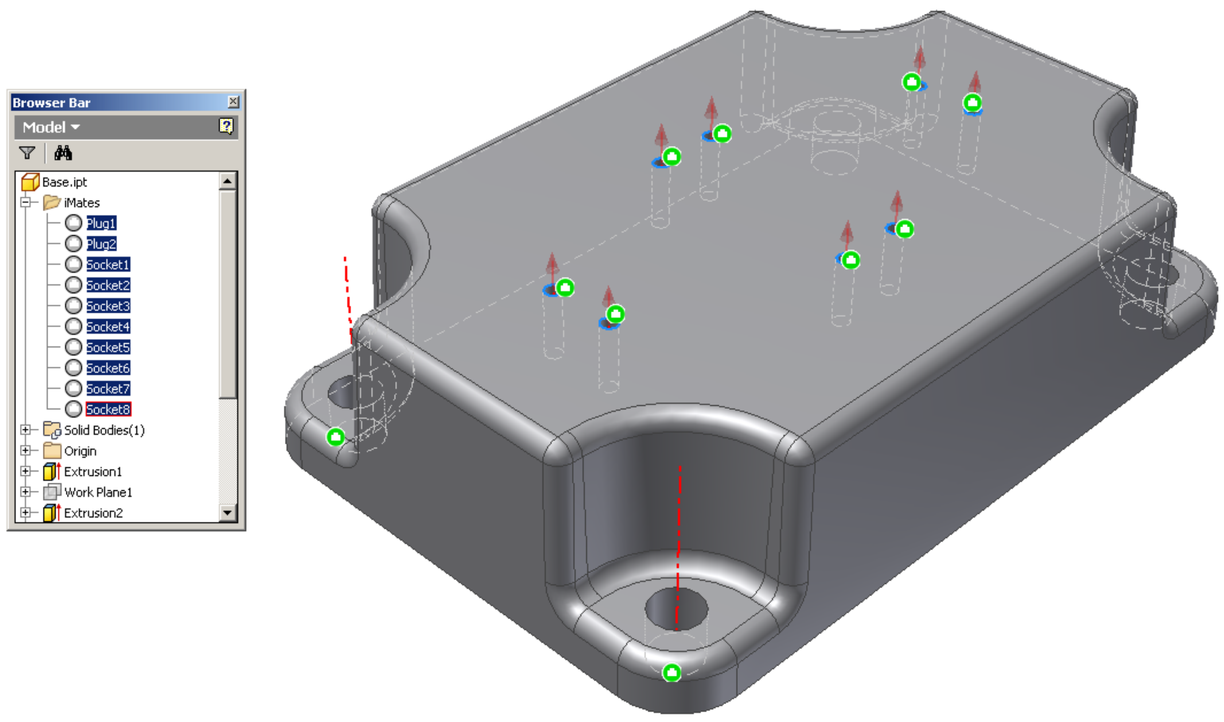
What has worked well for me when assembling parts is to envision putting a plug into a socket. For a part what would have other parts attached to it, I name geometry "Socket1", "Socket2", etc. for each attachment point. On the part that will be attached I named geometry "Plug1", "Plug2", etc. "Plug1" doesn't have to match up with "Socket1" when the assembly is put together. That's one of the advantages of not using iMates in the standard way; they can be assembled in any way you want without limitations because of the name or type.

For consistency, I've chosen to always use Insert iMates. Insert iMates are always associated with a circular edge so I know what to expect when I get the associated geometry.

So you don't have to see the iMate symbols on the part or assembly, you can suppress the iMates. Since Inventor isn't going to use these to create a constraint it doesn't matter if they're suppressed or not since the API can still access the associated geometry you need to be able to create a constraint.

Connecting two parts together can be accomplished using one or two insert constraints. One constraint can be used if you don't care about its rotation, like a bolt, and two connect constraints can be used to fully constrain it.

Here's an example that illustrates this. This picture below shows the model and the ten iMates it contains. By looking at the names in the browser, you can see there are two plugs and eight sockets. The two plugs are used to position this part when it's placed into an assembly as they are plugged into sockets on another part. The sockets on this part are used to attach other parts to this one.



The front-end of the configurator lets the user define the assembly they want, creates a description of the assembly and passed it to the back-end which builds up the assembly by placing the specified parts and constraining them to one another. The knowledge of what needs to be placed, and how they connect is logic within the back-end of the configurator.

Here are the VB.Net utility functions that make this socket-plug concept relatively easy to implement. The PlugIn function takes two occurrences and the names of the imates within each occurrence that you want to connect. The only assumption is that the iMates used are insert iMates. The function creates an insert constraint between the two occurrences and returns the new constraint, if it was successful. The GetEdgeByName function is used by the first to find an iMate using a name.

```
' Utility function that plugs one part into another by placing constraints between
' the two parts by using geometry that has been named by an iMate.
'  TopAssembly - The top-level assembly where the parts are being assembled.
'  SocketOcc - The base part the plug part is being attached to.
'  SocketName - The name of the iMate that is attached to a circular edge.
'  PlugOcc - The part that is being attached to the socket part.
'  PlugName - The name of the iMate that is attached to a circular edge.
'  Reversed - Indicates if the normals of the geometry should be reversed or not.
'  Offset - Offset distance between the two circles.
Public Function PlugIn(ByVal TopAssembly As AssemblyDocument, _
                       ByVal SocketOcc As ComponentOccurrence, _
                       ByVal SocketName As String, _
                       ByVal PlugOcc As ComponentOccurrence, _
                       ByVal PlugName As String, _
                       Optional ByVal Reversed As Boolean = True, _
                       Optional ByVal Offset As Double = 0) As InsertConstraint
    Try
        ' Get the edge in the plug occurrence.
        Dim plugEdge As Edge = GetEdgeByName(PlugOcc, PlugName)

        ' Get the edge of the socket.
        Dim socketEdge As Edge = GetEdgeByName(SocketOcc, SocketName)

        ' Create an insert constraint.
        Dim insert As InsertConstraint
        insert = TopAssembly.ComponentDefinition.Constraints.AddInsertConstraint( _
                                             socketEdge, plugEdge, Reversed, Offset)
        Return insert
    Catch ex As Exception
        ' Something failed so return Nothing.
        Return Nothing
    End Try
End Function

' Find an iMate by name in the document referenced by the provided occurrence.
' It returns a proxy of the edge that the iMate is attached to.
Private Function GetEdgeByName(ByVal Occ As ComponentOccurrence, _
                               ByVal Name As String) As Edge
    ' Iterate through all of the imates in the document referenced by the occurrence.
    For Each iMateDef As iMateDefinition In Occ.Definition.iMateDefinitions
        ' See if the name of this iMate definition matches the input name.
        If UCase(iMateDef.Name) = UCase(Name) Then
            ' Make sure it's an insert iMate.
            If TypeOf iMateDef Is InsertiMateDefinition Then
                Dim insertiMateDef As InsertiMateDefinition = iMateDef

                ' Get the edge referenced by the iMate.
                Dim iMateEdge As Edge = insertiMateDef.Entity

                ' Create a proxy for the edge.
                Call Occ.CreateGeometryProxy(iMateEdge, iMateEdge)

                Return iMateEdge
            End If
        End If
    Next

    ' No match was found so return Nothing.
    Return Nothing
End Function
```

Here is some example code that exercises the function above. The SampleConfigurator example in the files associated with this paper demonstrates these concepts.

```
Private Sub BuildAssembly()
    ' Connect to a running instance of Inventor.
    Dim invApp As Inventor.Application = GetObject(, "Inventor.Application")

    ' Set the path where the library files are located.
    Dim libPath As String = "C:\My Documents\SampleParts\"

    ' Create a new assembly document.
    Dim asmDoc As AssemblyDocument
    asmDoc = invApp.Documents.Add(DocumentTypeEnum.kAssemblyDocumentObject, _
         invApp.FileManager.GetTemplateFile(DocumentTypeEnum.kAssemblyDocumentObject))
    Dim asmDef As AssemblyComponentDefinition = asmDoc.ComponentDefinition
    Dim tg As TransientGeometry = invApp.TransientGeometry

    ' Add the first part.
    Dim base As ComponentOccurrence
    base = asmDef.Occurrences.Add(libPath & "Base.ipt", tg.CreateMatrix)

    ' Turn off user interaction.  This will improve performance for an
    ' out of process application.
    invApp.UserInterfaceManager.UserInteractionDisabled = True

    ' Add a pump to the assembly.
    Dim pump As ComponentOccurrence
    pump = asmDef.Occurrences.Add(libPath & "Pump.ipt", tg.CreateMatrix)

    ' Plug the pump into the base.
    Dim connect As InsertConstraint
    connect = PlugIn(asmDoc, base, "Socket1", pump, "Plug1", True)
    connect = PlugIn(asmDoc, base, "Socket2", pump, "Plug2", True)

    ' Add a filter to the assembly.
    Dim filter As ComponentOccurrence
    filter = asmDef.Occurrences.Add(libPath & "Filter.ipt", tg.CreateMatrix)

    ' Plug the filter into the pump.
    connect = PlugIn(asmDoc, pump, "Socket1", filter, "Plug1")
    connect = PlugIn(asmDoc, pump, "Socket2", filter, "Plug2")

    ' Add a second filter.
    filter = asmDef.Occurrences.Add(libPath & "Filter.ipt", tg.CreateMatrix)

    ' Plug the filter into the pump using some other sockets.
    connect = PlugIn(asmDoc, pump, "Socket5", filter, "Plug1")
    connect = PlugIn(asmDoc, pump, "Socket6", filter, "Plug2")

    ' Add a gauge to the assembly.
    Dim gauge As ComponentOccurrence
    gauge = asmDef.Occurrences.Add(libPath & "Gauge.ipt", tg.CreateMatrix)

    ' Plug the gauge into the filter.
    connect = PlugIn(asmDoc, filter, "Socket1", gauge, "Plug1")

    MsgBox("Finished.")
End Sub
```

Providing the Model to the End User

For a sales configurator there can be special requirements about the final result. A typical option will be to allow the user to choose the format of the final model; Inventor, STEP, SAT, etc. These are easy to provide by using any of the translators that are delivered with Inventor to translate the final Inventor model into whatever format they choose.

Another option when working with assemblies is to convert the assembly into a single part. This can be convenient for you and your customer since you'll both only have to deal with a single file. You can convert an assembly into a part by creating a new part and using the **Derive Component** command to derive the assembly into the part. There are several options in the **Derive Component** command that can be useful. One of these is to keep the seams on planar faces. This will result in the final part model looking like the original assembly. You should look at the other options too, including some of the shrink-wrap options to see what provides the best result in your case.

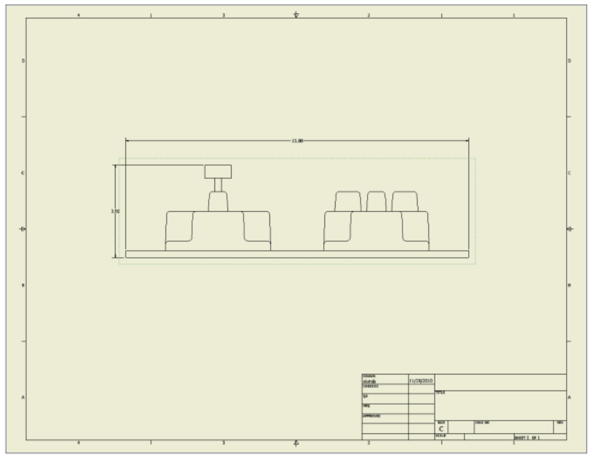
There are cases where you need to provide an assembly rather than a single part. For example, if the parts move relative to one another and the user needs to be able to reproduce this behavior in their assembly. To provide multiple parts, especially if you have a web interface, it's best to package them into a single file. I've used the command line interface for WinZip to do this in the past. The command line utility is available as a free download to work with your licensed copy of WinZip.

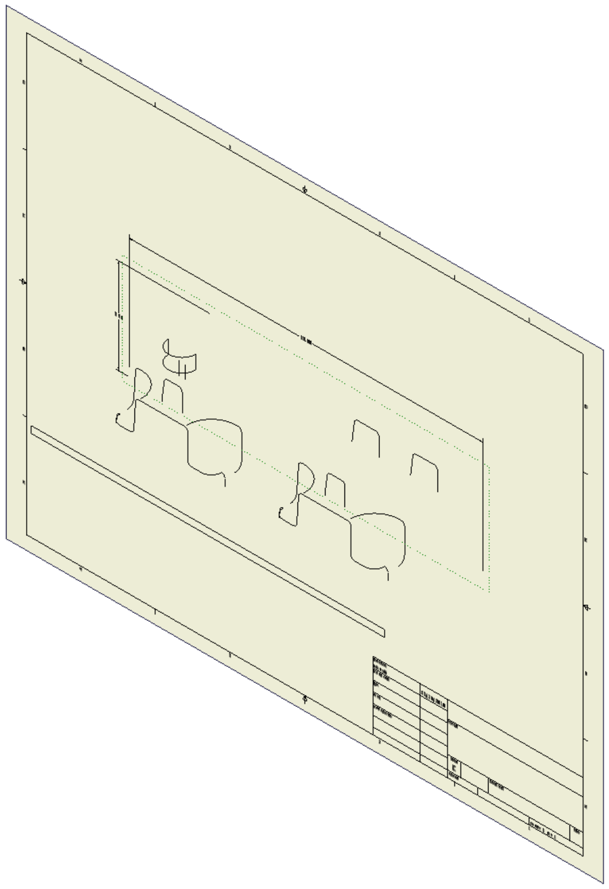
Creating Drawings

The automatic creation of drawings is typically more difficult than creating parts and assemblies. This is a combination of how drawings work in Inventor, the fact that everything you can do interactively in a drawing is not supported by the API, and that there are so many subjective decisions made about how to lay out a drawing.

The biggest thing to decide up-front is if the resulting drawing needs to be associative to the model or not. You'll make your life much easier if it doesn't need to be associative. Having said that, most things will be associative because that's the only way to create them in Inventor, but there are some things that will be easier if you don't have to worry about associativity; dimensions being the primary one.

The primary issue with creating dimensions, and other annotations, through the API is that it's very difficult to determine the geometry that you want to dimension to. A drawing view is a cleaned up representation of the 3D model. This is illustrated in the pictures below. The picture on the left is the drawing as you would typically see it. The one on the right shows the drawing as an iso view so you can see the view geometry as it really exists.





A large part of the clean-up is removing all of the edges that are not visible with respect to particular view. For example, when you create a drawing view of a box, there are only four edges seen in the view. The edges that are behind the visible edges aren't just covered up, but have been removed from the view and are not available. Because of this, you can't rely on certain geometry always being available in a drawing. For example, if you added attributes to two edges in a model because you know you need to create a dimension between them in the drawing, you may not be able to find one or both of those edges in the drawing because they may have been removed as part of the view clean up.

Below are some tricks I've discovered to work around some of these issues when creating drawings.

Retrieving Dimensions from the Model

One possibility to automate the dimensioning of a model is to carefully create dimension constraints in the model and then use the API to retrieve those into the model. If you create the dimensions in the part (typically as dimension constraints in a sketch) taking into account the dimensions that will be needed in the drawing, you can re-use these dimensions in the drawing by retrieving them.

There are many ways to approach this and the code below demonstrates one approach. This example takes a drawing view and a string as input. What it expects is that the dimensions you want to retrieve have been previously marked with an attribute. Specifically it looks for attribute sets named “GetDim” and then looks for the attribute within that set named “View” that has the same value as the string that is passed in. I’m using the string to specify which view orientation that dimension should be retrieved for. For example “Front”, “Top”, “Left”, etc. If you create a drawing of the SampleParts\Pump.iam part and run the RetrieveTest macro it will demonstrate this.

```
Public Sub AutoRetrieveDim(drawView As DrawingView, ViewName As String)
    ' Get the document associated with the drawing view.  Can be a part or assembly.
    Dim viewDoc As Document
    Set viewDoc = drawView.ReferencedDocumentDescriptor.ReferencedDocument

    Dim sh As Sheet
    Set sh = drawView.Parent

    Dim tg As TransientGeometry
    Set tg = ThisApplication.TransientGeometry

    ' Get the dimensions to retrieve in the front view by looking for any attribute
    ' sets named "GetDim" and an attribute named "View" with a value of "Front".
    Dim dimsToRetrieve As ObjectCollection
    Set dimsToRetrieve = viewDoc.AttributeManager.FindObjects("GetDim", "View", _
                                                              ViewName)

    ' If dimensions were found with the correct attribute, retrieve them.
    If dimsToRetrieve.Count > 0 Then
        Call sh.DrawingDimensions.GeneralDimensions.Retrieve(drawView, dimsToRetrieve)
    End If
End Sub
```

Retrieving Dimensions from the Drawing

Sometimes you know the exact locations that you want to dimension to and don’t want to bother with the geometry. The problem is that Inventor requires dimensions to be associated to geometry and can’t just dimension to a point in space. A trick to working around this is create a sketch on a drawing view, add points to the sketch that are in the locations you want to dimension to, add dimension constraints between the points that represents the dimensions you want, and then retrieve the dimensions onto the sheet.

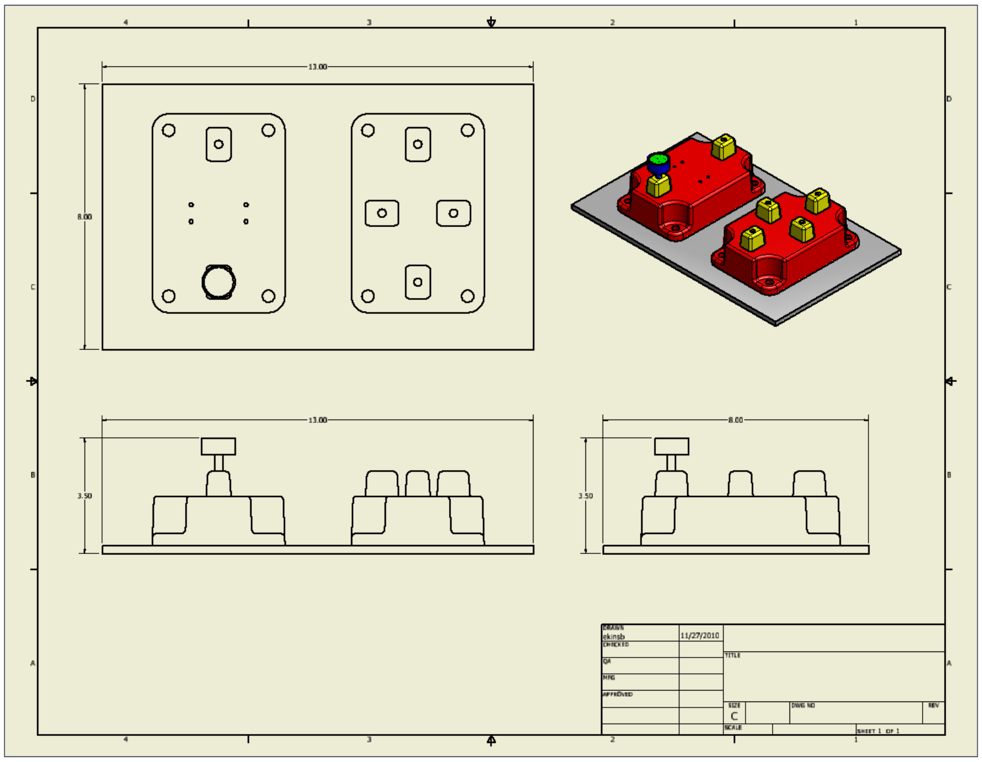
The result is not associative to the model at all. If the model was to change, these dimensions would stay in the original position since they're only dependent on the sketch points and not the model.

The sample code that demonstrates the overall dimension example below uses this technique.

Overall Dimensions

I've run into an issue a few times where I needed the overall dimensions for a part or assembly. This is not uncommon for a sales automation configurator. One issue that complicates this is that sometimes you need to place a dimension that you couldn't even create interactively because there's not a key point on the geometry at the location you need to dimension to. The TestOverallDimensions macro in the sample VBA project demonstrates an approach to this problem.

The sample examines the part or assembly geometry and finds the minimum and maximum coordinates. It then uses the trick described above to create a sketch, place sketch points at those locations, place dimension constraints between those points, and retrieve those dimensions into the drawing, as illustrated below.



Dimensioning to Named Work points

This is a fairly recent discovery for me but I think one that can solve many of the problems with automatically creating a drawing. With this approach, you create work points in the model at every location that a dimension will attach to. You'll need to figure out a naming scheme for the work points so you can easily find the correct ones when placing dimensions.

The process in this case is to find the work point of interest in the model and then retrieve that work point into the drawing as a center mark. You can turn off the visibility of the center mark to keep the drawing clean, but even when it is invisible the center mark still supports dimensions.

This approach has a lot of advantages. First, work points are always available to be retrieved into the drawing and aren't impacted by the view cleanup that Inventor performs. Second, work points can be identified by name. You can take advantage of this by using that name to find specific work points to dimension to. Third, these dimensions are associative to the model; editing the model will cause the work points to adjust, the center marks will update, and the dimensions will adjust.

The function below creates a dimension between two work points. The work points must be in the context of the part or assembly that is referenced by the drawing view. This means if the work points are in a part but the view references an assembly, you'll need to create proxies for the work points

```
Public Function DimToWorkPoints(ByVal DrawView As DrawingView, _
                ByVal WorkPoint1 As WorkPoint, _
                ByVal WorkPoint2 As WorkPoint, _
                ByVal TextPosition As Point2d, _
                ByVal AlignmentType As DimensionAlignmentTypeEnum) As GeneralDimension
    Try
        Dim sheet As Sheet
        sheet = DrawView.Parent

        ' Create centermarks based on the work points.
        Dim marks(1) As Centermark
        marks(0) = sheet.Centermarks.AddByWorkFeature(WorkPoint1, DrawView)
        marks(0).Visible = False
        marks(1) = sheet.Centermarks.AddByWorkFeature(WorkPoint2, DrawView)
        marks(1).Visible = False

        ' Create geometry intents for the center marks.
        Dim intent1 As GeometryIntent
        intent1 = sheet.CreateGeometryIntent(marks(0))
        Dim intent2 As GeometryIntent
        intent2 = sheet.CreateGeometryIntent(marks(1))

        ' Add a dimension.
        Dim genDim As GeneralDimension
        genDim = sheet.DrawingDimensions.GeneralDimensions.AddLinear(TextPosition, _
                                                     intent1, intent2, AlignmentType)
        Return genDim
    Catch ex As Exception
        Return Nothing
    End Try
End Function
```