Inventor API User's Manual

Getting Started with Inventor's API

What is an API?

For those of you who are new to customizing applications by writing programs, the first question might be: what is an API? API, or Application Programming Interface, is a term used to describe the functionality exposed by an application that allows it to be used through a program. For example, you can use Inventor's API to write a program that will perform the same types of operations you can perform when using Inventor interactively.

Having an API is important because it allows you to add functionality to Inventor that is specific to your needs. Inventor, by necessity, is a general CAD system, meaning that it's not aimed at any specific industry or used to model only certain types of products. By providing an API, Inventor allows you to add additional functionality and optimize repetitive operations to make it more productive for your individual needs.

By providing an API, Inventor also provides you the ability to better integrate Inventor into your overall Enterprise process. For example, you might write a program that interfaces with your company's inventory database to obtain the current price for components so that the price shown in a part list is always up-to-date. You might also write a program that extracts data from assemblies to provide MRP systems with their required information. All of this can be done manually, but by automating it using a program you're able to significantly increase productivity and minimize errors.

An API is also important for the reason that it allows third-party applications to integrate with Inventor. It's through the API that products for PDM, NC, and FEM are able to interact with Inventor.

Please Note: Developers are strongly cautioned against using any Inventor API object, method, property, event or constant that is marked as hidden, unsupported or for internal use only. These objects have been created or maintained for Autodesk's internal use, and there is no guarantee that they will continue to exist (or to function similarly) from release to release.

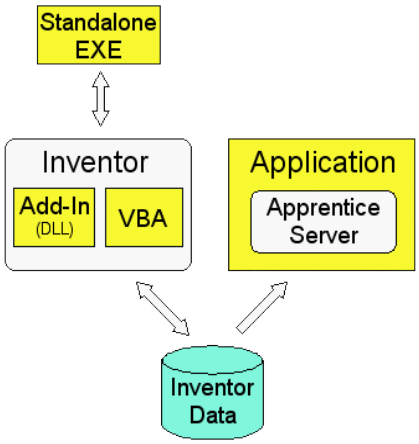
How Do I Access the API?

Inventor's API is exposed using technology from Microsoft called "Automation." (This was previously referred to by Microsoft as "OLE Automation.") Automation interfaces are common among applications written for Windows. For example, Microsoft Word and Excel expose Automation interfaces to allow you to customize them. AutoCAD and Mechanical Desktop also expose Automation interfaces. (They refer to them as ActiveX interfaces.)

There are some significant advantages to providing an API through an Automation interface. First, it allows you to use almost any of the currently popular programming languages: Visual Basic, Visual C++, C#, Delphi, Python, Java, etc. Second, if designed correctly, it exposes the functionality using standard concepts. This means that if you already have experience programming a well-designed Automation interface like Word or Excel, you already understand many of the concepts that are used by the Inventor API. Finally, it exposes the application's functionality in an object-oriented manner. Once the general concepts of how an object-oriented API works are understood, this type of API is easier to learn and use than function-oriented APIs.

Inventor exposes functionality through its API, and there are different ways to access the API. All of them are useful in certain cases, so it's important to have a basic understanding of the ways to connect to the API so you can make the best decision about how to write your program. The diagram below illustrates the different ways of accessing Inventor's API. A brief overview of each of the methods follows. Detailed information about these is provided in sections devoted to each method: VBA, Add-Ins, and Apprentice Server.

First, a quick explanation of the figure below. The white boxes represent components that provide access to Inventor's API. These are Inventor and *Apprentice Server*. We'll talk about Apprentice more in a minute. The blue cylinder at the bottom represents the Inventor data you're accessing, i.e. your .ipt and .iam files. All of the yellow boxes represent programs that you write. When one box encloses another box this indicates that the enclosed box is running in the same process as the box enclosing it. For example, VBA runs in the same process as Inventor. An "in-process" program will run faster than a program running out of process.



VBA

VBA, or Visual Basic for Applications, is a programming environment that is accessed from within Inventor. Programs written using VBA are frequently referred to as "macros." VBA is typically used by end-users to write small programs to automate repetitive tasks, although it is certainly not limited to this. A VBA program has the same access to all of the features of the API as any of the other methods of accessing the API (with the exception of Add-Ins, which we'll discuss later).

When deciding which method to use when programming Inventor, there are a few advantages to consider when using VBA. First, VBA is delivered with Inventor and does not require you to purchase an additional programming language. Second, you are able to embed programs within Inventor documents. If you have a program that is data-specific, this is a convenient way to keep the program with the data it is designed to use. (You're not forced to save your programs in Inventor documents. You can also save programs in separate files so they can be shared among users and documents.) Third, VBA runs in the same process as Inventor so you gain the performance advantages of being in the same process. Fourth, VBA was designed to work with Automation types of API's and is the most user-friendly environment for learning Inventor's API.

Add-Ins

Add-Ins are a special type of Inventor program. An Add-In is able to do one thing that none of the other methods of accessing the API is able to do; Inventor starts the Add-In automatically whenever Inventor is run. This has a huge advantage in that the add-in is able to insert itself into Inventor's user-interface and connect to events to be able to watch for and respond to activity within Inventor. This allows add-in functionality to appear the same as built-in Inventor functionality. Add-ins also have a distinct advantage over VBA in delivering your program to users and managing your source code.

Add-Ins can be written using any language that supports the creation of ActiveX DLLs, such as Visual Basic, C#, and Visual C++. Add-Ins cannot be created with VBA.

Standalone EXE

A standalone EXE is a program that runs on its own and connects to Inventor. This type of program is typically used in the case where you have a program that has its own interface and doesn't require the user to interactively work with Inventor. For example a batch plotting utility can be an EXE that runs independently of Inventor. It might monitor a database watching for new records to be added which describe documents that need to be plotted. When a new record is created in the database, the standalone EXE starts Inventor, if it's not already running, opens the desired document and plots it, all without any user interaction.

Standalone EXEs run out-of-process to Inventor, so there is some performance penalty, but since they are not usually used for interactive processes it's rarely an issue. Even in the case where performance is an issue it's possible to combine the use of an add-in and an exe. You can have an add-in that does the majority of the work and an exe that calls the add-in.

You also run in a standalone EXE mode when you write programs within another application's VBA. For example, if you write a program using Excel's VBA that connects to Inventor, your VBA program is running in the process space of Excel and is communicating with Inventor out of process.

Apprentice Server

Apprentice is an ActiveX server that can be used by other applications to get access to Inventor data. Apprentice is essentially a subset of Inventor that runs in-process to the application using it. Apprentice doesn't have a user interface and the only way to interact with it is through its API. Apprentice provides access to a limited set of full Inventor functionality with the primary areas being assembly structure, B-Rep, geometry, and iProperties. Most access to information through Apprentice is read-only; (a couple of exceptions to this are iProperties and file references).

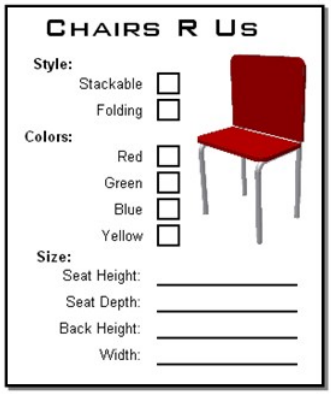
Apprentice is useful in any standalone application that needs access to information contained within Inventor documents. The alternative is to use Inventor. Using Apprentice is much more efficient than using Inventor to perform the same operations because Apprentice is able to run in the same process as your application and because it doesn't have a user interface it can perform many operations faster.

Another advantage of Apprentice Server is that it's available at no cost and is available on the public Autodesk website as a standalone application. More detailed information about Apprentice can be found in the Apprentice Server section.

Automation Programming Basics

Object Oriented Programming Concepts

A COM Automation interface exposes it functions through a set of objects. A programming object has many similarities to real-world objects. Let's look at a simple example to understand how object oriented programming terminology can be used to describe it. A company that sells chairs might allow a customer to design their own chair by filling out an order form, like the one shown below.



The options on the order form define the various properties of the desired chair. By setting the properties to the desired values the customer is able to describe the specific chair they want.

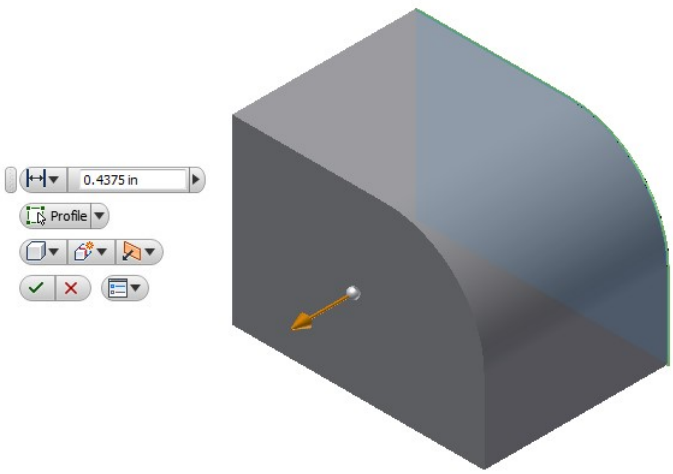
In addition to properties, objects also support methods. Methods are basically instructions that the object understands. In the real world, these are actions you would perform with the chair. For example, you could move the chair, cause it to fold up, or throw it in the trash. In the programming world the objects are smarter and instead of you performing the action you tell the object to perform the action itself; move, fold, and delete.

A third aspect of objects is that they can support events. In the real-world events are equivalent to installing sensors on an object to track when certain things happen to the object. For example, you can attach a sensor to the seat of the chair to be notified whenever anyone sits on it. In the programming world you can use events to be notified when certain things happen within Inventor. One final concept of object-oriented programming is that of a class. Going back to the chair object, you can think of the class as the order form the customer fills out to describe the specific chair they want. A class isn't the object itself but the template that defines all the properties, methods, and events of a particular type of object. The order form represents the class and the resulting chair is the object, or an instance of the class.

One final concept of object-oriented programming is that of a class. Going back to the chair object, you can think of the class as the order form the customer fills out to describe the specific chair they want. A class isn't the object itself but the template that defines all the properties, methods, and events of a particular type of object. The order form represents the class and the resulting chair is the object, or an instance of the class.

Objects and Inventor

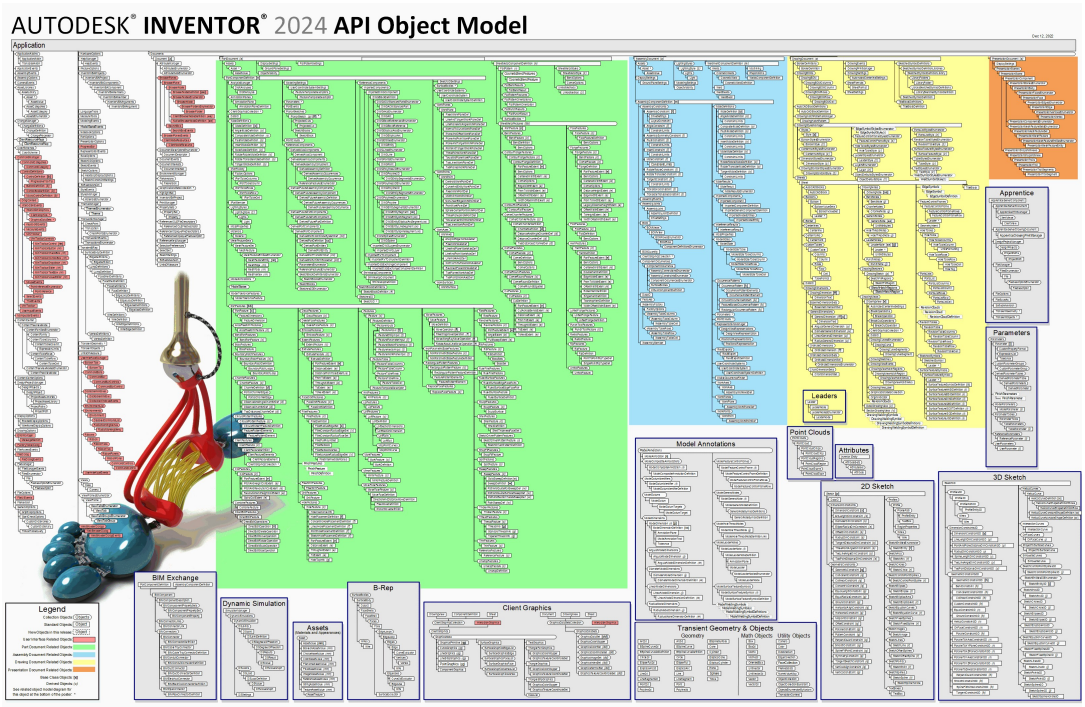
The objects exposed by Inventor's programming interface represent the things in Inventor that you're already familiar with. For example, look at the extrude feature shown below. For each extrude feature that you create in a part, there is an *ExtrudeFeature* programming object that represents it. The ExtrudeFeature object supports various properties and methods that allow you to query and edit the extrusion. These properties and methods provide equivalent functionality to what you specify when creating and editing an extrude feature the user interface. For example, the ExtrudeFeature object supports the Name property. This is the name of the feature that is displayed in the object browser. You can get the value of this property to see the current name of the feature and you can set the value of this property to change the name of the feature. The ExtrudeFeature object also supports the properties ExtentType, Operation, Profile, and others. The ExtentType property specifies that this is a distance extent. The Operation specifies that this is a "new solid" operation. The Profile returns the sketch information that defines the shape of the feature. These are the API equivalent of the options provided in the mini dialog shown below.



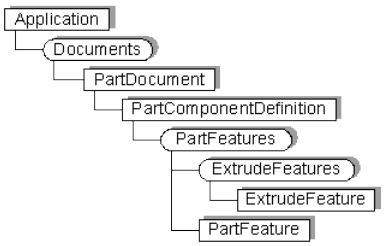
An important point to emphasize here is that most of the API is just another way besides the user-interface, of accessing Inventor's functionality. Because the API allows you to do the same things you can do through the user-interface there are strong similarities between the two. Having a good understanding of Inventor from the standpoint of an end-user will make the API easier to understand and use.

Accessing Objects

As discussed above, Inventor's API is exposed as a set of objects. By gaining access to these objects through the API you can use their various methods, properties, and events to control and react to Inventor. Let's look at how you access these objects. The first concept to understand before going further is the object model. The full Inventor object model is shown below. You can see that there are a lot of objects, but most programs will use only a small portion of them.



The object model is a hierarchical diagram that illustrates the relationships between objects. One of the most important objects in this hierarchy is the *Application* object. The Application object represents the Inventor application and is the top-most object in the hierarchy. The Application object supports methods and properties that let you control Inventor but most important, it supports properties that return other Inventor objects. Because of this, once you have the Application object you can access any other object in the hiearchy. The object model picture is useful as a tool because it illustrates how to get from one object to another. You just need to understand how to traverse the hierarchy to get to the specific object you want. The diagram below illustrates the portion of the object model that is needed to access the part extrude feature, (ExtrudeFeature object).



The code below illustrates using the relationships indicated in the object model diagram above to access the extrude feature named "Extrusion1".

```
Public Sub GetExtrudeFeature()
   Dim partDoc as PartDocument
   Set partDoc = ThisApplication.ActiveDocument

   Dim extrude As ExtrudeFeature
   Set extrude = partDoc.ComponentDefinition.Features.ExtrudeFeatures.Item(1)

   MsgBox "Extrusion " & extrude.Name & " is suppressed: " & extrude.Suppressed
End Sub
```

Examining the code above, there are a lot of basic concepts used that many people struggle with when first starting to use Inventor's API. These concepts are: obtaining the Application object, traversing the object model, collection objects, and derived objects. The following looks at each of these.

Obtaining the Application Object

The first thing illustrated in the sample is accessing the Application object. In Inventor's VBA you can use the ThisApplication global property to get the Application object, which is what the sample does. When writing an add-in, when Inventor starts the add-in it passes it the Application object. When writing an exe you need to use other API's to get the Application object. In this case you can choose to either get the Application object from a running instance of Inventor or to start Inventor and get the Application object from it. Below is some VB.Net exe code that is the equivalent the VBA code above. The first portion of the code uses the .Net GetActiveObject method to get the Inventor application object. This is wrapped within a Try Catch statement to handle the case where Inventor isn't running. The remaining code is almost identical to the VBA sample.

```
Imports Inventor
Imports System.Runtime.InteropServices
------

Public Sub GetExtrudeFeature()
    ' Get the Inventor Application object.
    Dim inventorApp As Inventor.Application = Nothing
    Try
        inventorApp = Marshal.GetActiveObject("Inventor.Application")
    Catch ex As Exception
        MessageBox.Show("Cannot connect to Inventor")
        Exit Sub
    End Try

    Dim partDoc As PartDocument
    partDoc = inventorApp.ActiveDocument

    Dim extrude As ExtrudeFeature
    extrude = partDoc.ComponentDefinition.Features.ExtrudeFeatures.Item(1)

    MessageBox.Show("Extrusion " & extrude.Name & " is suppressed: " & extrude.Suppressed)
End Sub
```

Here's the same example in C#.

```
using Inventor;
using System.Runtime.InteropServices;
------

private void GetExtrudeFeature()
{
    Inventor.Application inventorApp = null;
    try
    {
       // Attempt to get a reference to a running instance of Inventor.
       inventorApp =(Inventor.Application)Marshal.GetActiveObject("Inventor.Application");
    }
    catch
    {
       MessageBox.Show("Unable to connect to Inventor.");
       return;
    }

    PartDocument partDoc = null;
    partDoc = (PartDocument)inventorApp.ActiveDocument;

    ExtrudeFeature extrude = null;
    extrude = partDoc.ComponentDefinition.Features.ExtrudeFeatures[1];

    MessageBox.Show("Extrusion " + extrude.Name + " is suppressed: " + extrude.Suppressed);
}
```

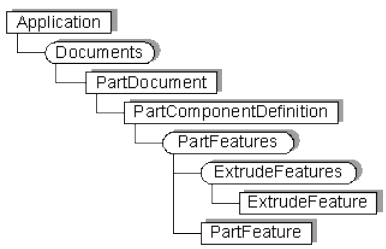
Traversing the Object Model

Once you have the Application object, the first step in getting a specific feature is to get the document the feature is contained within. In the sample above there's a slight discrepancy between the object model diagram and the sample code. Notice that the diagram shows that the Documents object is the next step from the Application object, but the sample program does not do this. The reason for this is that there are many cases in the API where short cuts are provided that make it easier to access specific objects. In this case the ActiveDocument property of the Application object is used which returns the Document object currently being worked on by the end-user, allowing us to bypass the Documents object.

After getting the PartDocument object the rest of the code illustrates traversing the object model to get down to the desired feature. The ComponentDefinition property of the PartDocument object is called which returns a PartComponentDefinition object. The Features property of the PartComponentDefinition object is called which returns a PartFeatures object. The ExtrudeFeatures property of the PartFeatures object is called which returns an ExtrudeFeatures object. Finally, the Item property of the ExtrudeFeatures object is called which returns the ExtrudeFeature object that has the specified name. Hopefully you can see that the code is using the relationships defined in the object hierarchy to traverse the object model to the desired object.

Collection Objects

Collection objects are a special type of utility object in the API. A collection object doesn't represent a particular entity within Inventor but instead provides access to a group of related objects. A collection object also supports the methods that allow you to create new objects. The portion of the object model to access the extrude feature is shown below. The rectangular boxes represent standard objects whereas the boxes with rounded corners represent collection objects. Collection objects can also be identified by their plural names. In this example the collection objects are the Documents, PartFeatures, and ExtrudeFeatures objects.



The primary difference between collection objects and other objects is the concept that they provide access to a set of objects. They do this by supporting the Count and Item properties. All collection objects support these two properties. The Count property returns the number of objects currently in the collection. For example, if you call the Count property of the ExtrudeFeatures collection object it will return the number of extrude features in that document. The Item property returns a specific object within the collection. Typically, when using the Item property you specify the index of the item you want from the collection. For example, the code below prints out the names of all of the ExtrudeFeature objects in the document by going through the contents of the collection one by one by using the Count and Item properties of the ExtrudeFeatures collection object.

```
Public Sub ShowExtrudeFeature()
   ' Get the active document.  This assumes it is a part document.
   Dim partDoc as PartDocument
   Set partDoc = ThisApplication.ActiveDocument

   ' Get the ExtrudeFeatures collection object.
   Dim extrudeFeatures As ExtrudeFeatures
   Set extrudeFeatures = partDoc.ComponentDefinition.Features.ExtrudeFeatures

   ' Iterate through the contents of the ExtrudeFeatures collection.
   Dim i As Integer
   For i = 1 to extrudeFeatures.Count
      ' Get a specific item from the ExtrudeFeatures collection.
      Dim extrude As ExtrudeFeature
      Set extrude = extrudeFeatures.Item(i)

      Debug.Print extrude.Name
   Next
End Sub
```

When the Item property is used with a value indicating the index of the item, the first item in the collection is 1 and the last item is the value returned by the collection's Count property. For some collections the Item property also supports specifying the name of the item you want. Instead of providing the index of the item you can supply a String that specifies the name. The code below demonstrates how to get the extrude feature named "My Extrude". The call will fail if there's not an extrusion with that name. The online help and the object browser (both are discussed below) can be used to determine if an Item supports indexing by name. All Item properties support indexing by value.

```
Dim extrude As ExtrudeFeature
Set extrude = extrudeFeatures.Item("My Extrude")
```

When iterating through the objects contained within a collection you can also use a For Each statement. The following does the same thing as the earlier sample but is more concise and will typically be faster.

```
' Iterate through the contents of the ExtrudeFeatures collection.
Dim extrude As ExtrudeFeature
For Each extrude In extrudeFeatures
   Debug.Print extrude.Name
Next
```

In addition to providing access to their contents through the Count and Item properties, many collections also support methods to allow you to create new objects within that collection. For example, the ExtrudeFeatures collection supports the Add method that lets you create new extrusions.

Derived Objects

The idea of *derived* and *base class* objects is usually a new concept for most end-users wanting to use Inventor's API. To help describe this concept let's look at a close parallel that most people will be familiar with; animal taxonomy or classification. For example within the Animal kingdom you have insects, birds, mammals, etc. Within the mammal classification there are many different species but all of them share the same characteristics of a mammal; have hair, produce milk, etc. This same idea can be used to understand the concept of derived and base class objects.

If we were to write a program that represented the animal classification discussed above we would have the Animal, Insect, Bird, and Mammal objects. The Animal object is a base class for the Insect, Bird, and Mammal objects. The base class object is an object that supports methods and properties that are common to any of the objects derived from it. The Animal object can represent any animal regardless of whether it is actually an insect, a bird, or a mammal. The Animal object also supports methods and properties that are applicable to any animal, i.e. weight, name, etc.

The Insect, Bird, and Mammal objects are derived from the Animal object, so they inherit all of the functionality of the Animal object. In addition they support methods and properties that are unique to that particular group. If we continued the classification further we might add the Human and Dog objects. The Mammal object will be the base class for these objects and they inherit all of the functionality of the Mammal object. This concept is useful in programs where you are working with a mixed set of objects but those object share some functionality. For example, if you're going to iterate over all of the animals in a zoo and get their names you can use the Animal object because it can represent any animal and supports the Name property.

Let's look at how this concept applies to Inventor. Inventor has base class objects and derived objects. An example is the Document, PartDocument, AssemblyDocument, and DrawingDocument objects. The base class object is the Document object. This object can represent any type of document. The specific document type objects are derived from the Document object. They support everything the Document object supports plus they support additional methods and properties that are specific to that document type. For example, from the Document object you can get the filename, referenced documents, and iProperty information. From a PartDocument object you can get all of that, plus you can get sketches, features, and parameters. Another example is the PartFeature object. This is the base class object for all features. Again, it can represent any part feature and supports the functions that are common to all features. The code below illustrates this concept in use. It checks every feature in the active part and unsuppresses any suppressed feature.

```
Public Sub SuppressOff()
    ' Get the active part document.  Assumes a part is active.
    Dim partDoc As PartDocument
    Set partDoc = ThisApplication.ActiveDocument

    ' Iterate through all of the features.
    Dim feature As PartFeature
    For Each feature In partDoc.ComponentDefinition.Features
        ' Check to see if the feature is suppressed.
        If feature.Suppressed Then
            ' Unsuppress the feature.
            feature.Suppressed = False
        End If
    Next
End Sub
```

In the sample above the variable feature is declared as PartFeature type. This is the base class object for all features. It then iterates through all of the features in the

part. All features can be suppressed so the Suppressed property is supported by the base class PartFeature. This sample checks the suppression state of each feature and unsuppresses any suppressed features by setting the Suppressed property to False.

If the PartFeature base class did not exist then to accomplish the same thing you would have to write a similar loop for every feature type. As new features are added in releases you would have to update your code to take these into account. Having a base class that represents any feature type is allows your program to be much simpler.

If you need specific information about a particular type of feature then you use the specific feature type. For example if you need to get the depth of an extrusion you use the ExtrudedFeature object, which provides the additional information that is specific to extruded features.

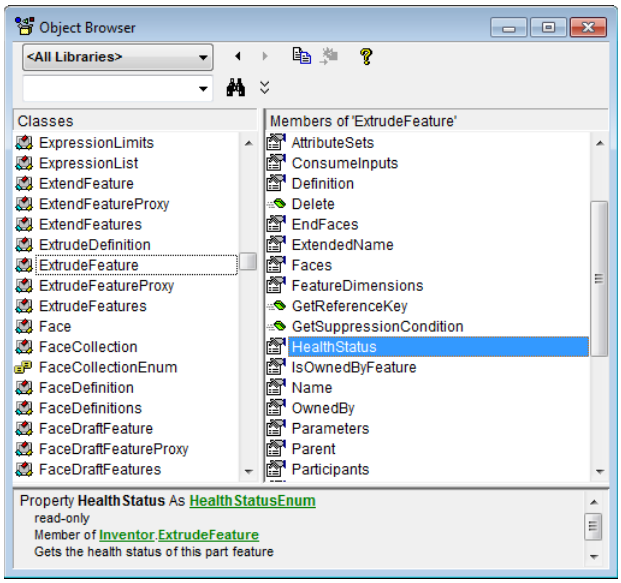
Unfortunately, a listing of what are the base and derived classes is an area of the API that is not documented very well. Ideally another chart, in addition to the object hierarchy chart, would exist that shows this class categorization. The best documentation currently available that describes these relationships is the object model diagram.

Programming Tools

There are tools available to help you when working with Inventor's API. The following describes each of these.

Object Browser

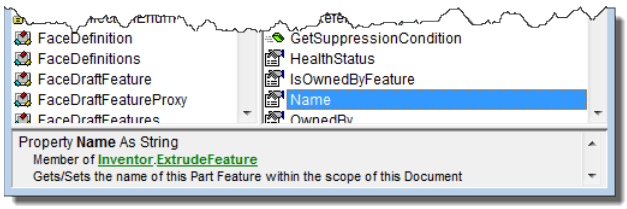
This is a tool you'll want to take advantage of when working with Inventor's API. You access it from the VBA programming environment by pressing the F2 key, the toolbar button  , or selecting Object Browser from the View menu. The Object Browser is shown below.



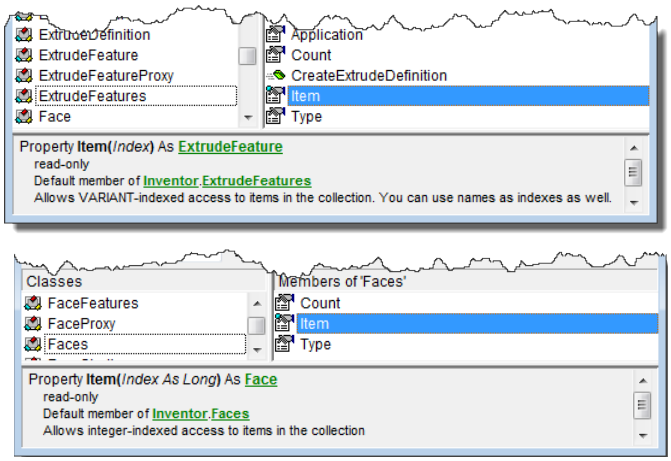
The left column of the browser, titled "Classes", contains a list of all the objects available in the libraries currently attached. In the drop-down list at the top-left of the browser, "" in the example above, you can specify a specific library to limit the list to only the objects in that library. In this example, the ExtrudeFeature class has been selected.

The right column displays a list of the methods, properties, and events supported by the selected class. Selecting one of these will display information about that function at the bottom of the object browser. In the example above the HealthStatus property has been selected. At the bottom of the dialog we learn that this property returns a HealthStatusEnum value. (Enum values are lists of values that are also defined in Inventor's type library.) We also learn that this property is read-only, meaning that we can get the value but we can't change it. A brief description of the selected function is also displayed at the bottom of the browser. The online help is useful in conjunction with the browser because it is context sensitive. Pressing F1 at this point will bring up the help page for the HealthStatus property.

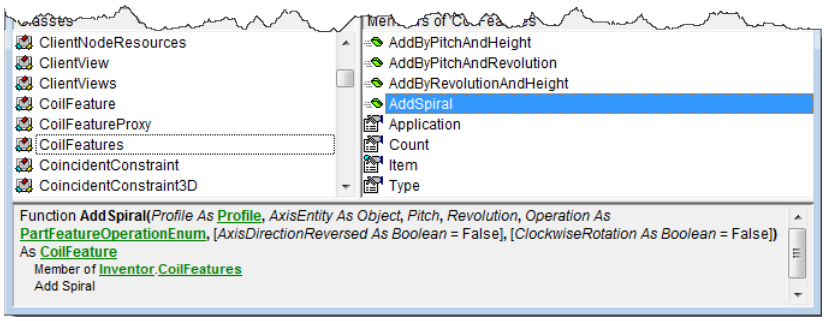
Here are more examples of the information that's displayed at the bottom of the object browser and how to interpret this information. In the example below, the Name property of the ExtrudeFeature object is selected. The property returns a String. Notice that unlike the previous example, it does not say that it is read-only so this property is read-write. Using this property you can get the name of the feature and you can also assign a String to this property to set the name of the feature.



The Item property of a collection object was discussed earlier. Remember that you can always use a value as the index to get a specific item and sometimes you can also use the name of the item. The two examples below show one technique to determine if using a name is supported. The first example is for the ExtrudeFeatures collection. It shows that the Item property has one argument called index. Notice that it doesn't say what type this argument is. Whenever the type is not specified that means it is a Variant. A Variant is a special type that can represent any variable type. In this case index is a Variant to allow you to provide either a Long or a String. Often, as in this case, this is also indicated in the description and in the online help. In the second example, index is specified to be a Long value. In this case you can only supply a value. Supplying a String will cause a failure.



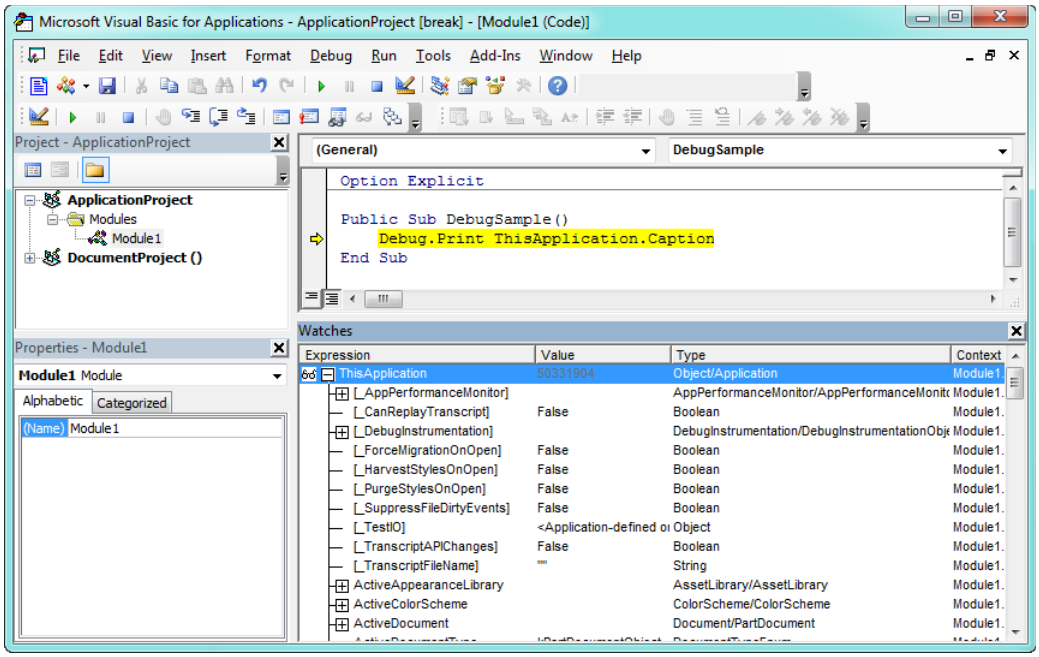
Here's a more complicated signature and more about what can be learned by looking at the signature in the object browser. We'll use the AddSpiral method of the CoilFeatures collection object. This method has several arguments. The first is called Profile and requires a Profile object as input. The second called AxisEntity doesn't have a specific type but will take any object. What this means is that this particular method will take more than one type of object for this argument so a specific type can't be specified. In this example it can be a sketch line or work axis. The Pitch argument does not have a type displayed because it is a Variant. In this case you can use either a Double specifying the pitch in centimeters or you can use a String and specify an equation which can include parameter values. The same is true for the Revolution argument; it can be a Double or a String. The Operation argument expect a value from an enum list. The last two arguments are optional. In this case they both happen to be Boolean arguments and both default to False if not provided. For more details about all functions and their arguments you can use the API help. You can easily access the help for a particular object or function by pressing F1 when using the object browser.



Using the VBA Debugger

Another very useful tool when working with Inventor's API is the VBA debugger. In this case we're not using it to debug a program but using some features of the debugger to better understand Inventor's object model. To use the debugger we need to be running a program. Fortunately we don't need much of a program to enable access to the Inventor objects. The specific steps to use the debugger are listed below.

1. Within any code module, create a sub, like the sub Test shown below.
2. Click the mouse anywhere within the sub and press F8 to step into the code. You're now running the sub one line at a time.
3. Click anywhere within the word "ThisApplication".
4. Run the Quick Watch command from the Debug menu and click "Add" in the Quick Watch dialog.
5. The debug window will appear and ThisApplication will appear within the debug window. This represents a live view of the Inventor Application object. You can click on the "+" sign next to ThisApplication to expand it and view its properties and their values. Properties that return other objects will also have the "+" sign next to them allowing you to continue to expand the objects and view their properties. This provides a live view of the object model.



VBA in Autodesk Inventor

Microsoft's Visual Basic for Applications (VBA) is integrated into Inventor. VBA is a powerful development tool for customizing Inventor and integrating Inventor with other applications and data. Visual Basic is one of the more popular development tools in the world today. There are several different versions of Visual Basic. Visual Basic 6 was the last version of the older development environment and has been replaced with VB.Net for creating add-ins and executables. VBA provides similar functionality as VB 6 but integrates the development environment into the host applciation. In this case the VBA development environment is integrated into Inventor. VBA is delivered as part of Inventor at not additional cost so anyone that has Inventor has the ability to write and use VBA macros. VBA does not create standalone applications, but always runs from within Inventor. VBA is the programming language used for Microsoft Word and Excel and many other popular applications. If you've used VBA to write programs for other applications you already know the VBA language and programming environment. To use it in Inventor you'll only need to learn the Inventor API. If you haven't used VBA in other applications, learning it within Inventor gives you a head start on customizing other VBA-enabled applications.

Autodesk Inventor's VBA

VBA is provided by Microsoft and integrated into Inventor. Although VBA is generic, it can be customized somewhat to meet the needs of the specific application it is integrated with. This section will discuss the things that are unique to the Inventor integration of VBA. Information about the general functionality of VBA can be found in the VBA help provided by Microsoft.

VBA allows you to create forms, class modules, and code modules. These modules are contained within projects. In most development environments, a project contains all of the source code that is used to create a single standalone component. Because VBA doesn't create standalone components, projects are used a bit differently. A VBA project is just a container for the various modules. Because of this, a single VBA project can contain a lot of unrelated functionality. Any number of functions can be written within a single project and executed independently.

Inventor's VBA supports three types of projects: document, application, and user. The primary difference between these types of projects is the location in which the VBA project is stored. Document projects are stored within Inventor documents. Application and user projects are stored in external files. Because Inventor supports saving VBA projects within Inventor documents or in external files, you can decide which approach is better for your particular situation. Below are some of the factors to consider when making this decision.

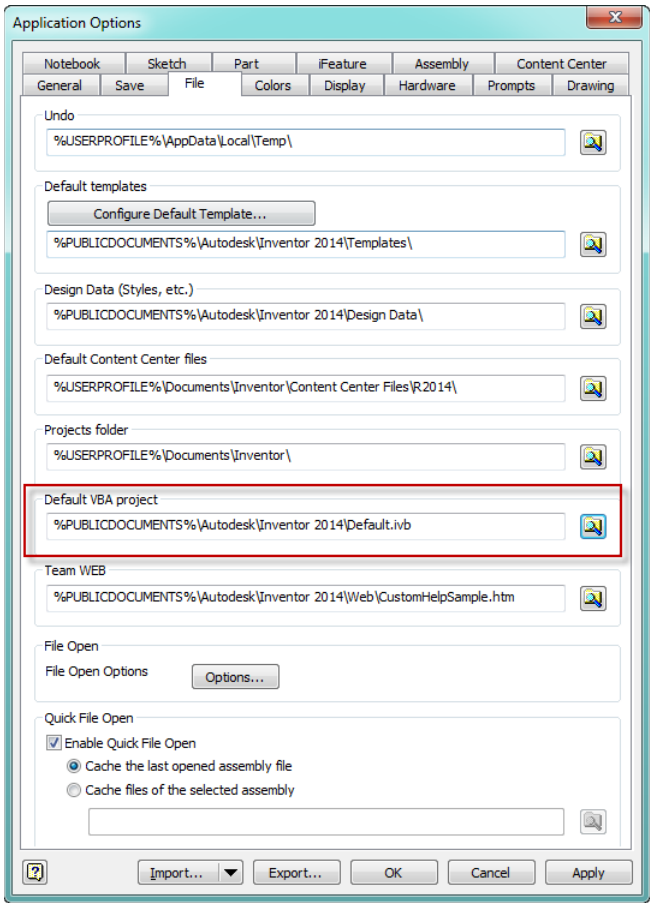
Document Projects (Saving in the Document)

Document projects allow you to easily deliver code that is specific to a document with that document. For example, if you have a standard part that is used to create members of a family of parts, you can write a program that sets the parameters of the part to create the various family members. It's convenient to embed this program in the Inventor document so that the program will always be available along with the part document.

Application and User Projects (Saving in an external .ivb file)

- Allows programs to be more easily shared between documents. Frequently programs are written to automate repetitive tasks. The usefulness of these programs is not isolated to a single document. With an application or user project, these programs become available to all documents.
- Allows code to be easily shared with other users. One person can write a program, which might consist of several functions and forms, and then send the project to anyone else, who can load the project and access the functionality. No additional work is required. By contrast, if you want to share a document project with other users, you need to send them the Inventor document that contains the embedded program and they must copy and paste the code into the desired Inventor document. Or you can export each of the modules as .frm, .bas, and .cls files and send them. These can then be imported into the VBA environment of the Inventor document where they're needed. As you can see, application and user projects provide a much simpler mechanism for code sharing and reuse.
- Allows easier management of source code. All of the code is contained within a single file, which makes updates much easier. In the case of document projects, every document that needs the functionality has the program embedded within it. As the program is updated and bugs are fixed it can be very difficult to keep track of which documents contain the program and which version of the program they contain.

So far, all of the behavior we have discussed makes it appear that application and user projects are the same. This is largely true, but the key difference between them is how they get loaded into the VBA environment. Inventor loads the application project automatically whenever the user starts Inventor. This makes macros within an application project always available. Only one project can be defined to be the application project. The project file that serves as the application project is defined using Inventor's Application Options dialog, as shown below. The "Default VBA Project" field on the File tab defines which user project to use as the application project.



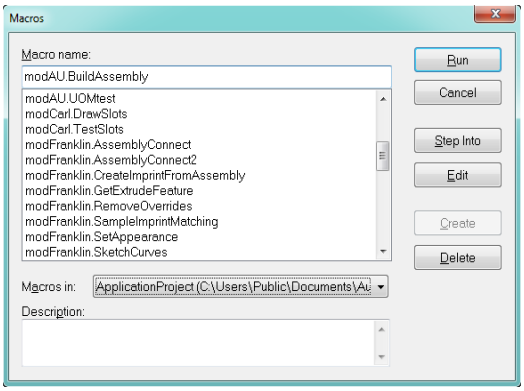
User projects are not automatically loaded but must be loaded manually using the "Load Project" command in the Inventor VBA File menu (In Inventor, choose the Tools tab, then VBA Editor, then File | Load Project...). New user projects can also be created using the "New Project" command in the VBA Files menu. There is no limit to the number of user projects that can be loaded.

Inventor's VBA interface is accessed within Inventor by through the **VBA Editor** command in the Tools tab of the ribbon. The **Macros** command lets you run existing macros and open the editor to a specific macro. The two commands are also available using the Alt+F8 and Alt+F11 keyboard shortcuts.

Before going any further let's define some terms. A *Sub* is essentially a function without a return values. The code is enclosed within the Sub and End Sub statements.. A Sub can have arguments that can be input and output but it doesn't have a return value. A *macro* is a special case of a Sub that doesn't have any arguments. Any Subs without arguments will be considered macros. For example, the following code will be treated by VBA as a macro.

```
Public Sub SampleMacro()
    MsgBox "This is a sample."
End Sub
```

Running the **Macros"** command in Inventor displays the dialog shown below. This dialog displays a list of all of the macros currently available. This list can be filtered using the "Macros in" combo box. This allows you to list the macros contained in a particular project (document, user or application project), or to list all of the macros in all of the loaded projects.



When a macro is selected from the list, all of the buttons on the right, except one, are enabled. The following describes the behavior of each of the buttons.

Run - The selected macro is executed without displaying the VBA programming environment.

Cancel - Dismisses the dialog.

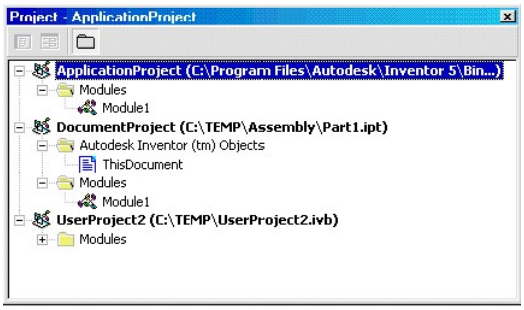
Step Into - Opens the VBA programming environment and begins running the selected macro by stepping into the function. This is useful when debugging a macro.

Edit - Opens the VBA programming environment with the cursor positioned at the top line of the selected macro.

Create - If an existing macro is selected, this button is disabled. If a new name is entered into the "Macro name" field, this button becomes enabled and when clicked will cause a macro of the specified name to be created.

Delete - Deletes the selected macro. This will remove the sub from the project.

Running the **VBA Editor** command in Inventor opens the VBA programming environment. This is commonly known as the "Integrated Development Environment" or IDE. Within the VBA IDE is the "Project Explorer" which provides a browser-like view of the currently open projects, as shown below.



Several important concepts of Inventor's VBA integration are illustrated in the Project Explorer. The first of these is the "Application" project. As discussed earlier, this is a project file that is stored in an external file and is loaded automatically whenever Inventor is run. The name of the project file is shown in parentheses next to the project's name.

The second project in the list is a document project. The name of the document in which the project is embedded is shown in parentheses next to the project's name. Document projects are shown for the documents currently open in Inventor. One difference between user and document projects that's apparent when looking at the Project Explorer is that the document project exposes the "ThisDocument" object. This represents the document that contains the macro. Writing code within the ThisDocument module give you direct access to the Inventor document. For example, the following code is valid within the ThisDocument module.

```
Public Sub DocDisplayName()
    Dim sDocDisplayName As String
    sDocDisplayName = DisplayName
End Sub
```

DisplayName is a property of the Document object. When writing code within the ThisDocument module, the methods and properties of the document are directly available. The document is also available within other modules of the project by using the ThisDocument global variable. For example, the code below can be used in other modules of a Document project.

```
Public Sub DocDisplayName ()
    Dim sDocDisplayName As String
    sDocDisplayName = ThisDocument.DisplayName
End Sub
```

The ThisDocument global variable is not available in application and user projects. In these projects, and in document projects, you can use the "ThisApplication" global variable. This provides direct access to the Inventor Application object. This sample can be used in both user and application projects to get the display name of the currently active document.

```
Public Sub DocDisplayName ()
    Dim sDocDisplayName As String
    sDocDisplayName = ThisApplication.ActiveDocument.DisplayName
End Sub
```

What is an Add-In?

Inventor's Add-In functionality is a way for a program to connect to Inventor and use it's API. The other common ways of accessing Inventor's API are from Inventor's VBA and from an external exe. All of these are valid ways of using Inventor's API and each has their own advantages and disadvantages. Which one to use will depend on what your program needs to do and how it will be used by the end-user. Add-ins are typically used in the case where you want to add new functionality to Inventor in the form of custom commands. Add-in's have the following capabilities that make them ideal for this.

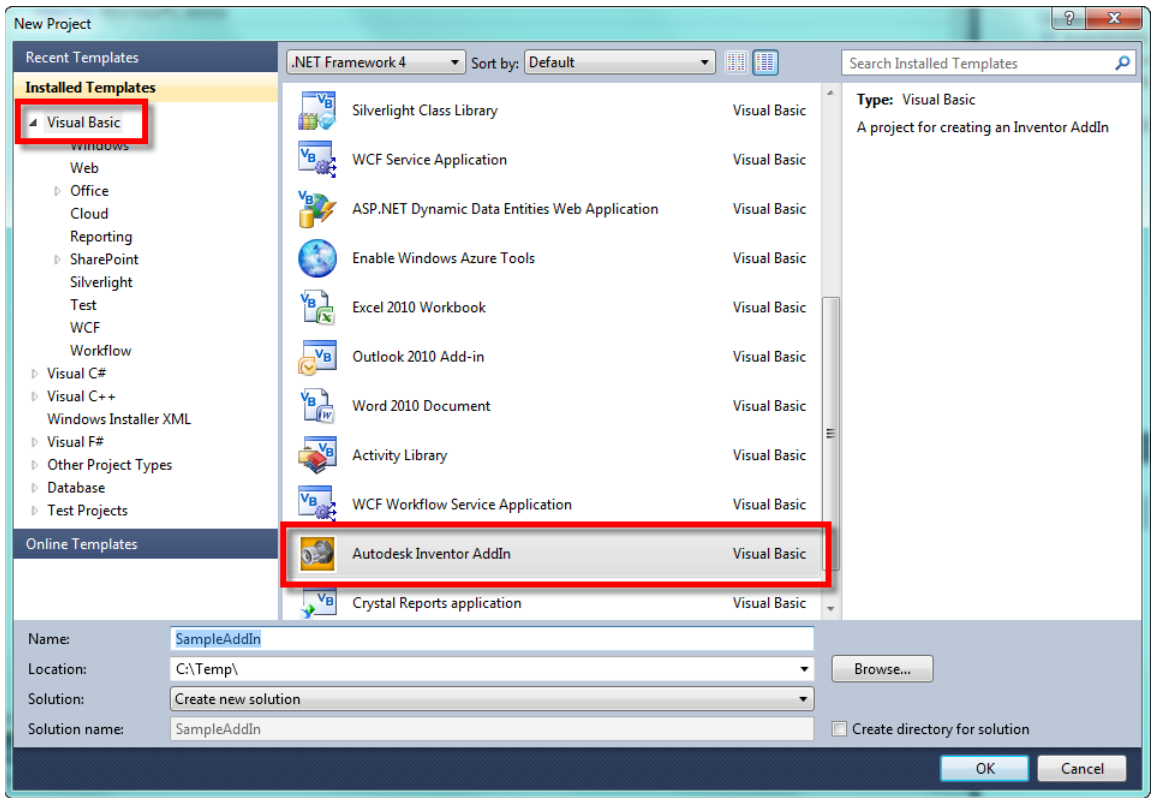
- Add-ins are loaded automatically when Inventor is started. This simple action provides some powerful capabilities. First, once an add-in has been installed on a computer, the user doesn't need to do anything special or know anything about programming to be able to access the capabilities it provides. Second, it allows the add-in to add its user-interface into Inventor's, i.e. ribbon buttons, browser tabs, etc. Third, the add-in can connect to events to monitor and respond to user activity in Inventor.
- Add-ins run in-process to Inventor, providing much better performance than an external exe.
- Add-ins can be written in any of the more popular languages so you're not limited to using a language you're not familiar with or that may be outdated but can take advantage of the latest programming technologies available.

Creating an Add-In

To create an add-in you'll need to use Visual Studio. Any version is supported. It's possible to create an add-in using any language that supports creating a dll COM component, but all of the samples and tools provided with the SDK are limited to Visual Basic, C#, and C++. It's possible to create add-ins using the free Express editions but with some severe limitations. For example, debugging is not possible in Visual Basic Express 2010. Because of this the professional version is recommended. The Express editions are adequate for developing standalone executable and are an excellent choice for getting started with Inventor's API. But for add-in development you should plan on Visual Studio Professional. The choice to use one of the other versions of Visual Studio with more features than the Professional version would be solely based on your needs and is not a requirement of Inventor.

for the next step, use the **Inventor Add-In** template in Visual Studio to create your add-in. In order to use the add-in template you'll need to install the SDK and then install the wizards. The location and installation instructions for the SDK is described in the Introduction to Using Inventor's Programming Interface overview.

With the add-in wizards installed, you can create a new project in Visual Studio selecting the **Autodesk Inventor AddIn** template as shown below. On the left of the dialog, you may need to select the top-level node in the tree for the language you're using to see all of the available templates.



The project that's created using the template is a skeleton add-in. When Inventor starts your add-in it begins by calling the Activate method, which is shown in the picture below. A critical thing happens in this call, an object is passed to your add-in through the addInSite argument of the Activate method. This object supports an Application property which your add-in can use to gain access to Inventor's Application object and the entire API. You'll notice that the template is assigning the reference returned by the Application property to a local member variable so the add-in can maintain a reference to the Application object. It's also typically in the Activate method where your add-in will add its user interface to the Inventor's user interface using Inventor's API. It will also connect to any events that it needs to monitor. Once it's been loaded and is running the add-in doesn't do anything except react to events as they occur. For example, it would respond to the click when the user clicks one of its custom commands.

```
Public Class StandardAddInServer
    Implements Inventor.ApplicationAddInServer

    ' Inventor application object.
    Private m_inventorApplication As Inventor.Application

#Region "ApplicationAddInServer Members"

    Public Sub Activate(ByVal addInSiteObject As Inventor.ApplicationAddInSite, ByVal firstTi

        ' This method is called by Inventor when it loads the AddIn.
        ' The AddInSiteObject provides access to the Inventor Application object.
        ' The FirstTime flag indicates if the AddIn is loaded for the first time.

        ' Initialize AddIn members.
        m_inventorApplication = addInSiteObject.Application

        ' TODO:  Add ApplicationAddInServer.Activate implementation.
        ' e.g. event initialization, command creation etc.

    End Sub
```

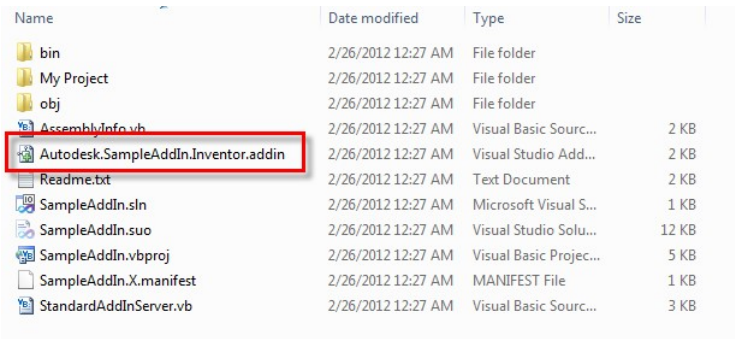
To quickly test that the use of the add-in template resulted in the creation of a working add-in you can compile the project as-is. This will compile your add-in as a dll. The next step is to make your add-in known to Inventor.

Making Your Add-In Known to Inventor

Every time Inventor starts it looks for add-ins that have been installed and loads them. In previous versions before Inventor 2012 it used the registry to find the add-ins. This registry-based look-up is still supported for legacy add-ins, but a new registry-free method of making your add-in known to Inventor is now supported and is the recommended approach to use. When an add-in is created using the add-in template a registry-free add-in is created.

In order to allow your add-in to be found by Inventor, you need to place a special file in one of a few different directories. This file describes your add-in and also indicates where your add-in dll is on the computer. When Inventor is started, it scans these directories and reads these files to determine which add-ins should be loaded.

The file that Inventor looks for has a *.addin* extension. A .addin file was automatically created for you when you created your add-in project, as is illustrated below.



The conents of the .addin file are shown below. As you can see it uses xml to format the data.

```
<Addin Type="Standard">
<!--Created for Autodesk Inventor Version 17.0-->
<ClassId>{51e6ad8e-5eaa-42a1-b845-a68802a26bf7}</ClassId>
<ClientId>{51e6ad8e-5eaa-42a1-b845-a68802a26bf7}</ClientId>
<DisplayName>SampleAddIn</DisplayName>
<Description>SampleAddIn</Description>
<Assembly>SampleAddIn.dll</Assembly>
<OSType>Win64</OSType>
<LoadAutomatically>1</LoadAutomatically>
<UserUnloadable>1</UserUnloadable>
<Hidden>0</Hidden>
<SupportedSoftwareVersionGreaterThan>16..</SupportedSoftwareVersionGreaterThan>
<DataVersion>1</DataVersion>
<LoadBehavior>2</LoadBehavior>
<UserInterfaceVersion>1</UserInterfaceVersion>
</Addin>
```

The following describes the different elements of the .addin file:

Addin - (Required) The outermost element to involve all other elements. The Type attribute specifies the addin type, valid values are "Standard" and "Translator".

ClassId - (Required) Specifies the ClassId GUID associated with an Add-in. This may or may not change from one release to the next. If you look at your add-in code you'll see this specified as a GuidAttribute for your add-in class. In almost all cases you don't need to do anything with this but use what is provided.

ClientId - (Required) Specifies a GUID that is used as the add-in identifier. This value should remain unchanged across releases and different versions of the add-in. This value is used to identify the owner of API-created objects such as ribbons, toolbars, etc. In almost all cases this will be the same as the ClassId.

DisplayName - (Required) Specifies the display name of the add-in as it will appear in the Add-in manager. The Language attribute can be specified for local languages, if not specified it defaults to English, below example sets the DisplayName for French:

<DisplayName Language="1036">Convertisseur: DWF</DisplayName>

Description - (Required) This is a description of your add-in and is displayed in the bottom of the Add-In Manager dialog when the add-in is selected from the list. The Language attribute can be specified for local languages, if not specified it defaults to English, below example sets the Description for French:

<Description Language="1036">Convertisseur Autodesk interne DWF</Description>

Assembly - This is the path to you add-ins dll. This can be a full path or a relative path where it's relative to the location of your .addin file. It can also be relative to the Inventor\bin directory however since you're not allowed to install into that directory with Administrator privileges it's not recommended that you use that directory. A path relative to the location of the .addin file is recommended and is discussed in more detail below.

OSType - (Optional) Specifies if your add-in will work with only a 64 or 32 bit operating system. Valid values for this are "Win32" or "Win64". If this value is not specified it's assumed the add-in is assumed to be valid for both.

LoadAutomatically - (Optional) Specifies whether the add-in should be allowed to load automatically as per the load behaviors defined by the add-in. Value can be 0 or 1. Assumed to be true (1) if this value is not specified. If set to false (0), the add-in needs to be manually loaded using the add-in manager.

LoadBehavior - (Optional) Specifies when the add-in should be loaded in Inventor. This is important for better startup performance. This option can be specified using one of the following values:

0 - Load immediately on startup **(not recommended)**
1 - Load when any document is opened
1 - Load when a part document is opened (same as previous)
2 - Load when an assembly document is opened
3 - Load when a presentation document is opened
4 - Load when a drawing document is opened
10 - Load only on demand, either through the API or using the Add-In Manager.

Assumed to be **0** if this value is not specified.

UserUnloadable - (Optional) Specifies whether the add-in should be allowed to load automatically as per the load behaviors defined by the add-in. Value can be 0 or 1. Assumed to be true (1) if this value is not specified. If set to false (0), the add-in needs to be manually loaded using the add-in manager.

Hidden - (Optional) Specifies whether the add-in should be hidden in the Add-in Manager's list of add-ins. Assumed to be false if this value is not specified (i.e. add-in is visible). Value can be 0 or 1.

SupportedSoftwareVersionEqualTo
SupportedSoftwareVersionGreaterThan
SupportedSoftwareVersionLessThan
SupportedSoftwareVersionNotEqualTo - (Optional) Specifies the version(s) of Inventor that the add-in should be available in. Combinations of these can be used. These values are ignored if the manifest file is located in a version-specific folder.Versions are declared in the format of Major#.Minor#.ServicePack# / or BuildIdentifier#. SupportedSoftwareVersionEqualTo and SupportedSoftwareVersionNotEqualTo support multiple version entries seperated by a semicolon (;).

DataVersion - (Optional) Specifies the version of add-in data contained within Inventor documents that this version of the add-in supports. This is used by add-ins that store migrating data in Inventor documents, which is indicated by the "DocumentInterests" set on the document.

UserInterfaceVersion - (Optional) Specifies the version of the add-in's user interface. Changing this version results in all of the add-in's UI getting cleaned up during Inventor start-up.

Below elements are specifically for translator add-ins.

FileExtensions - (Optional) Specifies the file extensions of the translator add-in can import from or export to. If multiple file extensions are specified the delimiter semicolon can be used between them. Below is sample to specify the FileExtensions:

<FileExtensions>.CATPart;*.CATProduct;*.cgr</FileExtensions>

FilterText - (Optional) Specifies the filter text for a translator add-in. The Language attribute can be specified for local languages, if not specified it defaults to English, below example sets the FilterText for French:

<FilterText Language="1036">Fichiers CATIA V5 (*.CATPart;*.CATProduct;*.cgr)</FilterText>

SupportsSaveCopyAs - (Optional) Specifies whether the translator add-in supports the Save Copy As. Value can be 0 or 1. Assumed to be false (0) if this value is not specified. If set to true (1), the FilterText will be availabe in the Save Copy As dialog.

SupportsSaveCopyAsFrom - (Optional) Specifies which documents the translator add-in supports to export. Valid values are the Inventor documents extensions with semicolon as delimiter. Below is sample to specify the SupportsSaveCopyAsFrom:

<SupportsSaveCopyAsFrom>.ipt;.iam</SupportsSaveCopyAsFrom>

SupportsOpen - (Optional) Specifies whether the translator add-in supports to open a foreign data. Value can be 0 or 1. Assumed to be false (0) if this value is not specified.

SupportsOpenInto - (Optional) Specifies which documents the translator add-in supports to open into. Valid values are the Inventor documents extensions with semicolon as delimiter. Below is sample to specify the SupportsOpenInto:

<SupportsOpenInto>.ipt;.iam</SupportsOpenInto>

SupportsImport - (Optional) Specifies whether the translator add-in supports importing data. Value can be 0 or 1. Assumed to be false (0) if this value is not specified. If set to true (1), the FilterText will be availabe in the Open dialog.

SupportsImportInto - (Optional) Specifies which documents the translator add-in supports to import into. Valid values are the Inventor documents extensions with semicolon as delimiter. Below is sample to specify the SupportsImportInto:

<SupportsImportInto>.ipt;.iam</SupportsImportInto>

Where to Put Your Files

Now that you have your add-in dll and have created your .addin file you need to know where to place those files so that Inventor can find and load your add-in.

The following four locations are supported. You can choose any one of the four depending on the needs of your add-in. Your .addin file can exist in any one of the following four locations or any subdirectory. The "%XXXX%" portion of each of the paths is an operating system defined variable. When using Explorer you can enter it as part of the path and Explorer will evaluate it to use the actual path defined by the variable.

1. **All Users, Version Independent**
   Windows 10/11 - %ALLUSERSPROFILE%\Autodesk\Inventor Addins\
2. **All Users, Version Dependent**(Since Invetnor 2024 the below folder is used to place the .addin manifest files instead of the legacy %ALLUSERSPROFILE%\Autodesk\Inventor 20xx\Addins\ folder)
   Windows 10/11 - %PROGRAMFILES%\Autodesk\Inventor 20xx\Bin\Addins\
3. **Per User, Version Dependent**
   In Windows 10/11 - %APPDATA%\Autodesk\Inventor 20xx\Addins\
4. **Per User, Version Independent**
   In Windows 10/11 - %APPDATA%\Autodesk\ApplicationPlugins

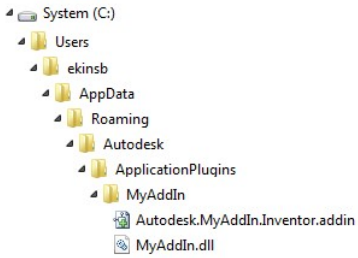
There are a couple of things to consider when determining where to put your add-in. If you choose a location that is available to all users it will require administrator privileges to install your add-in. In most cases, computers are rarely shared amongst multiple users so a per-user installation is usually sufficient.

If you plan on actively updating your add-in for each release of Inventor then making it version dependent can be good so that the user will only have access to an add-in that was written for and tested with the version of Inventor they're using. Because significant effort is made to allow the API to be upward compatible you should be able to run older add-ins with newer versions of Inventor. Because of this you can supply and add-in and not tie it to a specific version assuming that it will continue to run as newer versions of Inventor are released.

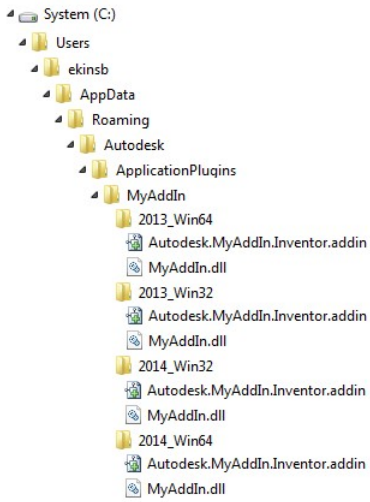
Also, because you can specify versions that your add-in is compatible within the .addin file you can still use a version independent .addin location and control the versioning through the .addin file.

Drag and Drop Add-In Deployment

A new feature added in Inventor 2013 is the ability to create an add-in that can be deployed by simply copying a directory into a certain location on the user's computer. The change made to enable this is that Inventor looks in the four directories above for .addin file and now also looks in any subdirectories within those four directories. If you create a directory that contains your .addin file and your addin dll and set the Assembly element of your .addin file to have just the name of your dll without any path, you can copy that folder to any of the folders above and Inventor will find and load your addin. Below is an example of the directories and files for an add-in named "MyAddin". This shows the full path for the "%APPDATA%\Autodesk\ApplicationPlugins" location.



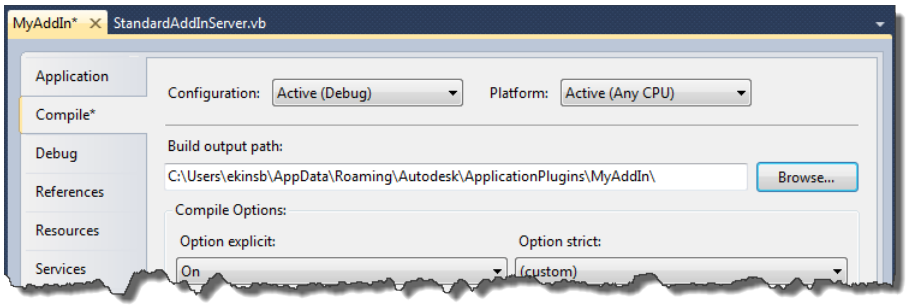
It's also possible to create much more complex configurations for a drag and drop type of add-in. For example you can have an add-in where you have one dll for 32-bit and another one for 64-bit. You might also have a dll that is specific to a certain version of Inventor and a different dll for another Inventor version. The possible directory structure to support this is shown below. In order for it to support multiple versions of Inventor you must put the folder in one of the two version independent folders. An example of how the folders might look for an add-in like this is demonstrated in the picture below. What's happening here is that you are essentially delivering four different add-ins, each with their own .addin file and dll. Which version and which OS the add-ins load is controlled by the contents of the .addin file for each of the add-ins.



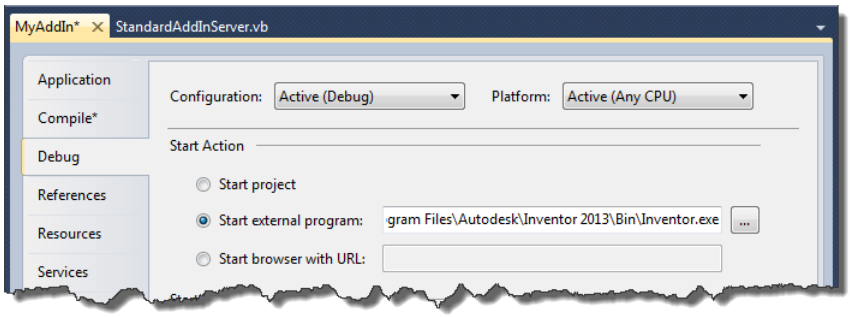
Debugging Your Add-In

Once you've compiled your add-in and have a dll you'll need to follow the steps below in order to enable debugging. With registry-free add-ins there isn't any difference between debugging with 32 or 64-bit Inventor.

1. You need to copy your .addin file and the add-in dll into one of the locations where Inventor looks for .addin files, as described above. This allows Inventor to find your add-in.
2. You need to change setting in your project so that when you compile your project, the dll is output to the directory above. This is so Visual Studio and Inventor will be using the same dll. The project setting to change is shown below.



3. You need to set the debug setting so that when you start your project from Visual Studio it will automatically start Inventor. The project setting to change is shown below.



Now you can add break points to your add-in and begin debugging (F5) from Visual Studio. Inventor should begin running and your break points should be hit when those lines are executed.

Writing Your Add-In Code

The actual code of your add-in isn't anything specific to an add-in but uses the Inventor API to interact with Inventor. The only thing that's somewhat unique to an add-in is that some API calls have an argument for a *ClientId*. This is the ClientID of your add-in as identified in your .addin file. This is typically used when you're creating user-interface elements in Inventor. Inventor uses this to remember which add-in created what.

Converting an Existing Add-In to be Registry-Free

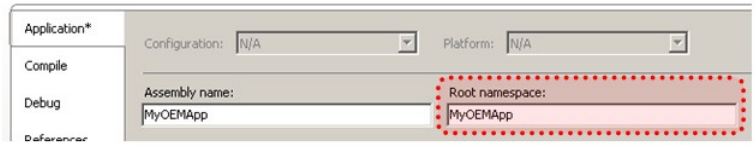
The following describes the process of converting a standard add-in into a registry-free add-in. Since the process is different for the different programming languages, the process is described for Visual Basic, C#, and VC++.

Making your Visual Basic Add-In Registry Free

1. Create a new file in the same folder as your project file with the name "MyOEMApp.X.manifest", where MyOEMApp will be replaced with the name of your add-in. Add the following to the manifest file. The portions highlighted in yellow need to be edited to match your add-in.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity name="MyOEMApp" version="1.0.0.0" />
  <clrClass clsid="{2bf59d73-5170-4524-b0e1-391760aaffa5}"
            progid="MyOEMApp.StandardAddInServer"
            threadingModel="Both"
            name="MyOEMApp.MyOEMApp.StandardAddInServer"
            runtimeVersion="" />
  <file name="MyOEMApp.dll" hashalg="SHA1" />
</assembly>
```

The “name” attribute of the clrClass element consists of three parts, separated by periods. The first is the name of the root namespace, which is highlighted below.

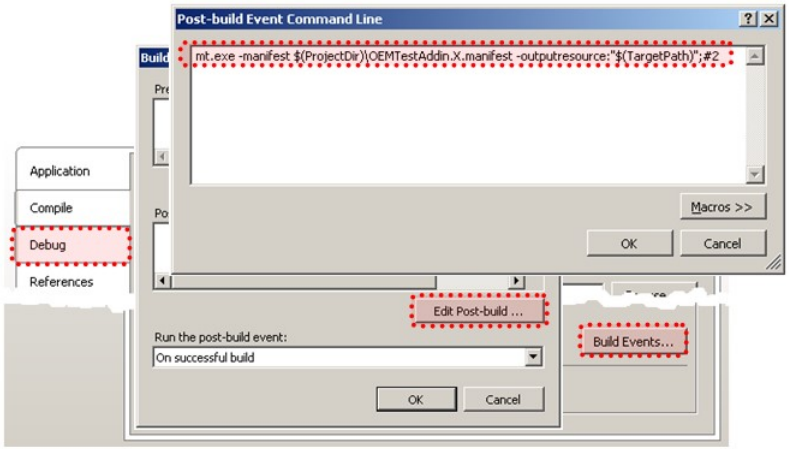


The second part is the namespace that the COM class is defined within. For this example is it MyOEMApp and is highlighted below. The Last piece is the name of the class that implements the ApplicationAddInServer interface, which is “StandardAddInServer” in this example and is also highlighted below.

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

Namespace MyOEMApp
    <ProgIdAttribute("MyOEMApp.StandardAddInServer"), _
    GuidAttribute("2bf59d73-5170-4524-b0e1-391760aaffa5")> _
    Public Class StandardAddInServer
        Implements Inventor.ApplicationAddInServer
```

2. The next step is to add a post-build process to incorporate this manifest into your dll. The post-build process calls mt.exe which is a Microsoft utility that will embed the manifest into your add-in's dll. You define a post-build step through the Compile tab of the Application Properties dialog. On the Compile tab, click the "Build Events…" button and then on the "Build Events" dialog click the "Edit Post-build…" button. Finally enter the lines below into the "Post-build Event Command Line" dialog, as shown below; OEMTestAddin is the name of your add-in.

```
call "%VS100COMNTOOLS%vsvars32"
mt.exe -manifest "$(ProjectDir)OEMTestAddin.X.manifest" -outputresource:"$(TargetPath)";#2
```



3. Remove any code associated with registering the add-in in the registry. This typically just means removing the Register and Unregister methods from your add-in class. The AddInGuid property is in the same region as the registration functions, so if you intend on using this property in other areas of your add-in you'll want to be careful not to delete it.

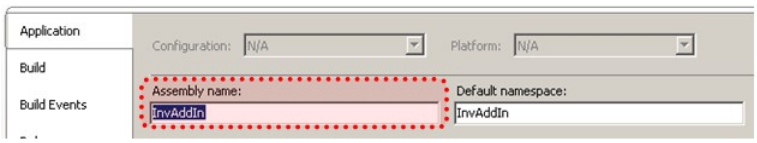
Skip to step 4 below, which is the same for all languages.

Making your C# Add-In Registry Free

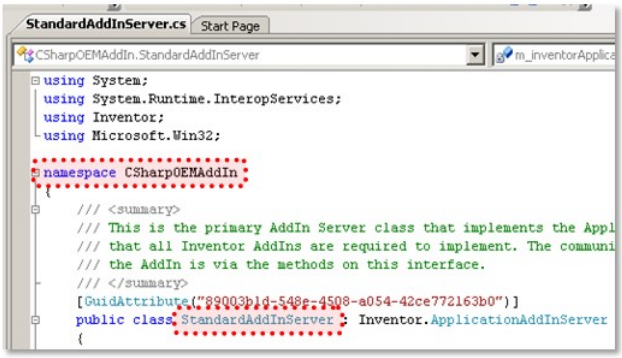
1. Create a new file in the same directory as your project file and name it "CSharpOEMAddIn.X.manifest", where CSharpOEMAddIn is the name of your add-in. Add the following to the manifest file. The portions highlighted in yellow need to be edited to match your add-in.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity name="InvAddIn" version="1.0.0.0" />
  <clrClass clsid="{2bf59d73-5170-4524-b0e1-391760aaffa5}"
            progid="CSharpOEMAddIn.StandardAddInServer"
            threadingModel="Both"
            name="CSharpOEMAddIn.StandardAddInServer"
            runtimeVersion="" />
  <file name="InvAddIn.dll" hashalg="SHA1" />
</assembly>
```

The "name" attribute of the assemblyIdentity element is the name of the assembly, which is highlighted below.

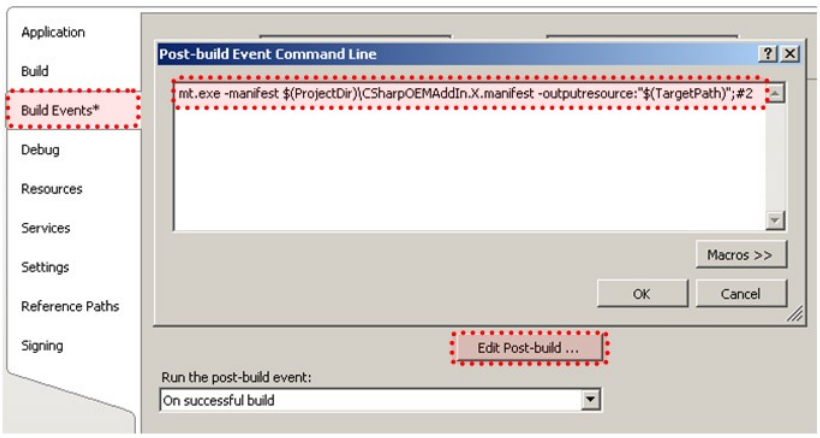


The "name" attribute of the clrClass element consists of two parts, separated by periods. The first is the name of the namespace that the COM class is defined within. For this example is it CSharpOEMAddIn, and the second part is the name of the class that implements the ApplicationAddInServer interface, which is "StandardAddInServer". Both of these are highlighted below.



2. The next step is to add a post-build process to incorporate this manifest into your dll. The post-build process calls mt.exe which is a Microsoft utility that will embed the manifest into your add-in's dll. You define a post-build step through the Build Events tab of the Application Properties dialog. On the Build Events tab, click the "Edit Post-build…" button. Enter the string below into the "Post-build Event Command Line" dialog, as shown below. Replace InvAddin with the name of your add-in.

```
call "%VS100COMNTOOLS%vsvars32"
mt.exe -manifest $(ProjectDir)INVAddin.X.manifest -outputresource:"$(TargetPath)";#2
```



3. Remove any code associated with registering the add-in in the registry. This typically just means removing the Register and Unregister methods from your add-in class. The AddInGuid function is in the same region as the registration functions, so if you intend on using this property in other areas of your add-in you'll want to be careful not to delete it.

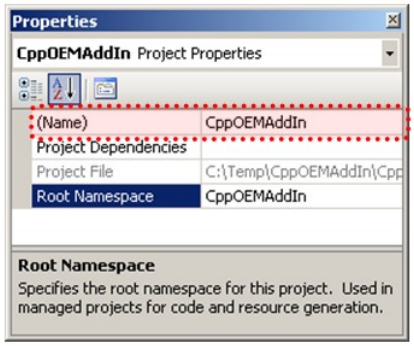
Skip to step 4 below, which is the same for all languages.

Making your C++ Add-In Registry Free

1. Create a new file in the same directory as your project file and name it "CPPOEMAddIn.X.manifest", where CPPOEMAddIn is the name of your add-in. Add the following to the manifest file. The portions highlighted in yellow need to be edited to match your add-in.

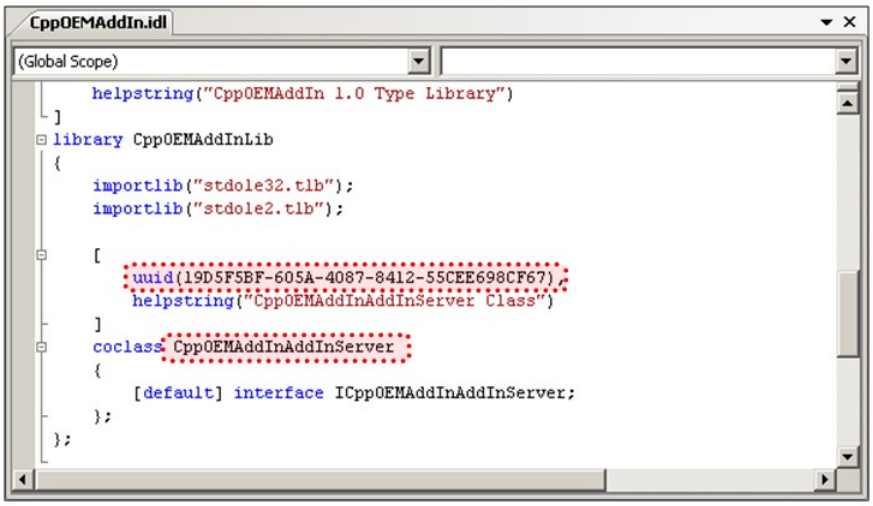
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity name="CPPOEMAddIn" version="1.0.0.0" />
  <clrClass clsid="{2bf59d73-5170-4524-b0e1-391760aaffa5}"
            progid="CppOEMAddIn.CppOEMAddInAddInServer"
            threadingModel="Both"
            name="CppOEMAddIn.CppOEMAddInAddInServer"
            runtimeVersion="" />
  <file name="CppOEMAddIn.dll" hashalg="SHA1" />
</assembly>
```

The "name" attribute of the assemblyIdentity element is the name of the project, which is highlighted below.

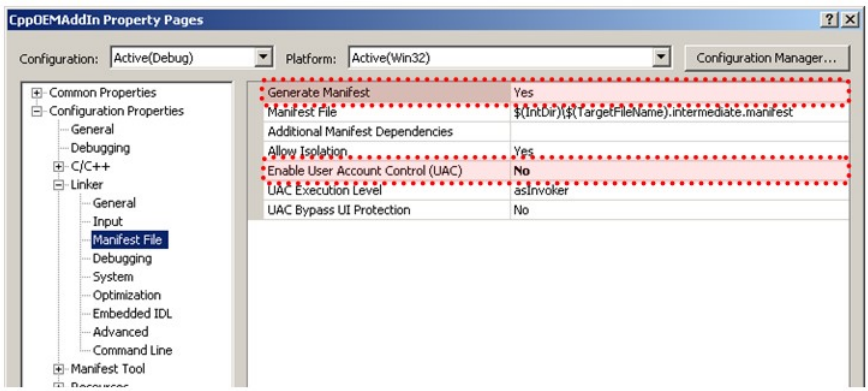


The clsid is the class ID for your add-in and can be found in the .idl file associated with your add-in, which is shown below. Remember that the braces are required in the manifest file.

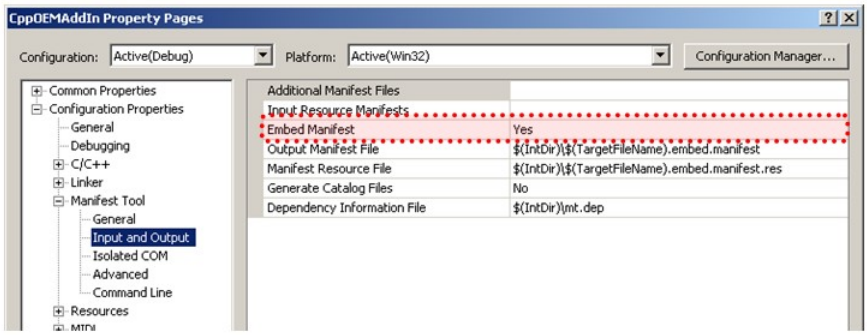
The “name” attribute of the clrClass element consists of two parts, separated by periods. The first part is the name of your project. For this example it is CppOEMAddIn. The second part is the name of the class that implements the ApplicationAddInServer interface, which is “CppOEMAddInAddInServer” for this example, as shown below.



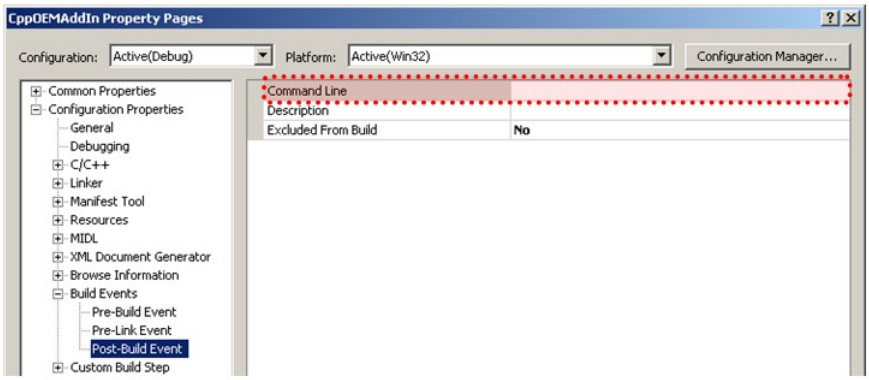
2. The next step is to configure some project settings so the manifest is correctly embedded within the dll. The first two settings are part of the Linker settings. The settings and the values to change them to are highlighted in the picture below. Some of these settings may already have the correct value.



The next setting is of the Manifest Tool settings. Make sure the “Embed Manifest setting is set to “Yes”, as shown below.



Finally you want to remove the post-build step that registers the add-in in the registry. This is found in the Build Events section of the settings. Clear out any text in the Command Line field, as shown below.



3. Remove any code associated with registering the add-in in the registry. This typically just means removing the Register and Unregister methods from your add-in class. The AddInGuid function is in the same region as the registration functions, so if you intend on using this property in other areas of your add-in you'll want to be careful not to delete it.

   Skip to step 4 below, which is the same for all languages.

====== The following steps apply to all languages. ======

4. Compile your project. This will cause the manifest to be embedded within your dll. If you get any errors when compiling, verify that the command line that you entered matches the one above.

   You can verify that the manifest has been correctly embedded by dragging the add-in dll into Visual Studio. You should see the "RT_MANIFEST" folder, as shown below. Double clicking on the "2" icon will open a window to show you actual manifest data.

5. A requirement for all registry-free add-ins is to have an .addin file. With a registry based add-in, Inventor relied completely on the registry to get information about the add-in. The .addin file takes the place of the registry for this same information. The addin filename is in the form of Autodesk.AddInName.Inventor.addin Below is the .addin file for an example primary add-in with the portions to change highlighted.

   Please note: When you manually copy the content to a .addin file, make sure there are no leading spaces and blank lines in it.

```
<?xml version="1.0" encoding="utf-16"?>
<Addin Type="Standard">
        <ClassId>{2bf59d73-5170-4524-b0e1-391760aaffa5}</ClassId>
        <ClientId>{2bf59d73-5170-4524-b0e1-391760aaffa5}</ClientId>
        <DisplayName>MyOEMApp</DisplayName>
        <Description>Inventor OEM Sample App</Description>
        <Assembly>MyOEMApp\MyOEMApp.dll</Assembly>
        <Hidden>0</Hidden>
</Addin>
```

   For most add-ins, the **ClassId** and **ClientId** elements will be the same value, which is the GUID at the top of the StandardAddInServer class.

   The value of the **DisplayName** element can be anything and is the name of the add-ins as shown in the Add-In Manager.

The **Description** element can also be anything and is the description of the add-in as shown in the Add-In Manager.

The **Assembly** element defines the name of the add-in dll. It also defines the path to the dll, relative to the OEM bin directory, but the subdirectory must always have the same name as the add-in dll.

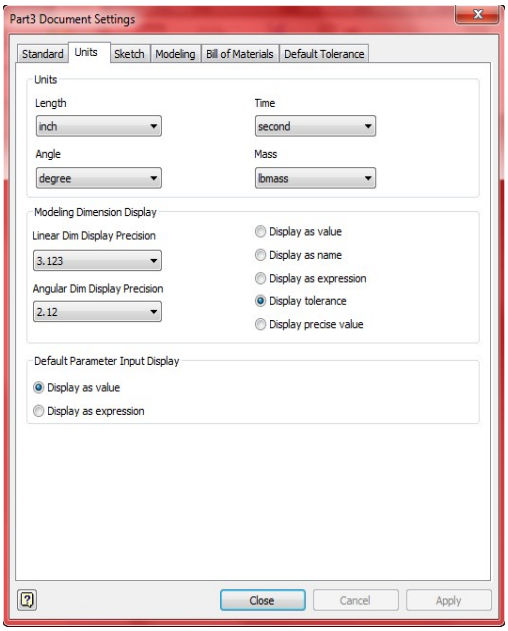
The **Hidden** element defines whether the add-in is visible in the Add-In Manager or not. A value of 1 indicates that it is hidden, although the end-user can still right-click within the Add-In Manager and choose “Show hidden members” to display all add-ins.

There are some other elements that you might see defined for other add-ins that either don’t apply or are ignored for an OEM primary add-in.

6. See the Creating an Inventor Add-In topic for more information about where the files go.

Unit of Measure

The units of measure portion of the API is a set of utility functions that allow you to perform several types of tasks related to units within Autodesk Inventor. The first of these tasks is getting and setting the current display units and their precision. The display units and their precision are attributes of the document and are saved with the particular document they are set for. The API access to the units and precision is equivalent to the functionality provided in the user interface through the Units tab of the Document Settings dialog, as shown below.



By setting the default units and precision you control how Autodesk Inventor displays all values. For example, if you set the length units to be inches and three decimal place precision, all display of lengths will be in inches and have three decimal place precision. Whenever you need to provide a value for a length, like when specifying the length of an extrusion, and you enter a value without specifying any units, it is assumed to be the default unit. For example if you enter "5" as the length of an extrusion it's assumed to be five inches since that is the default length unit for the document.

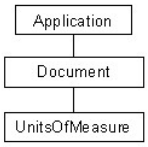
The user can change the default units and precision at any time, without impacting the model in any way. A peek within the internals of Autodesk Inventor will provide a better understanding of how the units work. Internally, Autodesk Inventor uses a consistent set of units regardless of what the user has specified as the document default. The precision is always double-precision floating point, regardless of the precision specified by the user. The internal units used by Inventor for the various types of units are listed below:

Unit Category	Measure
Mass	Kilogram
Length	Centimeter
Time	Second
Temperature	Kelvin
Angle	Radian

When the user uses the Units tab to specify display units, this does not affect the internal database units in any way. The data is always saved and utilized using the database units listed above. Autodesk Inventor uses the display units whenever it needs to interact with the user and display unit information or when it needs to interpret unit information the user has entered. For example, let's look at the extrusion depth in more detail. When the user specifies "5" as the depth of the extrusion, the document length unit is used to interpret this as five inches. This value is converted to the internal length unit of centimeters and stored as a double precision value. All internal calculations use this value. When the length of the extrusion needs to be shown to the user, the internal centimeter value is converted to the current document unit with the specified precision and displayed. The internal value for the length, however, remains in centimeters with the full double precision accuracy.

The API works the same way as Autodesk Inventor does internally. All API functions use the database units listed above as input and output. When you need to display unit information to the user you can use the Units of Measure portion of the API to create the correct string to display. When the user enters unit information you can use the API to convert the input string to the correct database unit value. This isolates the majority of your code from any unit conversion since you can assume it will always be in database units. The only time you need to worry about units is when you interact with the user.

The object hierarchy for UnitsOfMeasure is shown below. It consists of the Document object supporting the UnitsOfMeasure property, which returns the UnitsOfMeasure object. This object supports various methods and properties to allow getting and setting the current document default units and performing conversions between the document units and the database units.



Let's look at some examples of the use of the UnitsOfMeasure object. The AngleUnits, AngleDisplayPrecision, LengthUnits, LengthDisplayPrecision, MassUnits and TimeUnits properties allow you to get and set the document default angle, length, mass and time units and set the precision for angles and lengths. They provide an exact equivalent to the functionality of the Units tab of the Document Settings dialog. (The LengthDisplayPrecision is used for the precision of mass and time.) The GetValueFromExpression, GetStringFromValue, GetStringFromType and GetTypeFromString methods provide functionality in the API that is not exposed to the end user. Let's look in detail at the GetValueFromExpression method to understand units as they apply to the API. If you look at this function in the object browser you'll see the following prototype.

```
GetValueFromExpression(Expression As String, UnitsSpecifier) As Double
```

This method has two input arguments and returns a Double value. The first argument, Expression, is a string that contains the value entered by the user. For example, let's say the user entered "3.2" in a dialog where a length is expected. The second argument, UnitsSpecifier, doesn't have a type specified, which means it is a Variant. Valid input for this argument is either a value from the UnitsTypeEnum enum list or a string. Most common unit types are defined as an enum in the enum list. For example, if you want the input string to be evaluated as an inch you use the enum value kInchLengthUnits. A more common use is that you don't know the specific unit, only that the value is a length, and want it evaluated using the current document default for length. In this case you use the enum value kDefaultDisplayLengthUnits. Corresponding values exist for the other unit categories of angle, mass, and time.

Like the GetValueFromExpression method just discussed, many of the methods supported by the UnitsOfMeasure object require to you to specify the type of unit you are working with. For example, let's say you have a dialog where the user needs to enter a length and they've entered "3.2". Using the GetValueFromExpression method you can pass in the string and the expected unit type and get back the value in the appropriate database value. The code below takes a string from a text box and evaluates it as a length string. Error checking is performed around the GetValueFromExpression call because if the input string does not define a valid length an error will occur. If successful, dLength will contain the resulting length in database length units (centimeters).

```
' Set a reference to the UnitsOfMeasure object of the active document.
Dim oUOM As UnitsOfMeasure
Set oUOM = ThisApplication.ActiveDocument.UnitsOfMeasure

' Get the length defined by the contents of a text box.
Dim dLength As Double
On Error Resume Next
dLength = oUOM.GetValueFromExpression(txtLength.Text, kDefaultDisplayLengthUnits)
If Err Then
    MsgBox "Invalid length specified."
End If
On Error GoTo 0
```

As seen above, when using an enum as the input for the UnitsSpecifier argument of the UnitsOfMeasure, you can specify a particular unit or whatever the current document default is for a unit category. This argument can also be a string to allow you to specify units that are not defined in the UnitsTypeEnum enum list. For example, all of the volume measurements are those used primarily for liquids: cup, gallon, liter, ounce, pint, and quart. If you want to use a different volume measurement, i.e. cubic inches, you'll need to define the unit using a string. The following example creates a string that defines volume units using the current default document length unit.

```
' Get the enum value that defines the current default length units.
Dim eLengthUnit As UnitsTypeEnum
eLengthUnit = oUOM.LengthUnits

' Get the equivalent string of the enum value.
Dim sLengthUnit As String
sLengthUnit = oUOM.GetStringFromType(eLengthUnit)

' Create a string that defines a volume using the current length unit.
Dim sVolumeUnit As String
sVolumeUnit = sLengthUnit & "^3"

' Create a string showing the volume in the current units.
Dim sVolume As String
sVolume = oUOM.GetStringFromValue(36.567, sVolumeUnit)
MsgBox "Parts volume: " & sVolume
```

The first section gets the enum value that specifies the current length unit. To define a custom unit we need a string, so the next step calls the GetStringFromType to get the string that is equivalent to the enum value. Let's assume the document default length unit has been set to inches. The call to the LengthUnits property will return kInchLengthUnits. Passing this value into the GetStringFromType method will return "inch". The next step uses this string to create a string that defines a volume. In this case, the result is "inch^3", for cubic inches. This string is used as the UnitsSpecifier argument in the call of the GetStringFromValue method to convert the value 36.567 to cubic inches and return a correctly formatted string. The result is "2.231 in in in". The input value is assumed to be in database units or in this case cubic centimeters.

The ability to use a string to define units is also important in the cases where you need complex composite units. For example, density is not defined in the constant list but defining the correct string can create a unit type for density. For example, the string "gm / cm^3" will define grams per cubic centimeter as the unit type. The UnitsOfMeasure sample program, delivered with Autodesk Inventor, demonstrates much of the functionality of the UnitsOfMeasure object.

Transient Geometry

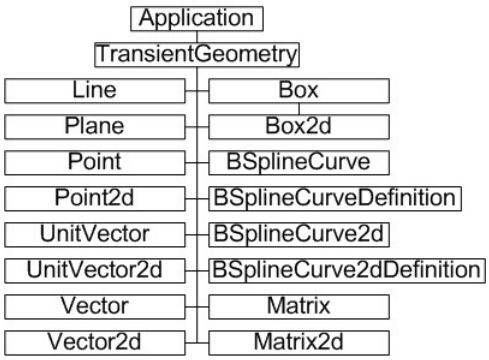
Introduction to Transient Geometry objects

The term transient geometry object can be interpreted many ways, even in the context of Autodesk Inventor. For the purposes of this overview, this topic covers those utility constructs required to accomplish many mathematical tasks, specifically geometric tasks, through the API. For example, point objects, vectors and matrices.

The purpose of Transient Geometry

These transient objects, created through the appropriate method provided by the TransientGeometry object, are used extensively throughout Autodesk Inventor application code. Unlike most other API geometry objects, they are invisible mathematical representations with no graphical component. They are typically the means through which graphical objects are manipulated, as they provide a convenient way to create 2D and 2D points (not workpoints or sketchpoints), create matrices for object transformations, and so on.

Transient Geometry Object Model Diagram



Working with Transient Geometry Objects - the Point and Matrix objects

An important point to remember about these transient geometry objects is that they do not represent the boundaries of actual graphical objects. For example - the Line object is infinite in extent, unlike a SketchLine object. The Line object has a direction, represented by a UnitVector. Similarly, the Plane object is not bounded like a face - it too is infinite in extent.

This overview will look at some of the most commonly used transient objects - the Point, Point2d and Matrix objects.

Using a Point2d object

The Point object is a list of three coordinates, representing the X Y and Z of a point in space. Similarly, the Point2d object represents the X and Y of a point on a plane. These objects also support properties giving access to their coordinates, and to methods that perform functions such as equality testing or transformation of the point.

Point2d objects are frequently used when working with 2D sketches. To create a SketchPoint object on a sketch, use the Add method of that sketches SketchPoints collection. You might expect code similar to the following.

```
Dim oSkPnts As SketchPoints
Call oSkPnts.Add(10, 20)  ' This is wrong
```

The Add method needs to know the location to create the SketchPoint object, but this point may be the result of any number of transformations or translations, or it may be provided by another object. Therefore the Add method expects a Point2D object. The correct code is as follows. The code omits error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

```
Dim oApp As Inventor.Application
Set oApp = ThisApplication

Dim oPartDoc As PartDocument
Set oPartDoc = oApp.Documents.Add(kPartDocumentObject, _
   oApp.GetTemplateFile(kPartDocumentObject))

Dim oSketch As PlanarSketch
Set oSketch = oPartDoc.ComponentDefinition.Sketches.Add _
  (oPartDoc.ComponentDefinition.WorkPlanes.Item(3))

Dim oTG As TransientGeometry
Set oTG = oApp.TransientGeometry

Dim oSkPnts As SketchPoints
Set oSkPnts = oSketch.SketchPoints
Call oSkPnts.Add(oTG.CreatePoint2d(10, 20), False)
```

The preceding code will create a sketch with a single sketchpoint at X=10, Y=20. The last argument of the Add method is a boolean value, indicating whether the SketchPoint will form the center point of a hole feature. The main difference is that the graphical representation of the resulting SketchPoint will differ slightly.

SketchPoints in a SketchPoints collection are always 2D, and are typically provided by the CreatePoint2D method. SketchPoint3D objects in a 3D sketch are contained within a SketchPoints3D collection.

Using the Matrix object

The Matrix object is one of the least understood objects in the Autodesk Inventor API, but its use in most common situations is fairly straightforward. This overview will focus on the Matrix object used for 3D space, though Autodesk Inventor also supports the Matrix2d object for 2D space.

The Matrix object is typically used either to place an object with a specific orientation, such as when inserting an instance of a part into an assembly. In other words it is used to define a coordinate system. Also, it is used to move, or transform, an object from one location to another. The Matrix object provides a convenient means of manipulating a set of coordinates in 3D space. It has a set of useful methods to set the matrix data or to act on that data.

For example, suppose you have two planes, and you plan to move objects from one plane to the other. Ideally, you would use a matrix to perform these translations most efficiently. The Matrix object provides the SetToAligh CoordinateSystems method that accepts the coordinates system (origin, X, Y and Z) of the source plane and the coordinate system (origin, X, Y and Z) of the target plane. Thus the matrix rows and columns are set to the required values and the matrix can be used repeatedly to translate any objects from the old plane to the new. The Matrix object provides a number of methods and properties designed to shield the user from many of the complexities of matrix math.

A matrix as a coordinate system

Let's first look at a matrix as a coordinate system definition. The default Autodesk Inventor coordinate system follows the conventional rectangular Cartesian system with the positive X direction to the right, the positive Y direction upwards, and the positive Z direction towards you. We'll set a matrix to represent a similar system, but one that is rotated 45 degrees about the Z axis, and with the origin at 10,5,0.

Imagine a vector pointing along the required X axis. Remember we now want this to be at 45 degrees, so this can be thought of as pointing at 1,1,0. However, it's often convenient to use the UnitVector object, which always has a unit length of 1. So the direction vector actually points to 0.707, 0.707, 0. The Z axis does not change, but the origin becomes 10,5,0. A 3D matrix to represent this would appear as follows. Reading the columns left to right, they are X Y Z, and origin.

0.707	-0.707	0	10
0.707	0.707	0	5
0	0	1	0
0	0	0	1

Using the previously noted SetCoordinateSystem method, the following code calculates and sets a matrix object to the above values. Autodesk Inventor internal units are centimeters and radians.

```
Dim oTG As TransientGeometry
Set oTG = ThisApplication.TransientGeometry
Dim dPi As Double
dPi = Atn(1) * 4

' Define the origin point.
Dim oOrigin As Point
Set oOrigin = oTG.CreatePoint(10, 5, 0)

' Define the axis vectors.  Pi/4 is 45 degrees in radians.
Dim oXAxis As UnitVector
Dim oYAxis As UnitVector
Dim oZAxis As UnitVector
Set oXAxis = oTG.CreateUnitVector(Cos(dPi / 4), Sin(dPi / 4), 0)
Set oYAxis = oTG.CreateUnitVector(-Cos(dPi / 4), Sin(dPi / 4), 0)
Set oZAxis = oTG.CreateUnitVector(0, 0, 1)

' Create the matrix and define the desired coordinate system.
Dim oMatrix As Matrix
Set oMatrix = oTG.CreateMatrix
Call oMatrix.SetCoordinateSystem(oOrigin, oXAxis.AsVector, oYAxis.AsVector, oZAxis.AsVector)
```

A typical use for such a matrix definition would be placement of parts within an assembly. You can construct a matrix that, when referenced by the Transformation property of a part occurrence, determines exactly where the part should be placed.

Note: Individual matrix cells can be accessed directly through the Cell method of the Matrix object.

Using a matrix for transformation

The use of a matrix to move an object is a little different than using a matrix to define a coordinate system for object placement. The example shown previously defines a static coordinate system. To move objects from one location to another implies a difference in location coordinates. In other words, a delta - the difference between one coordinate system definition and another. This delta can be represented by a matrix.

The matrix defined in the preceding code, when referenced by the TransformBy method of a second matrix object, results in the second matrix being recalculated to reflect the new location. The referenced matrix acts as a delta matrix for the transformation. In other words, a new matrix definition is created from a source matrix and a delta matrix.

In this case, the delta is a 45 degree rotation about the Z axis. All we need to do is change the call to SetCoordinateSystem, as we no longer wish to change the origin, as follows.

```
Call oMatrix.SetCoordinateSystem(oTG.CreatePoint, oXAxis.AsVector, oYAxis.AsVector, oZAxis.AsVector)
```

Thereafter, to rotatee an object such as a part occurrence about its Z axis, obtain its location matrix through its Transformation property, then modify the matrix by calling its TransformBy method, referencing the delta matrix. Then reapply the updated location matrix to the occurrence, again through its Transformation property. The occurrence will move to its new location.

The matrix object supports a set of methods intended to help with many common tasks, as follows.

GetCoordinateSystem	Invert	PostMultiplyBy
PutMatrixData	SetToAlignCoordinateSystems	SetToRotateTo
SetTranslation	GetMatrixData	IsEqualTo
PreMultiplyBy	SetCoordinateSystem	SetToIdentity
SetToRotation	TransformBy	

Summary

The TransientGeometry object allows the creation of a number of transient geometry objects. These objects do not represent actual graphical geometry. Transient geometry objects are used for mathematical calculations. Two of the most commonly used objects include points and matrix objects. Matrices allow accurate placement and movement of components.

Also consider

TransientGeometry objects are not limited to point, line or planar objects. For example, the Box object can expand to incorporate specified points, or can check for interference with another box, or can check if a specified point is contained within the box.

Proxies

Introduction to proxies

For the Autodesk Inventor user, the concept of proxy files relates to the use of AutoCAD files in the Autodesk Inventor environment. Proxies in the API context mean something different entirely.

An object proxy is a reference to an object through a particular occurrence. Think of it as a pairing with its respective native object, but returns context-specific data.

An example is a face of a part within an assembly. When this face is queried through the API, what do the results mean - are coordinates returned in the context of the assembly, or the native part? What if the part is inserted into the assembly multiple times; how is a particular instance of the face identified? A key purpose of proxies is to provide answers to these questions. In this case, the proxy of the face object performs the necessary transformations to provide point data in the required context, and also provides information about which occurrence the face is associated with.

The purpose of proxies

To the user, the notion of proxies is transparent. When a user selects a part through the Autodesk Inventor user interface, conceptually the part is in the assembly. It doesn't matter to users that they are working with a proxy that is presenting coordinates and faces and so on in the assembly context and not in the context of the referenced part. But developers need to obtain data specifically in the part or assembly context. There is no geometry in the assembly, only a proxy that is performing the required transformations based on the position of the part within the assembly.

> **Note:** The idea of proxies can be extended to cover the ComponentOccurrence object (the concept is very similar). Consider nested assemblies, or subassemblies, where the level of nesting is three or more levels. An occurrence can be an instance of an assembly definition, not just a part definition. At the top level (the instance), this occurrence is a ComponentOccurrence object as expected, but at other levels these occurrences are represented by ComponentOccurrenceProxy objects.

There is no object model diagram for proxies. Proxies are present for anything that is available for selection in an assembly context, including constraints, features, faces, edges, loops, sketches, profiles, and work features.

Working with proxies through the API

Object proxies are paired with their respective native objects. In fact, proxies are derived from their native objects, and so have common methods and properties. However, an object proxy is a reference to an object through a particular occurrence. In other words, it's a reference to the native object through the occurrence path. This path is potentially long when working with deeply nested subassemblies, so there are means to traverse this path.

Proxy objects have two additional properties; NativeObject and ContainingOccurrence. NativeObject returns the definition of the object - the object without its containing occurrence context. So for a FaceProxy object, NativeObject returns the corresponding face object within the part definition. The ContainingOccurrence property returns the containing object - in this case, the occurrence that contains the face.

> **Note:** When working with an object or its proxy, be aware that calls to their respective Parent properties return different objects. The parent of a proxy object will also be a proxy object. For example, the parent of an EdgeProxy object may be a SurfaceBodyProxy.

The starting point at the top of the path is the ComponentOccurrence. The CreateGeometryProxy method of the ComponentOccurrence object creates the required proxy information for a given face, feature, constraint, and so on.

Sample demonstrating the difference between a proxy and its native object

The following code omits error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

Through the Autodesk Inventor user interface, create a simple part (for example, extrude a circle). Save the part, and then start a new assembly. Place your saved part in the assembly, unground the assembly, move it a little, then save the assembly.

The following code prints the X and Y coordinates of one corner of the range box that encompasses the extrude feature of the part. It demonstrates that different coordinates are returned for the proxy when compared to the native occurrence.

First, obtain the component occurrence from the assembly document. In this case it's simplified as we know the document contains only one.

```
Dim oDoc As AssemblyDocument
Set oDoc = ThisApplication.ActiveDocument

Dim oCompOcc As ComponentOccurrence
Set oCompOcc = oDoc.ComponentDefinition.Occurrences.Item(1)
```

Then obtain the part component definition to gain access to the part features.

```
Dim oPartCompDef As PartComponentDefinition
Set oPartCompDef = oCompOcc.Definition

Dim oExtrudeFeature As ExtrudeFeature
Set oExtrudeFeature = oPartCompDef.Features.ExtrudeFeatures.Item(1)
```

To get the proxy for the feature, use the CreateGeometryProxy method of the component occurrence, with the required feature as an argument.

```
Dim oExtrudeFeatureProxy As ExtrudeFeatureProxy
oCompOcc.CreateGeometryProxy oExtrudeFeature, oExtrudeFeatureProxy
```

Print some coordinate data, in this case the X Y values of one corner of the feature rangebox. These figures will differ, indicating that the first set come from the native object, while the second set come from the proxy object - in other words, the second set of values are in the assembly context.

```
Debug.Print oExtrudeFeature.RangeBox.MaxPoint.X
Debug.Print oExtrudeFeature.RangeBox.MaxPoint.Y

Debug.Print oExtrudeFeatureProxy.RangeBox.MaxPoint.X
Debug.Print oExtrudeFeatureProxy.RangeBox.MaxPoint.Y
```

The preceding code demonstrated that coordinates for a known point differed according to the context of the feature. Example output might look as follows.

```
 0.232857970569028
 0.96726442148297
 0.177086160521056
 0.988809173640979
```

In the assembly context, the part was moved; therefore the coordinates returned by the proxy reflect this move (the first and second lines in the output). These coordinates are different from those returned by the native object, the object definition, with no containing context (the third and fourth lines in the output).

Summary

An object proxy is a reference to an object through a particular occurrence. When the user picks an object (edge, face, loop and so on) in an assembly, they are really picking an object proxy. The various proxies available through the API are derived from their real counterparts and contain the same methods and properties, as well as others specific to proxies (such as NativeObject, which returns the object definition the proxy is paired with). The purpose of object proxies is to provide transformation and assembly context information for occurrences, features, constraints and so on. Object proxies are transparent to the Autodesk Inventor user.

Also consider

To fully understand object proxies, it is important to understand component occurrences and component definitions.

File and Document References

Files and Documents

A **File** object represents a storage in the file system (i.e. a file on disk). A **Document** object represents an instance of a model or drawing in memory. A Document can only have a single associated File object. However, since it is possible to have multiple instances (or model states) of a file in memory that are persisted in the same storage on the file system, multiple Document objects may be associated with the same File object.

File and Document References

A **FileDescriptor** object describes the reference from a File to another File. A **DocumentDescriptor** describes the reference from a Document to another Document. A descriptor contains all the information needed to find the referenced file/document as well as the state of the reference (healthy, unresolved, replaced, etc.). The File and FileDescriptor objects represent the consolidated view of all of the representations of a Document. The figure below shows the relationships between the FileDescriptor, File, DocumentDescriptor and Document objects.

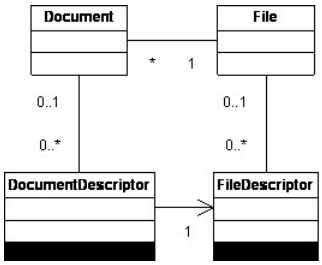


Figure 1 : The File References API Model

Relevant API Properties

Note: The Document interface is replaced by ApprenticeServerDocument in Apprentice. All the property names remain the same.

File Object

Property	Returned objects	Description
AvailableDocuments	Document objects	Documents currently in memory.
AllReferencedFiles	File objects	All the files referenced by this file (including recursively nested references).
ReferencingFiles	File objects	All the files in memory that directly reference this file.
ReferencedFiles	File objects	All the files directly referenced by this file. Unresolved & suppressed references are skipped.
ReferencedFileDescriptors	FileDescriptor objects	A collection describing all the direct references held by this file. This is a consolidated view of all the document references. Includes unresolved and suppressed references. Also includes foreign file references (xls, bmp, etc.)

Document (and **ApprenticeServerDocument**) Object

Property	Return type	Description
File	File object	The associated File object.
AllReferencedDocuments	Document objects	All the documents referenced by this document (including recursively nested references).
ReferencingDocuments	Document objects	All the documents in memory that directly reference this document.
ReferencedDocuments	Document objects	All the documents directly referenced by this document. Unresolved & suppressed references are skipped.
ReferencedDocumentDescriptors	DocumentDescriptor objects	A collection describing all the direct references held by this document. Includes unresolved and suppressed references.
ReferencedOLEFileDescriptors	ReferencedOLEFileDescriptor objects	A collection describing all the direct foreign file references held by this document. Includes unresolved references.
ReferencedOpaqueFileDescriptors	ReferencedOpaqueFileDescriptor objects	A collection describing all the in-direct foreign file references held by this document. Includes missing references.

FileDescriptor Object

Property	Return type	Description
FullFileName	String	Full path of the referenced file.
ReferencedFile	File object	The referenced file. Returns Null if reference is missing (unresolved or suppressed).
ReferenceMissing	Boolean	Whether the reference is missing for any reason and will the ReferencedFile property return a File object.

DocumentDescriptor Object

Property	Return type	Description
FullDocumentName	String	Full path of the referenced document.
ReferencedDocument	Document object	The referenced document. Returns Null if reference is missing (unresolved or suppressed).
ReferenceMissing	Boolean	Whether the reference is missing for any reason and will the ReferencedDocument property return a File object.
ReferenceSuppressed	Boolean	Whether the reference is suppressed.

The following code sample demonstrates the file references traversal and prints the file names of all the files referenced by the active document.

```
Public Sub FileReferenceSample()

  Dim oFile As File
  Set oFile = ThisApplication.ActiveDocument.File

  Call ProcessReferences(oFile)
```

```
End Sub

Private Sub ProcessReferences ( ByVal oFile As File )

  Dim oFileDescriptor As FileDescriptor
  For Each oFileDescriptor In oFile.ReferencedFileDescriptors

    Debug.Print oFileDescriptor.FullFileName

    If Not oFileDescriptor.ReferenceMissing Then

      ' Since the ReferenceMissing has returned False, the ReferencedFile will return a File
      ' Recurse unless this is a foreign file reference
      If Not oFileDescriptor.ReferencedFileType = kForeignFileType Then

        Call ProcessReferences(oFileDescriptor.ReferencedFile)
      End If
    End If
  Next
End Sub
```

Also consider

If you are working with very large assemblies, also refer to the Large Assembly Management (LAM) overview.

Consistent Materials (Materials and Appearances)

A Big Change from Materials and Render Styles

If you've used the Material and RenderStyle API objects in the past to work with materials and colors in Inventor there has been a large change in Inventor that has introduced completely new API objects for materials and colors.

Previous to Inventor 2013, Inventor had its own system for defining materials and colors, which was exposed through the API as the Material and RenderStyle objects. Recently Autodesk has introduced a common way of defining materials and colors that allows a single set of materials and colors to be used by all Autodesk applications. This way models transferred from one application to another will carry their material and will appear the same. This new Autodesk component is referred to as Consistent Materials.

Inventor 2013 was updated to use Consistent Materials, but the API was not. Instead, the existing Material and RenderStyle objects were re-plumbed to work on top of Consistent Materials. This works in most cases but since they're not a complete 1-for-1 match there are some things that don't work and other things you don't have access to. Most existing programs that still use the Material and RenderStyle objects should continue to work. Inventor 2014 introduces new API functionality specifically for Consistent Materials and new programs should begin to use the new API and existing programs that use the Material and RenderStyle objects can be upgraded as-needed. The Material and RenderStyle objects are now hidden and are no longer officially supported.

The new API functionality provides full access to consistent material information and provides support for using, editing, and creating new materials and colors and for working with the libraries that contain them. Consistent materials provides better integration between Autodesk applications and also supports additional capabilities relative to materials and colors. However, as a result of the additional capabilities and flexibility the API is not as easy to use as the previous Material and RenderStyle objects. The new consistent material API is discussed below.

Introduction to Consistent Materials

For those of you familiar with the Material and RenderStyle objects a brief comparison between with consistent materials will be useful. For those of you not familiar with the old API objects this will still be useful in introducing some of the concepts used in the consistent materials portion of the API.

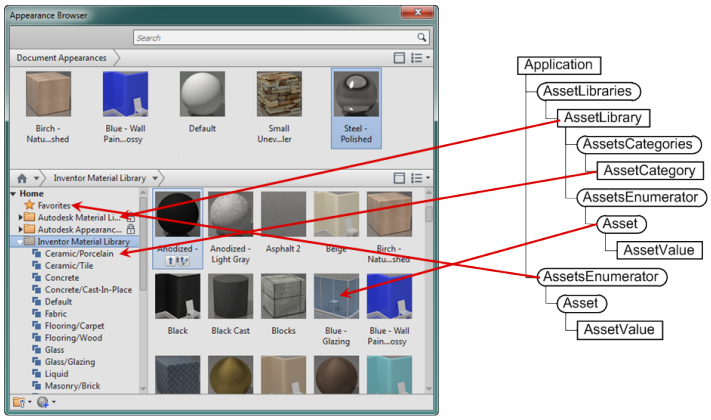
With the old Material and RenderStyle API there were distinct objects with properties that defined all of the information associated with that particular object. For example, the Material object had properties like Density, PoissonsRatio, SpecificHeat, etc. This made it simple to use from an API developer's standpoint because it was easy to see what the object supported and you could easily get and set the values associated with the object.

Consistent materials takes a different approach. Rather than having specific objects for each class of data and a unique property for each value, consistent materials takes a very general approach. The primary object in consistent materials is an Asset object, where Asset objects represent materials, physical properties, and colors (which are now referred to as appearances). An asset is essentially a collection of values. The set of values associated with an asset is not pre-defined by the API definition like it was with the Material and RenderStyle objects. You now have to determine, from either examining existing objects or using documentation, what values to expect for asset objects that represent certain types of materials, physical properties, or appearances. There are six API samples that make examining existing assets relatively easy by writing out all of the information associated with the material, appearance, and physical property assets.

- Write out all appearances
- Write out document appearances
- Write out all materials
- Write out document materials
- Write out all physical properties
- Write out document physical properties

Asset Libraries

Assets exist within libraries. A single library can contain multiple assets of different types. For example it's common to have a single library that contains material, physical property, and appearance assets. Libraries are represented by .adsk files. The Appearance Browser and Material Browser provide the user-interface to these libraries and the assets they contain. The Appearance Browser is shown below along with the portion of the API object model that provides access to these libraries and their assets.

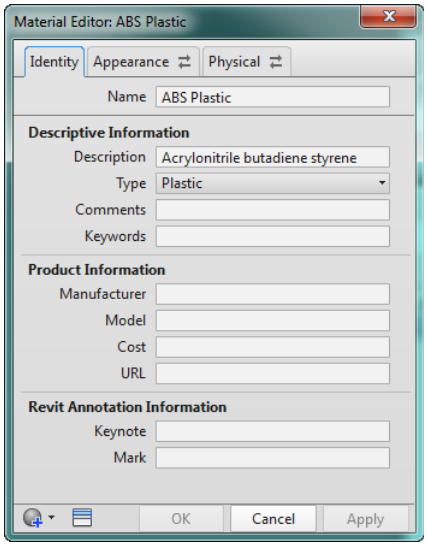


Assets are categorized within a library. The API also provides access to these categories and the assets within each category.

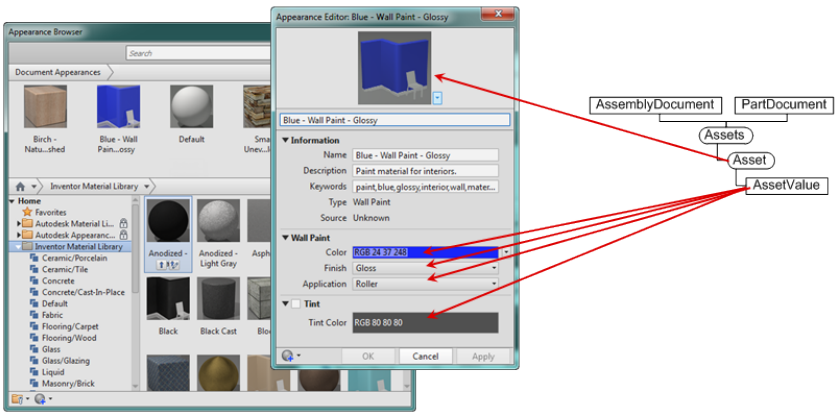
Libraries can be locked which means they're read-only and as a result any of the properties that edit a library will fail. Even in the case where a library is not locked, the types of edits you can do are limited; you can delete and rename assets but not create or modify them. Creation or editing of assets is only done on assets contained within a document. There is functionality to easily copy assets from one library to another or between libraries and documents. Using this you can copy and asset to a document, edit it, and copy it back to the library. To create a new asset you create it in a document and then copy it to a library.

Assets

As discussed above, an asset is a generic object that contains a collection of values. In Inventor there are three primary types of assets; materials, physical properties, and appearances. Material assets are the simplest. They primarily serve to group a physical asset and appearance together. Conceptually a material can be described by its physical properties (density, Poisson's ratio, yield strength, etc.) and how it looks or its appearance. A material property has some basic information that identifies it, (name, description, type, etc.), and it references a physical properties asset and an appearance asset. You can see this when using the user-interface too, as shown below. After selecting a material in the Material Browser, the attributes of the material are represented on three tabs in the Material Editor; Identity, Appearance, and Physical. The Appearance and Physical tabs allow you to view and edit the associated appearance and physical property assets.



As already discussed, an asset is a collection of values. These values can be of different types. They can be simple types like floats, integers, strings, and Booleans, but they can also be more complex types like color, filename, reference to other assets, textures, and choices. When editing an asset you need to be aware of what values to expect and their types. For example, choice asset values always need to be set with one of the pre-defined choices. The picture below shows the user-interface for editing an asset and the API objects that provide the equivalent functionality.



Using Assets

Consistent materials are more complicated to use than the previous Material and RenderStyle objects but for the most common operations there isn't too much difference. The most common use is to assign an existing material to a part or an appearance to an occurrence, part, body, feature, or face. It is possible to create new assets and edit existing ones but it's not that common to use the API to do that.

Below is an example that illustrates assigning an existing appearance to an assembly occurrence. This sample demonstrates the issue discussed earlier where an asset has to exist in the document before it can be used. This checks to see if the asset is already local and if it is uses it, and if not, copies it into the document from a library.

The principles demonstrated in the sample below also apply when working with materials. The only differences being that you'll use the MaterialAssets collection on the library and you'll set the ActiveMaterial property of the PartDocument instead of the appearance property.

```
Public Sub SetOccurrenceAppearance()
    Dim asmDoc As AssemblyDocument
    Set asmDoc = ThisApplication.ActiveDocument

    ' Get an appearance from the document.  To assign an appearance is must
    ' exist in the document.  This looks for a local appearance and if that
    ' fails it copies the appearance from a library to the document.
    Dim localAsset As Asset
    On Error Resume Next
    Set localAsset = asmDoc.Assets.Item("Bamboo")
    If Err Then
        On Error GoTo 0

        ' Failed to get the appearance in the document, so import it.

        ' Get an asset library by name.  Either the displayed name (which
        ' can changed based on the current language) or the internal name
        ' (which is always the same) can be used.
        Dim assetLib As AssetLibrary
        Set assetLib = ThisApplication.AssetLibraries.Item("Autodesk Appearance Library")
        'Set assetLib = ThisApplication.AssetLibraries.Item("314DE259-5443-4621-BFBD-1730C6CC9AE9")

        ' Get an asset in the library.  Again, either the displayed name or the internal
        ' name can be used.
        Dim libAsset As Asset
        Set libAsset = assetLib.AppearanceAssets.Item("Bamboo")
        'Set libAsset = assetLib.AppearanceAssets.Item("ACADGen-082")

        ' Copy the asset locally.
        Set localAsset = libAsset.CopyTo(asmDoc)
    End If
    On Error GoTo 0

    ' Have an occurrence selected.
    Dim occ As ComponentOccurrence
    Set occ = ThisApplication.CommandManager.Pick(kAssemblyOccurrenceFilter, "Select an occurrence.")

    ' Assign the asset to the occurrence.
    occ.appearance = localAsset
End Sub
```

Model States

Introduction to Model State

Model state is introduced to both Part and Assembly documents since Inventor 2022, and the LOD(Level of Detail) in assembly will be removed from UI, the LOD related APIs will be hidden, so all the LOD related funcitons need to be migrated to Model State. In assembly, most of the functions that can work in legacy LOD can also work in model state, only some functions in legacy LOD are changed, and in model state there are some enhancements against legacy LOD(e.g. the file properties can be overriden for each model state). Because of this big change, API developers may need to know how this will impact the API behaviors.

The behavior changes from Level of Detail to Model State

1. The legacy LevelOfDetailRepresentation objects are migrated to ModelState objects. So the FullDocumentName will represent the "FullFileName<ModelState>" instead of legacy "FullFileName<LevelofDetail>".

- Legacy master LevelOfDetailRepresentation object is migrated to primary ModelState

- Legacy custom LevelOfDetailRepresentation objects are migrated to custom ModelState objects.
- Legacy system LevelOfDetailRepresentation objects are removed. When call the Documents.Open/OpenWithOptions with specifying the legacy system LevelOfDetailRepresentation, the document will be opened with primary ModelState active and a proper system design view for it to make use of the visibility indicating the occurrences' suppression status.

Legacy system Level of Detail		Design View in Inventor 2022
All Components Suppressed		Nothing Visible
All Parts Suppressed		Nothing Visible
All Content Center Suppressed		Hide Content Center Components

2. In legacy assembly data the iAssembly and LOD can exist in the same assembly file, while since Inventor 2022 the iAssembly and model states are mutually exclusive.It means ModelState is not supported in iAssembly document, and vice versa. Like ComponentOccurrences.AddiAssemblyMember doesn't support the ModelState in Options argument.

3. Suppress ComponentOccurrenceProxy is not allowed. If a legacy data has a suppressed ComponentOccurrenceProxy then you can only unsuppress it in Inventor 2022 to remove the Suppress(Override) status:

4. Document for Primary and custom ModelState shares the same Document but each substitute has its own Document pointer. That means in legacy Inventor when open assembly document with different LOD specified the returned AssemblyDocument pointers are different, while in Inventor 2022:

- When open document with specifying different primary/custom ModelState, only one AssemblyDocument pointer is returned.
- A substitute’s Document pointer is different from any other ModelState’s Document’s pointer(the same as legacy behavior).

5. OnActivateModelState won't be triggered when activate or deactivate a substitute model state but the OnActivateDocument should be used to monitor this action instead.

6. When open assemly with Primary or custom model state activated, the document(s) referenced by substitute model states will be also included in the Document.ReferencedDocuments/AllReferencedDocuments collection(in Inventor 2022 RTM they won't be included but since Inventor 2022.1 will). But when the subsitute model state is activated the assembly's Document.ReferencedDocuments/AllReferencedDocuments collection will return the referenced document(s) by current substitute document only.

7. Document.File.AllReferencedFiles returns all referenced files for the IAM file including the substitutes and suppressed components.

8. It is required to save dirty assembly before adding a substitute into it, otherwise AddSubstitute will raise error.

9. When use Documents.Open or OpenWithOptions to open one IAM file multiple times with different ModelState specified, it would just switch active ModelState in the same document window instead of displaying in different document windows.

10. For Documents.ItemByName, if model state factory document and model state member document with the same FullDocumentName are both opened in memory it will return model state factory document firstly.

11. The model state factory's Document.FileSaveCounter will increase by 1 for each model state's save, this means if there are multiple model states require to save, when save the model state factory document may cause the FileSaveCounter to increase by a number that is bigger than 1. This also applies to File or FileDescriptor.FileSaveCounter. The model state member's Document.FileSaveCounter will only record the model state member document's save count.

New Model State APIs in Inventor 2022

1. PartComponentDefinition.ModelStates/AssemblyComponentDefinition.ModelStates: Access Model States.

2. AssemblyComponentDefinition/PartComponentDefinition.IsModelStateFactory/IsModelStateMember : Determine whether the document is model state factory or member document. If the document is a model state member document, ComponentDefinition/ModelState.FactoryDocument can be used to get the relative model state factory document.

3. ComponentOccurrence.ActiveModelState:Switch ComponentOccurrence model state. It is required that ComponentOccurrence.Definition.Document is up-to-date; otherwise, Save dialog would pop up to ask for document save, a failure will occur if user chooses not to save.

4. ComponentOccurrence.Replace/Replace2 will use the last active model state by default.

5. BOMView.ModelStateMemberName returns the model state member name that current BOMView is based on.

6. PropertySets edit is impacted by ModelStates.MemberEditScope.

If MemberEditScope is set to kEditActiveMember, editting Property’s value for active ModelState just overrides the value for the active ModelState; otherwise the value will be applied to the Property for all ModelState. The Document.FilePropertySets represents the iProperties in Shell Extension for Part/Assembly documents.

7. AddBendTableWithOptions:Support to add bend table with specified model state.

8. AddConfigurationTable:Support to create table with specifying model states.

9. DocumentDescriptor.ReferencedModelState:Tell the model state type of the DocumentDescriptor.

10. ModelingEvents.OnGenerateModelStateMember:To monitor when the model state member is generated.

11. FileOpenOptions.DefaultModelStateInAssembly&DefaultModelStateInPart:Set default model state when opening assembly or part document.

12. In Apprentice: ApprenticeServerComponent.Open can open part or assembly with specifying model state in FullDocumentName, but it always opens the model state factory document. ApprenticeServerDocument.ComponentDefinition.ModelStates is not accessible at present. FileManager.GetModelStates/GetModelStateName/GetLastActiveModelState are supported in Apprentice.

13. For custom addin: Environment.PreserveWhenSwitchModelState allows addin to preserve its environment when switch model states. For built-in addin this API will always return True to keep legacy behavior.

Equivalent APIs(Level of Detail vs Model State)

Level of Detail APIs	Model State APIs
FullDocumentName (e.g. “C:\Assembly1.iam<Level of Detail1>”)	FullDocumentName (e.g. “C:\Assembly1.iam<Model State1>”)
ComponentOccurrence. ActiveLevelOfDetailRepresentation	ComponentOccurrence.ActiveModelState
DerivedAssemblyDefinition. ActiveLevelOfDetailRepresentation	DerivedAssemblyDefinition.ActiveModelState
	DerivedPartDefinition.ActiveModelState
ShrinkwrapDefinition.ActiveLevelOfDetailRepresentation	ShrinkwrapDefinition.ActiveModelState
AssemblyComponentDefinition. RepresentationsManager. LevelOfDetailRepresentations	AssemblyComponentDefinition.ModelStates
	PartComponentDefinition.ModelStates
RepresentationsManager.ActiveLevelOfDetailRepresentation	ModelStates.ActiveModelState
FileOpenOptions.DefaultLevelOfDetail	FileOpenOptions.DefaultModelStateInAssembly
	FileOpenOptions.DefaultModelStateInPart
	SaveOptions.DefaultToSaveForModelStateUpdates
FileManager.GetLastActiveLevelOfDetailRepresentation	FileManager.GetLastActiveModelState
FileManager.GetLevelOfDetailName	FileManager.GetModelStateName
FileManager.GetLevelOfDetailRepresentations	FileManager.GetModelStates
	AssemblyComponentDefinition.IsModelStateFactory
	AssemblyComponentDefinition.IsModelStateMember
	AssemblyComponentDefinition.FactoryDocument
	PartComponentDefinition.IsModelStateFactory
	PartComponentDefinition.IsModelStateMember
	PartComponentDefinition.FactoryDocument
LevelOfDetailEnum	ModelStateTypeEnum
RepresentationEvents	ModelStateEvents
RepresentationEvents.OnActivateLevelOfDetailRepresentation	ModelStateEvents.OnActivateModelState
RepresentationEvents.OnDelete	ModelStateEvents.OnDeleteModelState
RepresentationEvents.OnNewLevelOfDetailRepresentation	ModelStateEvents.OnNewModelState
	BOMView.ModelStateMemberName
AssemblyDocument.LevelOfDetailName	AssemblyDocument.ModelStateName
	PartDocument.ModelStateName
	ModelingEvents.OnGenerateModelStateMember
DocumentDescriptor.ReferencedLevelOfDetail	DocumentDescriptor.ReferencedModelState
DrawingView.ActiveLevelOfDetailRepresentation	DrawingView.ActiveModelState & SetActiveModelState
DesignViewRepresentation.CopyToLevelOfDetail	DesignViewRepresentation. CopyComponentVisibilityToSuppression
	ModelState.CopyComponentSuppressionToVisibility
BrowserNode(Level of Detail root).NativeObject = LevelOfDetailRepresentation	BrowserNode(ModelStates).NativeObject = ModelState

Working with Model State API

Get model state names in a document. Place an Assembly1.iam to C:\Temp folder firstly:

```
Sub GetModelStateNames()
    Dim sFileName As String
    sFileName = "C:\Temp\Assembly1.iam"

    Dim oDoc As AssemblyDocument
    Set oDoc = ThisApplication.Documents.Open(sFileName)

    Dim oAssemblyCompDef As AssemblyComponentDefinition
    Set oAssemblyCompDef = oDoc.ComponentDefinition

    ' Get ModelStates collection object.
    Dim oModelStates As ModelStates
    Set oModelStates = oAssemblyCompDef.ModelStates

    ' Get all model state names from ModelStsates object.
    Dim oModelState As ModelState
    For Each oModelState In oModelStates
        Debug.Print oModelState.Name
    Next

    oDoc.Close True

    ' Get all model state name from FileManager
    Dim oFileManager As FileManager
    Set oFileManager = ThisApplication.FileManager

    Dim sModelStateNames() As String
    sModelStateNames = oFileManager.GetModelStates(sFileName)

    Dim sName As Variant
    For Each sName In sModelStateNames
        Debug.Print sName
    Next

End Sub
```

Switch model state for a ComponentOccurrence. Open an assembly with occurrences(not virtual component):

```
Sub SwitchActiveModelStateOfOccurrence()
    Dim oDoc As AssemblyDocument
    Set oDoc = ThisApplication.ActiveDocument

    Dim oAssemblyCompDef As AssemblyComponentDefinition
```

```
    Set oAssemblyCompDef = oDoc.ComponentDefinition

    ' Specify an occurrence to change its active model state.
    Dim oOccu As ComponentOccurrence
    Set oOccu = oAssemblyCompDef.Occurrences(1)

    ' Get native document ComponentDefinition of occurrence.
    Dim oOccuDef As ComponentDefinition
    Set oOccuDef = oOccu.Definition

    ' Before switch the active model state of an occurrence you should check if the document is up to date.
    If oOccuDef.Document.RequiresUpdate Then
        MsgBox("Can not switch active model state because the native document is not up to date!")
    Else
        Dim oModelStates As ModelStates
        Set oModelStates = oOccuDef.ModelStates

        Dim oModelState As ModelState
        For Each oModelState In oModelStates
            If oModelState.Name <> oOccu.ActiveModelState Then

                oOccu.ActiveModelState = oModelState.Name
                Exit Sub
            End If
        Next
    End If
End Sub
```

More info about model state

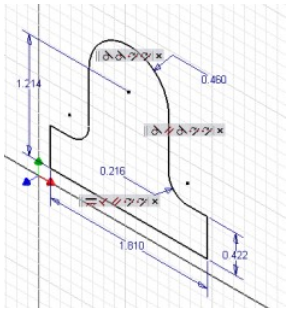
As each substitute in an assembly has its own Document pointer, when query each and every occurrences in an .iam file you should query the occurrences from Primary model state and also iterate each and every substitue to get their occurrences. Also when deal with attributes and reference keys each substitute has its own document context, so when use AttributeManager or ReferenceKeyManager you should consider the Document for each substitute but not only the Primary model state.

Model state member document is generated(if not yet) when a ModelState is specified to create a ComponentOccurrence, DrawingView, DerivedPart and DerivedAssembly etc., a model state member document does not allow to edit its model contents like create, edit or delete features, sketches etc., the changes can be done in model state factory document and then update to model state member document.

Sketches

Introduction

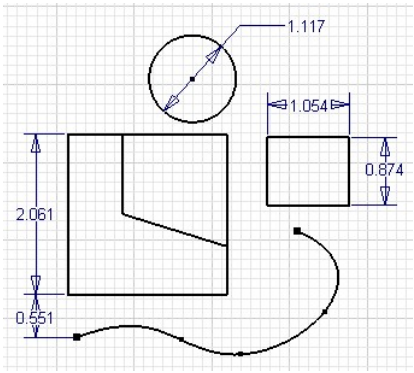
Sketches are an extremely important component of Inventor. All 2D work in Autodesk Inventor is performed using sketches. Sketches are used to draw geometry that will define the profile of a feature. Sketches are also used within drawings to design borders and title blocks, and when drawing on a sheet or within a view. All 2D geometry in Autodesk Inventor is created using sketches.



A sketch can be thought of as a 2D plane that contains any number of 2D entities. In a 3D environment like the part environment, the 2D sketch plane is positioned within 3D space. Because a sketch is 2D, the sketch entities it contains are true 2D objects. That is, any coordinates defined for 2D sketch entities need only their X and Y coordinates defined. Because a sketch exists within 3D space, it has its own 2D coordinate system and all creation and query of entities within a sketch are with respect to the sketch coordinate system.

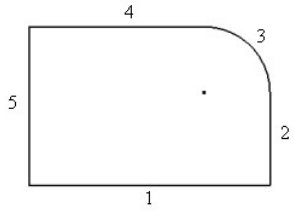
Sketches can contain several different kinds of 2D entities, depending on the type of document they are used in. In Part documents, you can create sketches that contain arcs, circles, elliptical arcs, ellipses, lines, splines, geometric constraints, and dimension constraints. Methods for accessing and creating sketches in Part documents (called PlanarSketches) are discussed in Sketches in Part Documents. Sketches in drawings support the same entities as part sketches, but in addition also help provide access to text and have some special requirements with regard to editing. Methods for accessing and creating sketches in drawing documents are discussed in Sketches in Drawing Documents.

A sketch can be thought of as a container for 2D entities. The sketch itself does not add any information to the 2D entities. For example, look at the sketch shown below. It contains several different types of sketch entities: lines, circles, splines, and dimension and geometric constraints. The sketch defines only the 2D coordinate system for these entities. The fact that there are multiple outlines and some are connected and some aren't isn't information that the sketch is concerned with. That type of information is defined by profiles, which are discussed in more detail later.



Some sketch concepts are slightly different when using the API than when using sketches interactively. The API exposes the sketches at a lower level than is exposed through the user interface. Most of the sketch concepts learned when using Autodesk Inventor interactively carry over to the API, but there are some differences that need to be understood to make using the sketch portion of the API more intuitive.

One difference between the API and interactive use of sketches is in their creation. When interactively defining a sketch, you are able to gain a significant amount of control of the placement because of the automatic inference Autodesk Inventor does as you sketch. As an example, let's look at the simple sketch shown below. To create the sketch the only inputs required from you are a click to start line 1, a click to create line 1, a click to create line 2, a drag to create arc 3, a click to create line 4, and finally a click to create line 5. All that was required were four clicks and a drag to define the entire profile. Let's look in a little more detail at what actually happened during this process.

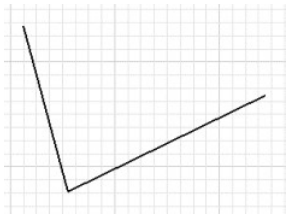


- To create line 1, you click the mouse to define the starting point. As you move the mouse, Autodesk Inventor displays a glyph to indicate when the line will be inferred to be horizontal or vertical. After clicking the mouse to define the second point, a horizontal constraint is automatically applied.
- Since you're still in the same sequence within the Line command, ending line 1 has started line 2. A coincidence between them has been inferred. As you move the mouse Autodesk Inventor will indicate when line 2 is begin inferred to be perpendicular to line 1. When the mouse is clicked to define line 2, the coincident and perpendicular constraints are automatically applied.
- Arc 3 is created by dragging the mouse off the end of line 2. A coincidence and tangency to line 2 is inferred. After line 3 is created the coincident and tangent constraints are automatically placed.
- Line 4 is started by finishing arc 3. Using the glyphs, Autodesk Inventor notifies you when the line is tangent to the arc and parallel to line 1. After clicking to define line 4, the coincident, tangent, and parallel constraints are automatically placed.
- Line 5 is started by finishing line 4. As you move over the end of line 1, a glyph is displayed telling you you're over the end. Clicking over the end indicates that you want the lines connected. A glyph is also displayed showing that a parallel constraint to line 2 is being inferred. After clicking, the coincident constraints to lines 1 and 4 are automatically placed and the parallel constraint to line 2 is automatically placed.

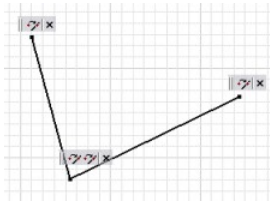
Four clicks and a mouse drag resulted in the creation of 5 sketch entities and 11 constraints.

Now let's look at the same construction from the point of view of the API. There are several differences between interactively creating sketches and creating them through the API. First, there is no inference of constraints when using the API. Everything must be explicitly defined when using the API. Inference is entirely based on the context of the command and how the user moves the mouse during the command. The API methods do not have this type of context information.

Another significant difference between the API view and the user interface view is that the API exposes some of the underlying details of the sketch that are hidden in the user interface. The primary feature being hidden is that all sketch geometry is actually dependent on sketch points. For example, when sketching two lines interactively, you click the mouse twice to define the two points for the first line and click again to define the second line.

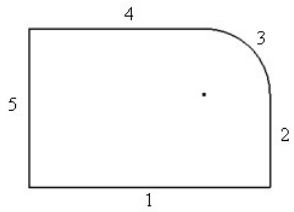


The result to the end-user is two lines that have a coincident constraint holding them together. The actual result is that you have created three sketch points, two lines, and four coincident constraints. Sketch points are the only sketch entities that can exist on their own. All other types of sketch entities are dependent on points to define their position. The sketch entities are tied to the points by coincident constraints. The picture below illustrates the true situation.



To simplify working with sketches, the user interface hides most of these sketch points. Attempting to hide these points in the API would have resulted in many inconsistencies, and as you'll see having access to these points simplifies many API functions.

Let's look at creating the original example using the API to see the differences.



If you look at the online help for the AddByTwoPoints method of the SketchLines object you'll see the method takes two points to define the line.

```
AddByTwoPoints(StartPoint As Object, EndPoint As Object) As SketchLine
```

Notice that the type of object expected as input for the points is defined as "Object." This allows you to input either a Point2d object or an existing SketchPoint object. A Point2d object describes a location on the sketch. When you provide a Point2d object as input a SketchPoint is created at that location and the line is attached to the sketch point with a coincident constraint. In the case where a SketchPoint is input, the line is attached directly to the input sketch point using a coincident constraint.

Because sketch entities are always dependent on sketch points, the API provides direct access to the driving sketch points. To see how this works, let's look at the code that will create the previous sketch. This sample assumes that a sketch has somehow already been obtained. (This will be discussed in more detail in the next section.)

When you define the location of sketch entities using coordinates on the sketch, they are defined using Point2d objects. These objects don't define a graphical point but only a coordinate in 2D space. These lines set a reference to the TransientGeometry object to facilitate its use later in the program.

```
Dim oTransGeom As TransientGeometry
Set oTransGeom = ThisApplication.TransientGeometry
```

This defines two Point2d objects that will be used to define the ends of the line.

```
Dim oCoord1 As Point2d
Set oCoord1 = oTransGeom.CreatePoint2d(0, 0)
Dim oCoord2 As Point2d
Set oCoord2 = oTransGeom.CreatePoint2d(5, 0)
```

This creates the actual SketchLine object. The two inputs are the two Point2d objects. Automatically, two sketch points are created at those locations and the line is attached to them with coincident constraints.

```
Dim oLines(1 To 4) As SketchLine
Set oLines(1) = oSketch.SketchLines.AddByTwoPoints(oCoord1, oCoord2)
```

This creates the second sketch line. In this case a SketchPoint and a Point2d object are input as the start and end points of the line. The SketchPoint is obtained by asking the first line for its end sketch point.

```
Set oCoord1 = oTransGeom.CreatePoint2d(5, 3)
Set oLines(2) = oSketch.SketchLines.AddByTwoPoints(oLines(1).EndSketchPoint, _
                                                   oCoord1)
```

This creates the arc. The center and end point are defined using Point2d objects and the start point is defined using the end sketch point from the previous line.

```
Set oCoord1 = oTransGeom.CreatePoint2d(4, 3)
Set oCoord2 = oTransGeom.CreatePoint2d(4, 4)
Dim oArc As SketchArc
Set oArc = oSketch.SketchArcs.AddByCenterStartEndPoint(oCoord1, _
                                          oLines(2).EndSketchPoint, oCoord2)
```

This creates the third line and connects it to the arc by inputting the arcs end sketch point.

```
Set oCoord1 = oTransGeom.CreatePoint2d(0, 4)
Set oLines(3) = oSketch.SketchLines.AddByTwoPoints(oArc.EndSketchPoint, _
                                                   oCoord1)
```

This creates the final line using sketch points from the first and last line as inputs.

```
Set oLines(4) = oSketch.SketchLines.AddByTwoPoints( _
                                        oLines(1).StartSketchPoint, _
                                        oLines(3).EndSketchPoint)
```

At this point you will have a sketch that looks similar to that shown in the previous figure, but if you attempt to manipulate the sketch at all you will notice that the only constraints are the coincident constraints tying everything together. Because of this, the lines will not stay horizontal and vertical, and the lines will not remain tangent to the arc. The following code will add the same constraints that were placed automatically when interactively creating the sketch.

```
Call oSketch.GeometricConstraints.AddHorizontal(oLines(1))
Call oSketch.GeometricConstraints.AddPerpendicular(oLines(1), oLines(2))
Call oSketch.GeometricConstraints.AddTangent(oLines(2), oArc)
Call oSketch.GeometricConstraints.AddTangent(oLines(3), oArc)
Call oSketch.GeometricConstraints.AddParallel(oLines(1), oLines(3))
Call oSketch.GeometricConstraints.AddParallel(oLines(4), oLines(2))
```

If you edit the sketch now it will have the behavior you expect.

All other types of sketch entities are placed in a similar way. From this you can see that sketch entities are dependent on sketch points. When creating sketch entities you can define coordinate points, but sketch points will automatically be created for the entity to be tied to. When modifying sketch entities, most modifications must be performed on the sketch points they're tied to. For example, to move a line you need to move the two sketch points it is tied to.

You can also see that when creating sketch entities through the API there is no constraint inference (except for the coincident constraint placement when you use a sketch point as input). When creating sketches using the API you'll need to explicitly create the geometric constraints.

The API also exposes additional constraint types that don't have corresponding commands. You implicitly create these constraints as the result of using some commands. For example, the offset constraint is created as a result of using the Offset command. Here's a complete list of all of the geometric constraints exposed through the API. (More information can be found in the API reference guide.)

CoincidentConstraint
: Ties a sketch point to any other sketch entity.

CollinearConstraint
: Makes a line or the specified axis of an ellipse collinear to another line or ellipse axis.

ConcentricConstraint
: Makes a circle, arc, ellipse, or elliptical arc concentric to another circle, arc, ellipse, or elliptical arc.

EqualLengthConstraint
: Makes two lines equal in length.

EqualRadiusConstraint
: Makes the radius of a circle or arc equal to another circle or arc.

GroundConstraint
: Causes any sketch entity to be fully constrained.

HorizontalAlignConstraint
: Makes two sketch points align along the same horizontal axis. In other words, the two sketch points will have the same Y coordinate value.

HorizontalConstraint
: Causes a lines or the specified axis of an ellipse to be horizontal.

MidpointConstraint
: Causes a sketch point to be positioned at the midpoint of a line.

OffsetConstraint
: Makes four entities behave in a way where two are the result of offsetting from the other two.

ParallelConstraint
: Makes a line or the specified axis of an ellipse parallel to another line or ellipse axis.

PatternConstraint
: Defines the relationship between entities that were the result of creating a pattern.

PerpendicularConstraint
: Makes a line or the specified axis of an ellipse perpendicular to another line or ellipse axis.

SplineFitPointConstraint
: Defines the connection between a spline and the sketch points it is tied to.

SymmetryConstraint
: Causes two sketch entities of the same type to be symmetric about a line.

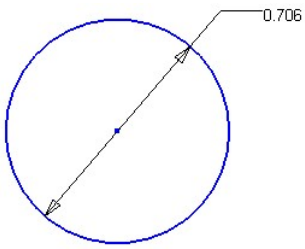
TangentSketchConstraint
: Causes two sketch entities to be tangent to each other.

VerticalAlignConstraint
: Makes two sketch points align along the same vertical axis. In other words, the two sketch points will have the same X coordinate value.

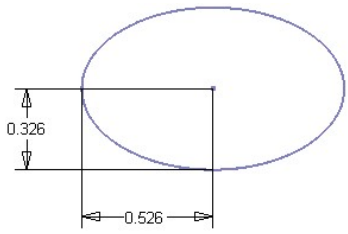
VerticalConstraint
: Causes a lines or the specified axis of an ellipse to be vertical.

The list below illustrates the entire set of dimension constraints exposed by the API.

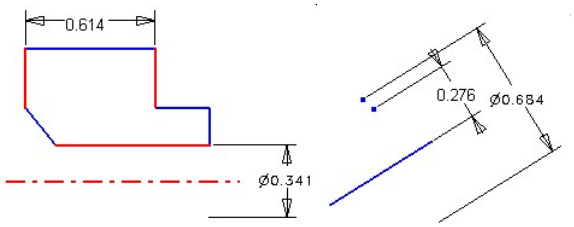
DiameterDimConstraint



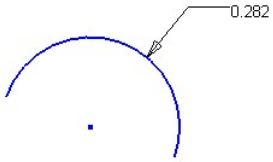
EllipseRadiusDimConstraint



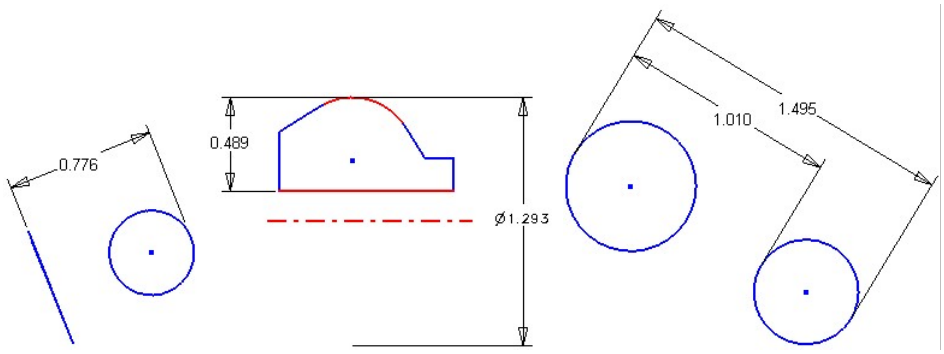
OffsetDimConstraint



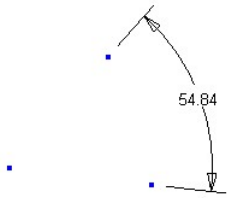
RadiusDimConstraint



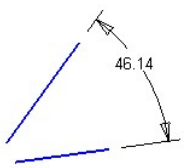
TangentDistanceDimConstraint



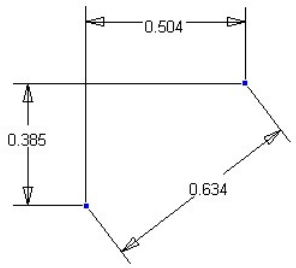
ThreePointAngleDimConstraint



TwoLineAngleDimConstraint



TwoPointDistanceDimConstraint



Sketches in Part Documents

Sketches can contain several different kinds of entities, depending on the type of document they are used in. In Part documents, you can create sketches that contain arcs, circles, elliptical arcs, ellipses, lines, splines, geometric constraints, and dimension constraints. The previous example began by assuming you had obtained a sketch somehow. This section will discuss some methods of obtaining existing sketches and creating new sketches.

Many programs that deal exclusively with sketches will not want to create a sketch but instead will want to access the currently active sketch. In the case where a sketch is not active you will need to know this so you can handle it gracefully. Determining if a sketch is active can be accomplished using the ActiveEditObject property of the Application. This property returns the object that is currently active for edit. Currently this can be any of the various document types or a sketch. The sample code below illustrates checking to see if a sketch is active and if one is active, setting a reference to it. If a sketch isn't active, it displays a message telling the user a sketch must be active.

```
' Determine if a sketch is active.
If Not TypeOf ThisApplication.ActiveEditObject Is Sketch Then
    MsgBox "A sketch must be active."
    Exit Sub
End If

' Set a reference to the active sketch.
Dim oSketch As Sketch
Set oSketch = ThisApplication.ActiveEditObject
```

The Sketches collection object also provides access to all of the existing sketches in a document. Through this you can obtain any existing sketch, and if you know its name you can access it directly using the Item method of the Sketches collection object. The code below iterates through all of the sketches in the document and prints their names.

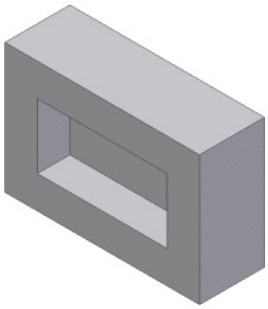
```
Dim oSketch As Sketch
For Each oSketch in oPartDoc.ComponentDefinition.Sketches
    Debug.Print "Sketch: " & oSketch.Name
Next
```

This code sets a reference to the sketch named "Sketch2." If a sketch named "Sketch2" does not exist in the part the call of the Item property will fail. The On Error statement allows you to check for and handle this error.

```
' Enable error trapping
On Error Resume Next

Dim oSketch As Sketch
Set oSketch in oPartDoc.ComponentDefinition.Sketches.Item("Sketch2")
If Err Then
    Err.Clear
    MsgBox "A sketch named ""Sketch2"" does not exist."
End If

' Turn off error trapping
On Error Goto 0
```

When creating models you'll usually need to create sketches. The Sketches collection supports two methods for creating a sketch. The Add method works identically as the 2D Sketch command. That is, you provide a planar entity, (a work plane or planar face) as input and the sketch is created. It uses built-in logic to determine the orientation of the sketch on the selected entity. Sometimes this is acceptable, but unlike the end-user who is working visually with the system and can easily see and react to the default orientation, you usually need explicit control of the orientation of the sketch. The AddWithOrientation method provides this control.

Let's look at an example to see why controlling the orientation can be important. This will also help to explain the concept of drawing on a 2D plane that is positioned within 3D space. In this case, we want to create the part shown below.

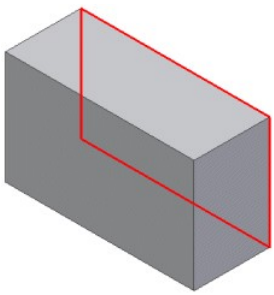


The part consists of two extrusion features: one to create the base block and another to create the pocket.

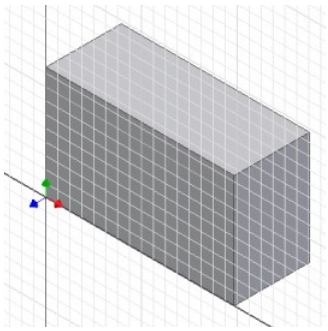
The code below creates the first extrusion. (The profile creation step is discussed in detail in a subsequent section.)

```
' Set a reference to the component definition.
Dim oPartCompDef As PartComponentDefinition
Set oPartCompDef = ThisApplication.ActiveDocument.ComponentDefinition

' Create a new sketch.
Dim oSketch As Sketch
Set oSketch = oPartCompDef.Sketches.Add(oPartCompDef.WorkPlanes.Item(3))

' Draw a rectangle.
With ThisApplication.TransientGeometry
    Call oSketch.SketchLines.AddAsTwoPointRectangle( _
                                        .CreatePoint2d(0, 0), _
                                        .CreatePoint2d(5, 3))
End With

' Create a profile.
Dim oProfile As Profile
Set oProfile = oSketch.Profiles.AddForSolid

' Create the extrusion feature.
Dim oExtrude As ExtrudeFeature
Set oExtrude = oPartCompDef.Features.ExtrudeFeatures.AddByDistanceExtent( _
                   oProfile, 2, kPositiveExtentDirection, kJoinOperation)
```

To create the sketch for this feature, the Sketches.Add method is used. This is the method that doesn't provide control over the orientation. You'll notice that the input plane provided is an existing work plane. The first three work planes in the WorkPlanes collection are the three that exist in every part. The first represents the YZ plane, the second the XZ plane, and the third the XY plane. (This is the same order as they appear in the Browser.) When a sketch is created with the Add method and a work plane is used as input, the sketch inherits the orientation and origin from the work plane. In this case, since it's the XY base work plane the origin will be at (0,0,0) and the X and Y directions of the sketch will be in the same directions as the X and Y axes of the model. The result after the creation of this feature is shown below, and the sketch used is highlighted.



The creation of the next sketch becomes more interesting. It's important to control its orientation and origin so the coordinates used to define the position of the rectangle will be in the correct position. In this case, we want to create a sketch that will be positioned as shown below.



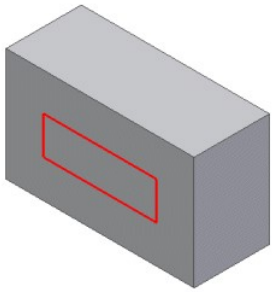
The following code creates a sketch in the desired position and uses the sketch to create the extrusion for the pocket.

```
' Create a sketch on the end face using an existing work axis
' and the origin work point.
Set oSketch = oPartCompDef.Sketches.AddWithOrientation( _
                        oExtrude.EndFaces.Item(1), _
                        oPartCompDef.WorkAxes.Item(1), True, True, _
                        oPartCompDef.WorkPoints.Item(1), False)

' Draw a rectangle.
With ThisApplication.TransientGeometry
    Call oSketch.SketchLines.AddAsTwoPointRectangle( .CreatePoint2d(1, 1),
                                                     .CreatePoint2d(4, 2))
End With

' Create a profile.
Set oProfile = oSketch.Profiles.AddForSolid

' Create the extrusion feature.
Set oExtrude = oPartCompDef.Features.ExtrudeFeatures.AddByDistanceExtent( _
                   oProfile, 0.75, kNegativeExtentDirection, kCutOperation)
```

The interesting portion of this code sample is the AddWithOrientation method call. This allows you to create a sketch and fully define its orientation. The first argument is the face, which is obtained directly from the previous feature. The second argument is used to the define the X axis. In this case the system X work axis is provided as input. The third argument specifies if the direction of the sketch X axis should be in the same direction as the entity provided as the second argument. The

fourth argument specifies if the axis being defined is the X or Y axis. The fifth argument defines the origin of the sketch. In this sample the system origin work point is input. The final argument defines whether the edges of the input face should be copied onto the sketch, (as is done when creating a sketch interactively).

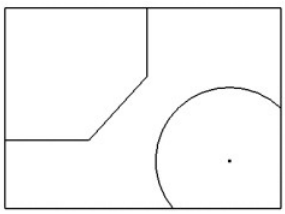
By explicitly defining the sketch you now know the correct coordinates to input to create the rectangle so it is oriented correctly, as is shown below.



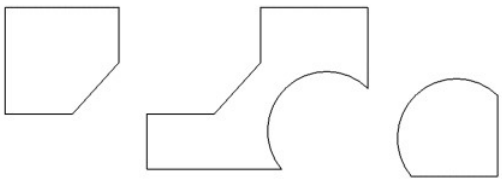
To help in diagnosing problems where sketch entities are not being drawn where expected, you can view the origin and orientation of an existing sketch by editing the sketch interactively. If the display setting for the "Coordinate System Indicator" is enabled on the Sketch tab of the "Application Options" dialog, a triad will be displayed when the sketch is being edited. The triad is positioned at the sketch's origin and is oriented to show the X and Y axes of the sketch.

Profiles in Part Documents

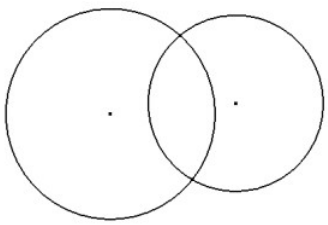
Another important concept that's not explicitly exposed through the user interface is the concept of profiles. A sketch is essentially just a container for 2D entities and their associated constraints. The sketch itself does not define connected sets of entities that can be used by a feature to define its shape. This information is defined by profiles. You can create the following sketch in Autodesk Inventor.



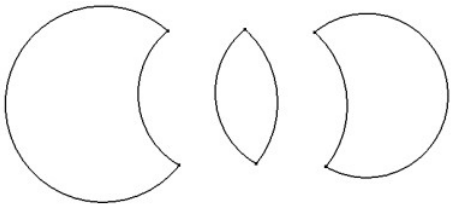
When creating an extrude feature, the first step is to define the profile. This is done interactively by moving the mouse within the different closed areas and clicking. Any combination of the following three shapes can be used to define the profile of the feature.



In the API, the profile is an explicit object that you can create to provide input to features and can also obtain from existing features. A profile is always associated with a particular sketch and essentially adds topological information to the sketch. Coincident constraints between sketch geometry controls which shapes can be defined from a given sketch. For example, if you create a sketch containing the two circles shown in the following figure and use this sketch for an extrusion, you can only select the inside of the circles.



However, if you place sketch points at the intersections of the circles and tie the points to the circles using coincident constraints, you can now achieve any combination of the following three shapes.



The API supports two methods for creating profiles. These are AddForSolid, which was used in the previous sample code, and AddForSurface. These two methods simplify the creation of the profiles for the developer. Generally, when creating profiles programmatically, it is easiest to define the final result. An end user may find it more convenient to extract existing geometry, perform trimming, edit dimension values, and other similar operations to arrive at the desired result. Also, when creating sketches interactively, the edges of a selected face are automatically projected onto the profile. This results in extra geometry that you may not end up using. In a workflow where additional geometry is created, designers need more control over how profiles are created so they can define which geometry to use in the creation of the profile. Developers typically doesn't require any intermediate geometry, but create the correct geometry to begin with.

The two profile creation methods provided by the API have optional arguments. Called with no arguments, they examine all of the geometry in the sketch and create the profile. The AddForSolid method checks for closed profiles that can be used in creating solid features. The AddForSurface allows open profiles that can be used for creating surfaces.

Optional Arguments

The AddForSolid method supports a Curves argument. This argument is an ObjectCollection of sketch curves. If supplied, this method creates a profile consisting of only those paths that contain all the sketch curves contained in the object collection. If not specified, all the possible profile paths are included.

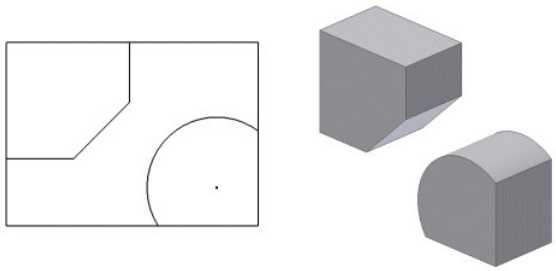
The AddForSurface method supports a Curve argument. This argument is a sketch curve. If supplied, this method creates a profile consisting of a path that contains all sketch curves that are connected to the input curve. If no other curve is connected to the input curve, the path contains just the input curve. If not supplied, the method creates a profile by examining the contents of the sketch and creating a single connected path. The result can be either open or closed.

The AddForSolid method supports a Combine argument. This specifies whether to combine the profile paths obtained when this method is used to create a new profile. For instance, take the example of a sketch containing two concentric circles. If this argument is specified to be true, the resulting profile will contain two profile paths, and the profile path corresponding to the inner circle will have its AddsMaterial flag set to false. Hence, the resulting profile is a circular disc with a circular cut-out. If the Combine flag is specified to be false, the resulting profile will contain 2 profile paths with the AddsMaterial flag set to true for both paths.

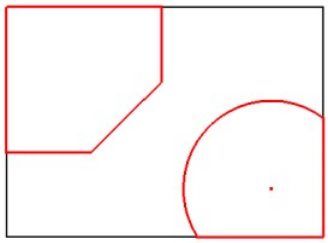
The AddsMaterial flag indicates whether the profile path causes material to be added or subtracted from the area defined by the profile. If specified to be true, the profile path adds material. A feature such as an extrusion would use the profile path.

Profiles versus sketches

Profiles are used as input when creating features and are also important when querying existing features. From sketch-based features you can obtain the profile that defines its shape. Just having a sketch isn't enough because it's the profile that adds topology information to a sketch. Let's look at the following example to see what information a profile provides. The sketch shown on the left was used to create the single extrusion feature shown on the right.

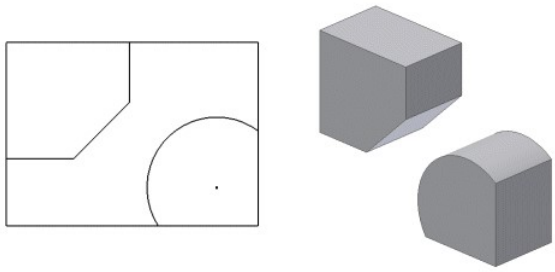


The profile obtained from the feature contains two ProfilePath objects, which are highlighted in the following figure. Each ProfilePath object returns a list of ProfileEntity objects. The ProfilePath returns the ProfileEntity objects in the order of how they are connected to each another. The ProfilePath also tells you if the path is open or closed and if it adds or removes material from the profile.

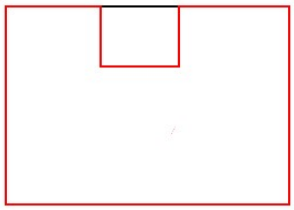


The ProfileEntity objects provide information about each piece of a ProfilePath. The ProfileEntity objects are oriented so they are connected head-to-tail to one another. ProfileEntity objects don't exist graphically within Autodesk Inventor but are simply an intermediate step between the sketch and the feature. A ProfileEntity returns the underlying sketch entity and the two sketch points that define its start and end. The underlying sketch entity will in many cases be different from the profile entity.

Let's look in detail at the previous example. The sketch that was used contains seven lines and one arc, plus all of the sketch points and various constraints tying it all together. From this sketch a single profile was created and used as input for the extrude feature. The profile consists of two ProfilePath objects, which when used for the extrusion resulted in a single solid with two enclosed volumes, as shown in the following image.

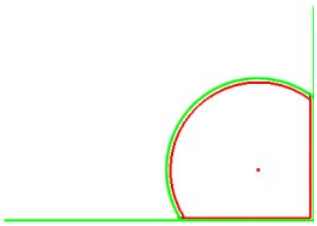


The preceding profiles are straightforward, but others are less so. Take the following sketch as an example - calling AddForSolid with no arguments would not necessarily create the desired profile paths. Using the combine argument and/or the delete method of the ProfilePath object allows a greater degree of control over what constitutes the profile.



Note: When editing sketches through the user interface, the user enters sketch edit mode. Leaving this mode automatically causes model features to be recomputed. Through the API, sketches can be modified at any time. Therefore the PlanarSketch object has a method, UpdateProfiles, that causes model features to be recomputed.

A ProfilePath consists of a set of ProfileEntity objects. For example, the profile path highlighted in the following figure consists of three ProfileEntity objects. A ProfileEntity object provides access to its underlying sketch entity and also a transient geometry object that represents the actual geometry of the ProfileEntity. This is frequently different from the geometry of the underlying sketch entity for two reasons. First, the profile entity may consist of just a portion of the sketch entity. The following figure illustrates one of the profile's paths (red) and the underlying sketch entities (green). The two profile entities that represent the lines are shorter than the sketch lines they're based on. The ProfileEntity also returns the two sketch points that define its extent along the sketch entity.



The second reason the geometry of the profile entity might be different from the geometry of the sketch entity is that the profile entities are returned such that they're connected head to tail. There is no implied direction to sketch entities. Because of this the profile entity might go in the opposite direction from the underlying sketch entity.

Sketches in Drawing Documents

There are two principal uses for sketches in drawing documents. The first are overlay sketches that are placed on the sheet or attached to a drawing view. The second are sketches used in the definition of drawing aid objects, such as border definitions, title block definitions, and sketched symbol definitions. There are subtle behavioral differences between these two sketch uses, and between sketches in drawing documents versus those in part documents.

An important difference between sketches in drawing documents versus those in part documents is that drawing sketches must be placed into an edit mode before changes are allowed to the sketch contents. This edit mode is equivalent to the sketch environment being activated in the user interface.

For sheet overlay and drawing view sketches, this is done by calling the Edit method on the DrawingSketch object. To leave edit mode, call the ExitEdit method. The code below places the first sheet overlay sketch on the active sheet in edit mode.

```
' Determine if there are any sheet overlay sketches.
Dim oSketches As DrawingSketches
Set oSketches = ThisDocument.ActiveSheet.Sketches
If oSketches.Count = 0 Then
    MsgBox "Active sheet does not contain any overlay sketches."
    Exit Sub
End If

' Set a reference to the first sketch.
Dim oSketch As DrawingSketch
Set oSketch = oSketches.Item(1)

' Place the sketch in edit mode.
oSketch.Edit

' Make changes to the sketch contents here.
```

```
' Return from edit mode.
oSketch.ExitEdit
```

To edit the sketch of a drawing aid definition, the definition object must be placed into edit mode which is done by calling the Edit method on the definition object (BorderDefinition, TitleBlockDefinition, or SketchedSymbolDefinition), not the DrawingSketch. Calling the Edit or ExitEdit method on the DrawingSketch owned by a drawing aid definition (obtained from the Sketch property on, for example, the BorderDefinition object) will always fail. The Edit method and drawing aid definitions has a single out argument which is the DrawingSketch that is in edit mode. Note that this is not a reference to the same sketch obtained from the drawing aid's Sketch property, but is a temporary copy of the definition's sketch on a temporary sheet that is activated for the edit operation. The code below places a border in edit mode.

```
' Get the desired border.
Dim oBorder As BorderDefinition
On Error Resume Next
Set oBorder = ThisDocument.BorderDefinitions.Item("MyBorder")
If Err Then
    Err.Clear
    MsgBox "A border named ""MyBorder"" does not exist."
    Exit Sub
End If
On Error GoTo 0

' Place the border in edit mode.
Dim oSketch As DrawingSketch
oBorder.Edit oSketch

' Make changes to the sketch contents here.

' Return from edit mode.
oBorder.ExitEdit
```

Because only one drawing sketch can be actively editing at a time, calling the Edit method on an overlay sketch or a drawing aid definition will fail if there already is a drawing sketch in edit mode. To determine if a sketch is in edit mode, or to obtain the currently editing sketch, use the ActiveEditObject property on the Application object, or the ActivatedObject property on the DrawingDocument object. Note that if you attempt to transparently change the actively editing sketch when the active editing sketch is for a drawing aid definition, the changes must be aborted, saved, or saved to a new definition. In order to exit sketch mode for a drawing aid, you have to decide whether to abort, save or save changes to a new definition, which makes it impossible to return the user to the previous state.

In order to create a new overlay sketch or drawing aid definition, the new sketch needs to be temporarily placed into sketch edit mode. This means that an attempt to create a new overlay sketch or drawing aid definition will fail if there is a sketch active in edit mode.

When a new overlay sketch is created through the user interface, the sketch being constructed is not yet added to the sheet. There is no browser entry for the new sketch and the sketch will not show up in the sheet or drawing view's sketches collection until the sketch is returned from edit mode, at which point it is added to the sheet. Overlay sketches in this user interface construction state are accessible through the ActiveEditObject property of the Application object or the ActivatedObject property of the DrawingDocument object.

When a new drawing aid definition object is created through the user interface, the definition object is not yet added to the appropriate drawing aid definition object collection. There is no browser entry for the definition and it will not show up in the appropriate drawing aid definition object collection until the sketch edit mode is exited and the changes saved, at which point it is added to the definition object collection. Note that in the new creation mode from the user interface, the temporary drawing sketch is placed on the active sheet, not a temporary sheet such as when the definition is being edited. The definition object in this user interface construction state are accessible through the Parent property of the DrawingSketch object available from the ActiveEditObject property of the Application object or the ActivatedObject property of the DrawingDocument object.

When a drawing aid definition is edited, the definition's sketch is copied into a temporary sheet. If the changes are aborted, the temporary sketch is discarded. If the changes are saved, then the original sketch held by the definition is destroyed and the temporary copy replaces the original. If the change is saved to a copy, a new definition is created and the temporary sketch is assigned to it. Because of this, care must be taken when holding references to drawing aid definition sketch geometry and constraints across Edit and ExitEdit boundaries of the definition object. A reference to any sketch geometry or constraint from the original definition object's sketch will become invalid when an edit operation on the definition is performed and the changes saved. A reference to any sketch geometry or constraint from the edit mode sketch or a drawing aid definition will become invalid when the edit operation is ended and the changes are not saved.

Sketch Constraints

Introduction to sketch constraints

Objects in Autodesk Inventor often relate to each other. The relationship may be as simple as being adjacent to each other, but it may be more complex, such as a combined tangential and vertical alignment. These relationships constrain objects and geometry together, so that any changes to the model cause the constraints to be recomputed and reapplied. If something changes sufficiently to prevent a constraint from being resolved, Autodesk Inventor flags that constraint.

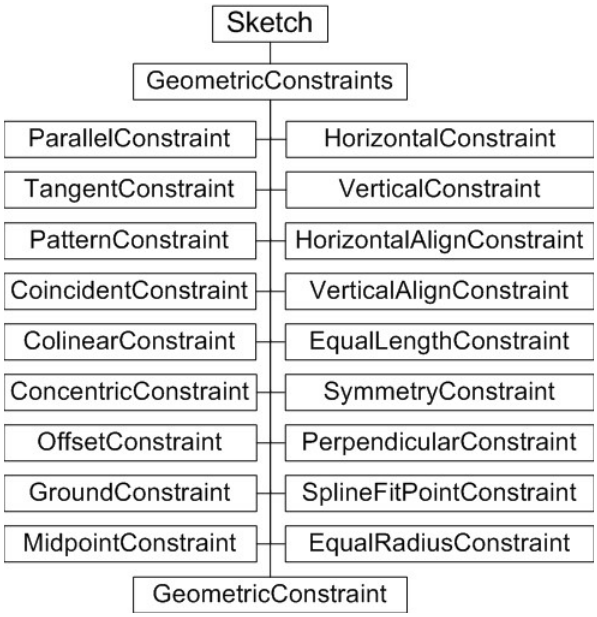
The purpose of sketch constraints

Sketches are the base geometry in Autodesk Inventor - they are the building blocks for parts. Placing geometric constraints on sketches enables and ensures that basic rules for the design and shape of the sketch are enforced. Consequently, the resultant profiles with which part features may be defined follow the same rules. Think of constraints as rules that govern the position, slope, tangency, dimensions, and relationships among sketch geometry. Geometric constraints control the shapes and relationships among sketch elements, effectively removing successive degrees of freedom. The Autodesk Inventor API has a suite of constraint objects, and their proxies, that can be applied to sketches.

There are other types of constraints not covered by this overview, but they perform a similar function. Geometric constraints can also be applied to assembly components, defining relationships between parts in an assembly. See the Assembling Parts overview for more information on working with parts in the assembly environment.

Dimensional constraints in sketches control size, for example, the radius of an arc or the width of a part. All these types of constraints are resolved sequentially during a model compute to achieve the final model structure.

Sketch Constraints Object Model Diagram



Working with sketch constraints through the API

The API has objects for each type of constraint. The GeometricConstraints collection has add methods corresponding to each constraint object type. For example, to create a tangent constraint, call GeometricConstraints.AddTangent. These add methods accept SketchLines, SketchArcs, and SketchPoints as appropriate to their type. For example, a Tangent constraint accepts a SketchLine and SketchArc.

A sample - the effects of constraining a sketch

The following code creates a sketch in a new part document. It defines a number of sketch points within a SketchPoints collection, and then creates SketchLine objects and an arc based on those sketch points. The code omits error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

First, create the new part document, and then add a new sketch:

```
Dim oApp As Inventor.Application
Set oApp = ThisApplication
Dim oPartDoc As PartDocument
Set oPartDoc = oApp.Documents.Add(kPartDocumentObject, _
    oApp.GetTemplateFile(kPartDocumentObject))
Dim oSketch As PlanarSketch
Set oSketch = oPartDoc.ComponentDefinition.Sketches.Add _
   (oPartDoc.ComponentDefinition.WorkPlanes.Item(3))
```

The code creates a number of arbitrary 2D points, using transient geometry to create the points, so create the transient geometry object.

```
Dim oTG As TransientGeometry
Set oTG = oApp.TransientGeometry
```

Now get the SketchPoints collection for the new sketch, and add some points.

```
Dim oSkPnts As SketchPoints
Set oSkPnts = oSketch.SketchPoints
Call oSkPnts.Add(oTG.CreatePoint2d(0, 0), False)
Call oSkPnts.Add(oTG.CreatePoint2d(1.0, 0), False)
Call oSkPnts.Add(oTG.CreatePoint2d(1.0, 0.5), False)
Call oSkPnts.Add(oTG.CreatePoint2d(2.2, 0.5), False)
Call oSkPnts.Add(oTG.CreatePoint2d(0.5, 1.5), False)
Call oSkPnts.Add(oTG.CreatePoint2d(0, 1.0), False)
Call oSkPnts.Add(oTG.CreatePoint2d(2.7, 1.5), False)
Call oSkPnts.Add(oTG.CreatePoint2d(0.5, 1.0), False)
```

Using the previously defined sketch points, add some lines to the SketchLines collection. Here the SketchLine objects are added to an array for easy reference later.

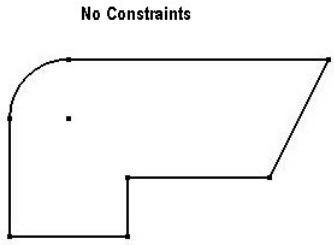
```
Dim oLines As SketchLines
Set oLines = oSketch.SketchLines
Dim oLine(1 To 6) As SketchLine
Set oLine(1) = oLines.AddByTwoPoints(oSkPnts(1), oSkPnts(2))
Set oLine(2) = oLines.AddByTwoPoints(oSkPnts(2), oSkPnts(3))
Set oLine(3) = oLines.AddByTwoPoints(oSkPnts(3), oSkPnts(4))
Set oLine(4) = oLines.AddByTwoPoints(oSkPnts(4), oSkPnts(7))
```

```
Set oLine(5) = oLines.AddByTwoPoints(oSkPnts(7), oSkPnts(5))
Set oLine(6) = oLines.AddByTwoPoints(oSkPnts(6), oSkPnts(1))
```

Add an arc to the SketchArcs collection.

```
Dim oArc As SketchArc
Dim oArcs As SketchArcs
Set oArcs = oSketch.SketchArcs
Set oArc = oArcs.AddByCenterStartEndPoint(oSkPnts(8), oSkPnts(5), oSkPnts(6))
```

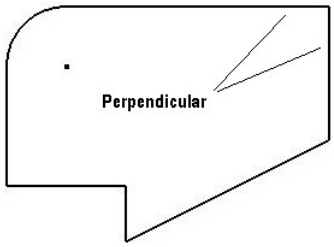
The code to this point creates a sketch with no constraints, appearing as follows:



Now add some constraints. First, add a perpendicular constraint between two lines.

```
Call oSketch.GeometricConstraints.AddPerpendicular(oLine(4), oLine(5))
oApp.ActiveView.Update
```

The preceding code modifies the appearance of the sketch as follows:



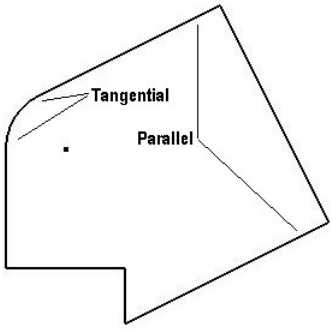
Add tangent constraints to the two lines that join to both ends of the arc.

```
Call oSketch.GeometricConstraints.AddTangent(oLine(5), oArc)
Call oSketch.GeometricConstraints.AddTangent(oLine(6), oArc)
oApp.ActiveView.Update
```

This has no apparent effect - yet - since the lines and arc are already tangential. To see the effect of the tangent constraints, add a parallel constraint between two lines:

```
Call oSketch.GeometricConstraints.AddParallel(oLine(3), oLine(5))
oApp.ActiveView.Update
```

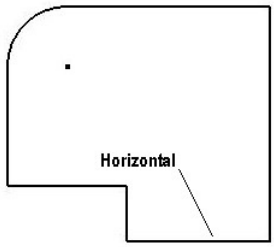
The cumulative effect of these tangential and parallel constraints is as follows:



Now constrain the lower angled line to always be horizontal.

```
Call oSketch.GeometricConstraints.AddHorizontal(oLine(5))
oApp.ActiveView.Update
```

The horizontal constraint results in the following change to the sketch:



Add breakpoints or comment out some of the preceding constraint code to see the effect of different constraints, including the effect of changing the order in which they are applied. The sketch points, unless grounded with a GroundConstraint, are not fixed despite the preceding code having supplied absolute coordinate points. The shape of the sketch is governed by the constraints.

API versus the User Interface

There are some situations, typically involving sketch points, where applying a constraint through the UI works, but the same constraint applied through the API apparently fails. This is not an error, but a reflection of the fact that the UI hides some of the complexities of constraints from the user, while the API does not. For example, creating a coincident constraint between two sketch line endpoints through the API may fail, while the same operation through the UI succeeds. The UI lets you apply a coincident constraint between these two points, but if you look at the results, there is no coincident constraint. In fact, Autodesk Inventor deleted one sketch point and changed both lines to share the remaining point. The API does not hide this from the user since the API expects to return the coincident constraint object, but cannot do that if none were really created. To emulate the UI, merge the sketch points using SketchPoint.Merge.

Summary

Sketch constraints remove successive degrees of freedom in how the sketch is evaluated. It is possible to overconstrain a sketch, so that constraints conflict and cannot be resolved. The UI provides a dialog box to resolve the conflict, while the corresponding API call fails. Sketches can also have dimensional (size) constraints applied to them.

Also consider -

The Autodesk Inventor user interface automatically highlights those parts of a sketch that are appropriate for creating a part feature, such as an extrusion. The API has a specific object to define what constitutes the input for feature creation - the Profile object.

Boundary Representation

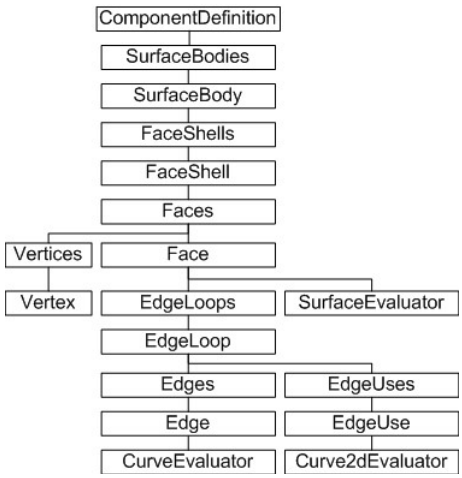
Introduction to Boundary Representation

Conceptually, Autodesk Inventor solids are comprised of features. Sketch profiles are extruded, lofted, revolved, and so on to create solids. Edges of solids have fillets and chamfers applied to them. None of these features are static. All of them are changeable by editing the feature, which may cause changes to other dependent features. Therefore, the surface of an Autodesk Inventor solid is very changeable, too. In other words, the surface information, defined as the boundary between the inside and the outside of the solid, is dynamic.

The purpose of Boundary Representation data

Developer applications frequently need access to this boundary representation data. For example, a cutting application needs to know the shape of a solid so it can prepare a path for a cutting tool. A way is needed to iterate through this boundary information, and that is the purpose of the Boundary Representation API (commonly known as the BRep API).

Boundary Representation (BRep) Object Model Diagram



Working with the BRep API

The BRep API has a number of hierarchical objects. The topmost is the SurfaceBodies collection of SurfaceBody objects. A SurfaceBody object represents a discrete part.

Faces enclose a volume. A Face represents any geometry; planes, cylinders, free-form surfaces, and so on. A simple example is a solid cylinder. The cylinder consists of three faces. Two planar faces that act as the ends of a cylinder and one cylindrical face that acts as the side of the cylinder.

A FaceShell object represents a connected set of faces. A FaceShell for a simple cylinder has the two planar end faces, and the non-planar side face. Typically a SurfaceBody contains a single FaceShell object, but it is possible to have more, as in the following illustration:

The SurfaceBody on the left has one FaceShell. Increasing the hole diameter results in the SurfaceBody on the right, comprised of two FaceShell objects.

EdgeLoop, Edge, and EdgeUse objects

EdgeLoop objects define a face boundary. Several EdgeLoop objects can exist for a given face. The following figure highlights the four EdgeLoop objects for the top face of this solid.

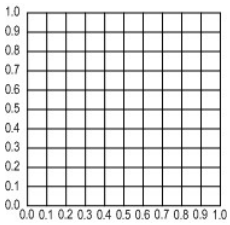
An Edge object is similar but defines only the connectivity between two adjacent faces, so typically represents only part of an EdgeLoop. An Edge object is shared between faces, unlike an EdgeUse object. A face has its own set of EdgeUse objects, which apply only to that face. They are defined in terms of the 2D parameter space of that face. Typically, there are two EdgeUse objects for each Edge Object. Edge objects are shared between faces but EdgeUse objects are not. They contain information specific to a face, including a conceptual flow direction for the face edges, useful in traversing faces.

In the BRep API, a vertex indicates a connection of at least two edges. Vertices are implied by the edges in the model.

BRep objects expose the Geometry property that returns the underlying geometry of the object.

Parameter space

When working with planar objects, it is common to define them in terms of their own plane. Typically, the object is divided into some conceptually convenient number of divisions to reference a given point on its surface. For example, a surface can be parameterized as follows:



Surfaces are parameterized in their 2D space. Edges (curves) are parameterized in a linear fashion. However, both can be evaluated to return coordinates in 3D space. This means that to obtain model-space X Y Z coordinates of a point on a surface, it is necessary to supply the 2D parameter-space coordinates. To obtain a point on an edge, supply a single parameter-space unit to obtain the model-space X Y X coordinates. The BRep API offers several Evaluator objects, through which you can define and obtain this parameter information. The evaluator objects provide methods for converting between parameter space and model space.

Using the API to traverse the BRep hierachy

This sample traverses the BRep hierarchy of an open part document, locating all planar faces. Using the SurfaceEvaluator object, it then determines the center of each face by dividing each side of the 2D parameter range box in half. A sketch point is added at every center point, each created on a new planar sketch.

The following code omits error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

As indicated by the object model diagram, BRep information is obtained from the component definition, so the first step is to get the part component definition for the open part document. Then iterate through the SurfaceBodies collection of SurfaceBody objects. The SurfaceBody object has a Faces property providing direct access to that objects collection of faces. The following code begins to loop through that collection of faces:

```
Dim oPartDef As PartComponentDefinition
Set oPartDef = ThisApplication.ActiveDocument.ComponentDefinition

Dim oSurfaceBody As SurfaceBody
Dim oFace As Face

For Each oSurfaceBody In oPartDef.SurfaceBodies
  For Each oFace In oSurfaceBody.Faces
```

This sample deals only with planar faces, so check the surface type before obtaining the surface evaluator and the parameter range box for this face:

```
  If oFace.SurfaceType = kPlaneSurface Then

    Dim oEval As SurfaceEvaluator
    Set oEval = oFace.Evaluator

    Dim oRange As Box2d
    Set oRange = oEval.ParamRangeRect
```

To calculate the center point of the range box, determine the range of each side and halve it.

```
    Dim params(0 To 1) As Double
    params(0) = oRange.MinPoint.X + (oRange.MaxPoint.X - oRange. _
        MinPoint.X) * 0.5
    params(1) = oRange.MinPoint.Y + (oRange.MaxPoint.Y - oRange. _
        MinPoint.Y) * 0.5
```

The point is in the 2D parameter space of the face. Use the GetPointAtParam method to obtain the X Y Z coordinate of that point in model space, and then create a corresponding Point object.

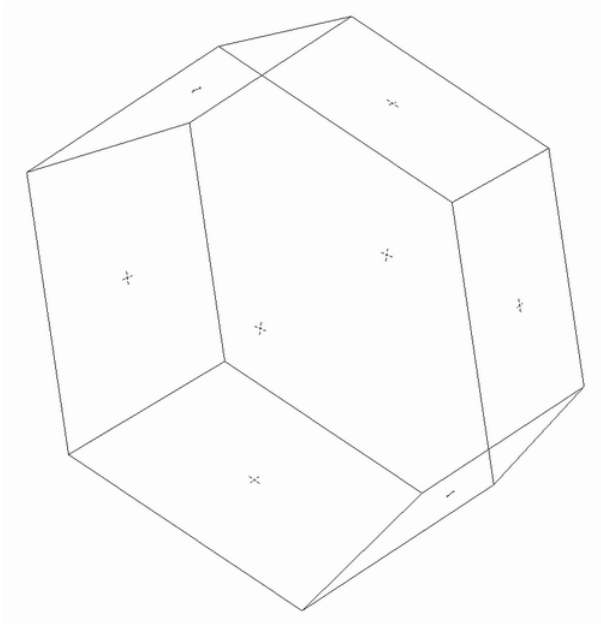
```
    Dim cenPt() As Double
    oEval.GetPointAtParam params, cenPt

    Dim oPoint As Point
    Set oPoint = ThisApplication.TransientGeometry.CreatePoint _
        (cenPt(0), cenPt(1), cenPt(2))
```

Add a new sketch to the sketches collection of the part component definition. Add a sketch point to the sketch, at the previously determined face center point. Use the ModelToSketchSpace method to create the point in the 2D plane of the sketch. Finally, move on to the next Face object or the next SubrfaceBody object.

```
    Dim oSketch As PlanarSketch
    Set oSketch = oPartDef.Sketches.Add(oFace)

    Dim oPoint2D As Point2d
    Set oPoint2D = oSketch.ModelToSketchSpace(oPoint)

    Dim oSkPnt As SketchPoint
    Set oSkPnt = oSketch.SketchPoints.Add(oPoint2D)
```

```
  End If

  Next
Next
```

The sample places new sketches and sketch points in the part document. The sketch points are located at the center of all planar faces. For example, a tapered hexagonal solid results in placement of the following sketch points (here shown in wireframe mode):



BRep objects and Reference Keys

Due to the transient nature of BRep objects, it is necessary to minimize the time a reference to a BRep object is maintained. Any time the model is recomputed, the BRep object references become invalid. Use reference keys to maintain a persistent reference to a particular BRep object between model recomputes.

For example, a reference to a BRep Face object becomes invalid if features that reference that face are added or modified. To handle this case, create a reference key to the face before any features edits, and then use this reference key to obtain the face after each feature has been added.

Summary

The BRep API comprises a group of objects that define the boundary between the inside and the outside of a solid. In addition, these objects provide topological information, indicating contiguous edges and adjacent faces so the developer can traverse these edges and faces. The BRep API also provides evaluator objects to work with BRep surfaces and edges (curves) in their respective parameter spaces. BRep objects expose a Geometry property, through which the underlying geometry of the object can be obtained.

Also consider

Boundary Representation data is transient. It is generated during the session, and changes as the model changes. The model cannot be changed through BRep data. Instead, parts and assemblies must be modified through regular modeling operations - editing features, moving parts, and so on. Refer to individual feature descriptions for their specific parameters.

Features

Introduction to Features

The term *feature* can be confusing to those new to parametric modeling. In common language, it might mean a distinguishing characteristic of some kind. In 3D parametric modeling, it refers to each of the mathematical processes used to build the model, hence the term feature-based modeling.

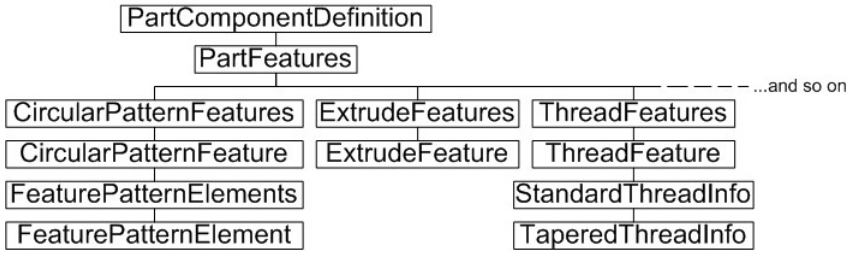
The model represents the result of applying a sequence of features and other changes. Interestingly, due to the hierarchical nature of the computations, a change to a feature, especially one early in the sequence, often results in changes to parts of the model that might have other features applied to them. So application or modification of a simple feature can have far-reaching effects, perhaps causing significant changes in the appearance of the model.

A common example of a feature is an extrusion. Imagine a sketched circle. It is 2D with no thickness. Apply an extrusion feature to this circle (actually to a profile based on the circle), specify an extrusion thickness, and a 3D model is generated. Depending on the thickness, it might look thin like a coin, or long like a cylinder. You can apply further features, such as a fillet feature to round off the edges of the cylinder.

The purpose of Features

The extrusion feature described previously is often the first feature used in building a model, but there are many more feature types; some that generate solids from sketches or profiles, some that modify existing solids. Autodesk Inventor maintains the list of features (and their respective parameters) so they can easily be changed. Imagine if this were not so. You complete a complex model, only to find some minor fillet or chamfer applied early in the process was wrong. In feature-based modelers, such as Autodesk Inventor, you can go to that earlier feature, make your change, and let the entire model be recomputed incorporating the change.

Features Object Model Diagram



The following is a list of current API feature objects. Not all are available at the same time - it depends on the document or environment type. For example, the FlangeFeature object only has meaning in the context of a sheet metal part.

BendFeature	BoundaryPatchFeature	ChamferFeature
CircularPatternFeature	CoilFeature	ContourFlangeFeature
CornerChamferFeature	CornerFeature	CornerRoundFeature
CutFeature	DecalFeature	DeleteFaceFeature
EmbossFeature	ExtrudeFeature	FaceDraftFeature
FaceFeature	FeatureBasedOccurrencePattern	FeaturePatternElement
FeaturePatternElements	FilletFeature	FlangeFeature
FoldFeature	HemFeature	HoleFeature
KnitFeature	LoftFeature	MirrorFeature
MoveFaceFeature	NonParametricBaseFeature	PartFeature
PartFeatureExtent	PunchToolFeature	RectangularPatternFeature
ReferenceFeature	ReplaceFaceFeature	RevolveFeature
RibFeature	ShellFeature	SplitFeature
SweepFeature	ThickenFeature	ThreadFeature

Working with Features through the API

A part component definition provides access to the collection of features to be applied to that part. This PartFeatures object is a collection of other part feature collection types. For example, the ExtrudeFeatures object is a collection of ExtrudeFeature objects. Creating an extrude feature is as simple as adding it to this collection. However, most features require some further input, perhaps in the form of mirror or rotation axes, sketches or profile objects, or radius dimensions, to help define the appropriate result.

Creating an extruded feature from scratch

This sample code steps through the process of creating a solid from an extruded sketch. A new part document is created, a sketch and profile is added, and the profile is extruded to form a 3D solid.

> **Note:** Extruding a sketch through the Autodesk Inventor user interface does not require creation of a profile. This step is hidden from the user. However, the API does require a Profile object (actually a collection of ProfilePath objects) to create some features.

The code omits error checking for the sake of clarity and brevity. Always check that return values are of the expected type. First, add a new part document to the documents collection:

```
Dim oApp As Inventor.Application
Set oApp = ThisApplication

Dim oPartDoc As PartDocument
Set oPartDoc = oApp.Documents.Add(kPartDocumentObject, oApp.GetTemplateFile(kPartDocumentObject))
```

Now add a new sketch to the sketches collection of this document's component definition. Here, the third item in the WorkPlanes collection indicates the XY plane, so that is the plane for the new sketch. For more information on WorkPlanes and other work features, refer to the Work Features overview.

```
Dim oSketch As PlanarSketch
Set oSketch = oPartDoc.ComponentDefinition.Sketches.Add(oPartDoc.ComponentDefinition.WorkPlanes.Item(3))
```

Some transient point objects are required when creating sketch lines, so obtain the TransientGeometry object.

```
Dim oTG As TransientGeometry
Set oTG = oApp.TransientGeometry
```

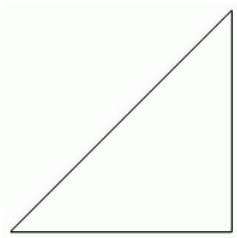
Add three points to the SketchPoints collection of the new sketch. These become the endpoints of three sketch lines forming a triangle. The false argument indicates that these points are not intended as hole feature center points.

```
Dim oSkPnts As SketchPoints
Set oSkPnts = oSketch.SketchPoints
Call oSkPnts.Add(oTG.CreatePoint2d(0, 0), False)
Call oSkPnts.Add(oTG.CreatePoint2d(1, 0), False)
Call oSkPnts.Add(oTG.CreatePoint2d(1, 1), False)
```

When the user draws sketch lines through the Autodesk Inventor user interface, sketch points are not required as they are inferred automatically. This is not the case with the API. Sketch points are required to add sketch lines. Use the preceding sketch points to add three sketch lines to the sketch.

```
Dim oLines As SketchLines
Set oLines = oSketch.SketchLines
Dim oLine(1 To 3) As SketchLine
Set oLine(1) = oLines.AddByTwoPoints(oSkPnts(1), oSkPnts(2))
Set oLine(2) = oLines.AddByTwoPoints(oSkPnts(2), oSkPnts(3))
Set oLine(3) = oLines.AddByTwoPoints(oSkPnts(3), oSkPnts(1))
```

The code thus far creates a new part document containing the following sketch:



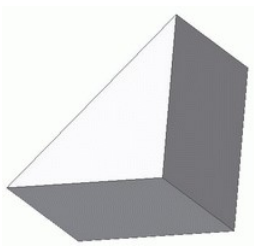
Extrude features require a profile object, so create a profile from this sketch. The AddForSolid method has optional arguments (not used here) that can be used to further manipulate the ProfilePaths within the Profile.

```
Dim oProfile As Profile
Set oProfile = oSketch.Profiles.AddForSolid
```

Now, add an ExtrudeFeature to the Features collection of the document's component definition. Here the AddByDistanceExtent method is used, which requires a distance for the extrude - in this case, 1.0. The other arguments are also similar to the user interface, indicating the extrude should occur in both positive and negative directions, and that a join Boolean operation should be performed.

```
Dim oExtFeature As ExtrudeFeature
Set oExtFeature = oPartDoc.ComponentDefinition.Features.ExtrudeFeatures.AddByDistanceExtent _
    (oProfile, 1.0, kSymmetricExtentDirection, kJoinOperation)
oApp.ActiveView.Fit
```

The code results in a 3D solid appearing as follows:



This sample demonstrated a simple extrude of a sketch to form a 3D solid. The following sample demonstrates creation of a 3D solid by revolving a sketch around an axis. Again, start a new part document and create some sketch points:

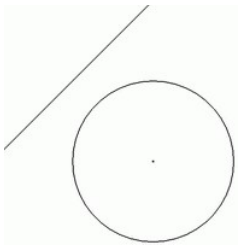
```
Dim oApp As Inventor.Application
Set oApp = ThisApplication
```

```
Dim oPartDoc As PartDocument
Set oPartDoc = oApp.Documents.Add(kPartDocumentObject, oApp.GetTemplateFile(kPartDocumentObject))

Dim oSketch As PlanarSketch
Set oSketch = oPartDoc.ComponentDefinition.Sketches.Add(oPartDoc.ComponentDefinition.WorkPlanes.Item(3))

Dim oTG As TransientGeometry
Set oTG = oApp.TransientGeometry

Dim oSkPnts As SketchPoints
Set oSkPnts = oSketch.SketchPoints

Call oSkPnts.Add(oTG.CreatePoint2d(0, 0), False)
Call oSkPnts.Add(oTG.CreatePoint2d(1, 1), False)
Call oSkPnts.Add(oTG.CreatePoint2d(1, 0), False)
```

Three sketch points have been created. Use two of them to create a sketch line, and the third to locate a sketch circle with a radius of 0.5:

```
Dim oLines As SketchLines
Set oLines = oSketch.SketchLines

Dim oLine As SketchLine
Set oLine = oLines.AddByTwoPoints(oSkPnts(1), oSkPnts(2))

Dim oCircs As SketchCircles
Set oCircs = oSketch.SketchCircles

Dim oCirc As SketchCircle
Set oCirc = oCircs.AddByCenterRadius(oSkPnts(3), 0.5)
```

The new sketch appears as follows:



The intention here is to rotate, or revolve, the circle around the axis formed by the sketch line, thus forming a doughnut shape. The next step is to form a profile from the circle sketch. Note that the AddForSolid method is used again. The only contiguous closed loop in the sketch is the circle, which is the only object that will be used in the resultant profile.

```
Dim oProfile As Profile
Set oProfile = oSketch.Profiles.AddForSolid
```

The only remaining step is to add the RevolveFeature to the RevolveFeatures collection. Here the AddFull method is used, indicating that the revolve should rotate through a full 360 degrees. The alternative is the AddByAngle method. The sketch line is used as the rotation axis.

```
Dim oRevFeature As RevolveFeature
Set oRevFeature = oPartDoc.ComponentDefinition.Features.RevolveFeatures.AddFull _
    (oProfile, oLine, kJoinOperation)
oApp.ActiveView.Fit
```

This RevolveFeature sample code results in a 3D solid appearing as follows:

Intelligent features - iFeatures

The Autodesk Inventor user interface supports insertion of iFeatures. iFeatures are features that you repeatedly use in your work, where some, but not all, parameters tend to change. An iFeature is stored as an .IDE file, and may prompt for required parameters during insertion. The base sketch is incorporated into the iFeature and usually has parameters and constraints applied. When an iFeature is inserted onto a model face, for example, the variable parameters are prompted for and the iFeature shape and placement modified appropriately. In most other respects, Autodesk Inventor treats iFeatures as regular features. The following is a list of current iFeature objects.

iFeatureComponent	iFeatureComponents	iFeatureDefinition
iFeatureEntityInput	iFeatureInput	iFeatureInputs
iFeatureParameterInput	iFeatureTemplateDescriptor	iFeatureTemplateDescriptors
iFeatureVectorInput	iFeatureWorkPlaneInput	

The following code extract shows how an iFeature, named MyiFeature.ide, might be added to a face of a part in an assembly (here the previously selected face object is passed as oFace).

As this sample assumes an assembly context, the face is also in the assembly context, so it gets the part component definition of the face in its part context using the Parent property of the native object. In other words, the face in its part context, not in its assembly context. For more information, see the Proxies overview.

```
Dim oPartCompDef As PartComponentDefinition
Set oPartCompDef = oFace.NativeObject.Parent.Parent
```

Using the part component definition, add a new iFeature component definition to its ReferenceComponents collection. In this case, the code references an iFeature file named MyiFeature.ide.

```
Dim iFeatDef As iFeatureDefinition
Set iFeatDef = oPartCompDef.ReferenceComponents._
    iFeatureComponents.CreateDefinition("c:\MyiFeature.ide")
```

The previously prepared iFeature is added to specified face of the part in the assembly. Use the iFeatureSketchPlaneInput object if additional control is required; for example, over the orientation of the iFeature.

Summary

Features are the building blocks of Autodesk Inventor's parametric feature-based modeler. Features represent a mathematical construct or modification of the model. Autodesk Inventor maintains a sequential list of features and their parameters, any of which can be changed, with the potential to cause the entire model to be recomputed. iFeatures takes this a step further, allowing parametric sketch-based features to be stored as .ide file. The API can add iFeatures to the model.

Also consider

Autodesk Inventor's iFeatures can take advantage of complex constraints, parameters and equations to change or maintain their size and shape. Refer to parameters for more information.

This type of intelligent application of features extends to other workflows too; refer to iMates, iParts, and iProperties.

The Autodesk Inventor user interface can be used to apply assembly features. These are features that apply to a part only in the assembly context. The API can currently query assembly features, but cannot yet create them.

Work Features

Introduction to work features

As an Autodesk Inventor user, it is often necessary to place or manipulate features in 3D space. These features may be in relation to existing geometry, and often are the result of some potentially complex inference from a surface or plane. If existing model geometry is not sufficient to locate new features, some form of construction geometry is required to determine this inferred location. This may be as simple as the division of a line, to a sketch-like construction comprising points, lines and planes. Autodesk Inventor provides this abstract construction geometry in the form of work features, available both through the user interface and through the API. Autodesk Inventor work features provide a rich toolset for building your model, including the ability to constrain to work features, and to project work features onto sketches.

The user interface provides access to work planes, work axes, and work points. These work features can also be created and modified through the API.

The purpose of API work features

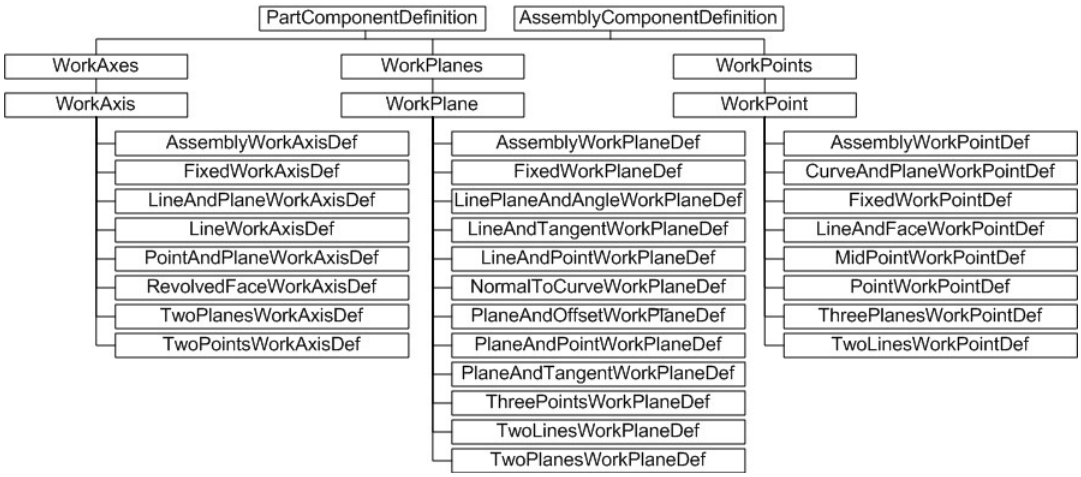
Use work features to construct abstract geometry, avoiding the need for complex calculations to determine feature location. The three types of work features are reflected in the API with the WorkPoint, WorkAxis, and WorkPlane objects, and their corresponding collection objects.

When working with work features through the user interface, all the information needed to place the work feature is inferred from the objects selected, where they were selected, and the order in which they were selected. This isn't feasible through the API, therefore there are a number of work feature subtypes derived from the three base types. For example, to create a WorkPoint at the intersection of three planes, use the ThreePlanesWorkPointDef object. Create this type of WorkPoint object by calling the AddByThreePlanes method of the WorkPoints collection. This takes three planar objects as arguments. Note that typically such arguments are quite flexible. In this example, the planar objects can be sketches, faces, or WorkPlane objects, or any combination of these. In this manner it is possible to build complex construction geometry.

Work features can also be placed using regular, circular or mirror pattern features. To determine whether a work feature is part of a pattern, and cannot therefore be edited, use the IsPatternElement method of the WorkAxis, WorkPoint or WorkPlane.

Note: The work feature collection objects (WorkPoints, WorkAxes, WorkPlanes) are never empty. By default, each contains their base origin point, axes, or planes, respectively. For example, the WorkAxes collection object will always contain the three base axis objects X, Y, and Z, in that order.

Work Features Object Model Diagram



Working with work features through the API

Many Autodesk Inventor API methods for sketch and feature placement accept work feature objects as input arguments. For example, the Add method of SketchSplines3D accepts WorkPoint objects to define the spline control points. Typically though, construction geometry is projected onto a sketch in order to for it to be consumed in feature generation. The nature of work features created through the user interface means they can be in-line, or nested. The API does not support in-line work features.

Note: To project work feature construction geometry on to a sketch, use the AddByProjectingEntity method of the sketch object. This method accepts work feature objects as arguments.

Creating a WorkPoint

As indicated by the preceding object diagram, the work feature collection objects are obtained from the PartComponentDefinition or AssemblyComponentDefinition objects.

The following example shows one way of creating a fixed WorkPoint object using the AddFixed method. Note the use of transient geometry to create a 3D point and then assign the X Y Z coordinates 2, 3, 4 to that point, before adding the WorkPoint to the WorkPoints collection.

```
Dim oPartDoc As PartDocument
Set oPartDoc = ThisApplication.ActiveDocument

Dim oPartCompDef As PartComponentDefinition
Set oPartCompDef = oPartDoc.ComponentDefinition

Dim oTrans As TransientGeometry
Set oTrans = ThisApplication.TransientGeometry

Dim oPnt As Point
Set oPnt = oTrans.CreatePoint(2, 3, 4)

Dim oWorkPoint1 As WorkPoint
Set oWorkPoint1 = oPartCompDef.WorkPoints.AddFixed(oPnt, False)
```

This sample and the following code omit error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

The code adds a WorkPoint to the WorkPoints collection, obtained from the PartComponentDefinition of the PartDocument. This will succeed even if the PartDocument is empty.

The False argument to the AddFixed method indicates that the WorkPoint is not to be considered application-specific construction geometry. This means the WorkPoint is displayed in the browser in the same manner as user-generated work features.

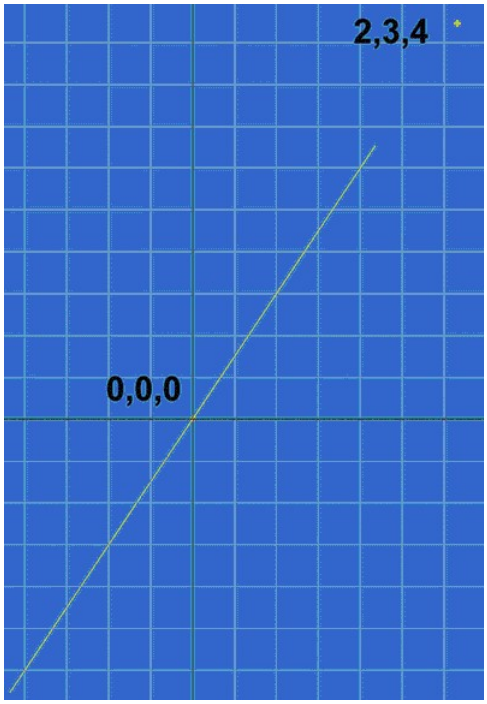
Note: A value of True for this Construction argument indicates this WorkPoint should be considered application-specific construction geometry. It is not displayed in the browser, but it can still be manipulated from code. Thus an application can generate many work features without overloading the browser with WorkPoints, WorkAxes, and WorkPlanes that mean nothing to the user.

Creating a WorkAxis

The following code creates a WorkAxis and adds it to the WorkAxes collection. The AddByTwoPoints method is used, so another WorkPoint is defined with X Y Z values of 0, 0, 0. So the WorkAxis will be defined by two points; the previously created WorkPoint at 2, 3, 4, and a new one at 0, 0, 0.

```
Dim oWorkPoint2 As WorkPoint
Set oPnt = oTrans.CreatePoint(0, 0, 0)
Set oWorkPoint2 = oPartCompDef.WorkPoints.AddFixed(oPnt, False)

Dim oWorkAxis As WorkAxis
Set oWorkAxis = oPartCompDef.WorkAxes.AddByTwoPoints(oWorkPoint1, oWorkPoint2, False)
```

The AddByTwoPoints method creates the WorkAxis and adds it to the WorkAxes collection. Note the third argument again indicates that the newly created work feature should not be considered application-specific construction geometry. It is displayed on screen, appearing something like this:



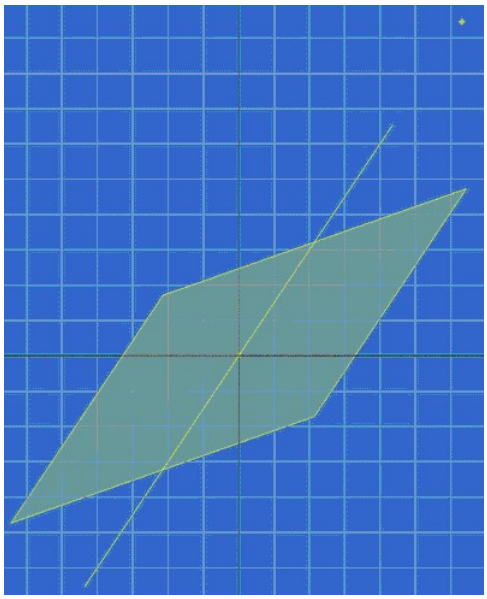
Creating a WorkPlane

With just two points, there is insufficient information to create a plane. There are a number of different methods to generate planes - for example, by using a third point, or through an axis, with the plane angle relative to another plane. The following example uses this latter method. Note that one of the predefined WorkPlane objects, the Y-Z plane, is used as a reference plane from which an angle of 45 degrees is measured.

Note: This code also demonstrates the ability to name work features. When dealing with a large number of objects, it is easier to reference them by name than by numerical index. The same applies to the base work feature objects. The name of the Y-Z plane, the first one in the WorkPlanes collection, is YZ Plane. This could be referenced as Item(1), but it is easier to read as Item("YZ Plane").

```
Dim oWorkPlane As WorkPlane
Set oWorkPlane = oPartCompDef.WorkPlanes.AddByLinePlaneAndAngle _
   (oWorkAxis, oPartCompDef.WorkPlanes.Item("YZ Plane"), 45, False)

oWorkPlane.Name = "MyFirstWorkPlane"
```

The last line of code names the newly created WorkPlane object MyFirstWorkPlane. Other code can use this name to reference this work feature. The new name is also displayed in the Autodesk Inventor browser, provided the work feature is not application-specific construction geometry. It is displayed as follows:



Summary

API work feature objects solve the same problem for the developer that work feature commands solve for the user. They provide the means to construct abstract geometry for the purpose of placing features, parts, sketches and so on. API work features cannot infer construction information from user object selection, so a number of methods are provided for specific input requirements. Objects of type WorkPoint, WorkAxis, or WorkPlane can be created and modified.

Also consider

For creating geometry that is to form a part (for example, a profile to be extruded and perhaps shared) use the Sketch objects such as SketchEllipse, SketchCircle, SketchLine, SketchPoint, and so on.

For creating transient geometry that is intended only as a visual cue, use the ClientGraphics objects to create custom graphics.

Assembling Parts

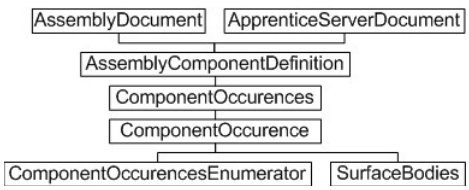
Introduction to parts in an assembly

Modeling in Autodesk Inventor is more than creating part features from sketch profiles. Typically, a library of part files is created and these parts are assembled together to create a composite model - an assembly. Assemblies themselves can be treated as discrete components, as subassemblies within a larger assembly. Autodesk Inventor has an extensive user interface for working with parts in the assembly environment, including positioning tools and assembly constraints for defining relationships between parts.

The purpose of assembling parts

This method of gathering parts into an assembly has several advantages. It enables model components to be worked on independently, and it often means the model is considerably smaller in file size, especially when common parts are re-used. The concept it similar to blocks in AutoCAD.

Assembly-level component occurrences and definitions - Object Model Diagram



Working with assembled parts through the API

The Autodesk Inventor API employs the concepts of component definitions and occurrences. These are somewhat analogous to AutoCAD block definitions and inserts. A component definition is also the base class for other component definition types - PartComponentDefinition, AssemblyComponentDefinition, SheetMetalComponentDefinition, WeldmentComponentDefinition, and WeldsComponentDefinition. These are used only when accessing functionality specific to these types.

Think of a component occurrence as an instance of a component definition. The component definition can be a subassembly or a part. The ComponentDefinition object is the base class for the AssemblyComponentDefinition object.

To insert parts into an assembly, add an occurrence to the ComponentOccurrences collection of the assembly document's AssemblyComponentDefinition object. The ComponentOccurrences collection supports several methods to add an occurrence - by file (the Add method) or by inserting another instance of an existing definition (the AddByComponentDefinition method). It also supports iParts and adding through iMates.

Adding occurrences is the first step

Assembling parts also requires a means to specify their positions, and to define how the parts fit together. The add methods described previously support matrices, allowing ComponentOccurrences to be positioned within the assembly document. However, it is typically more convenient to define assembly constraints to indicate how parts should position themselves relative to each other. Creating an assembly of parts through the API is therefore a two-stage process, since it is through the UI. First, assemble the parts, and then fit them together. The following sample code demonstrates both processes.

This code assumes a part file named cylinder.ipt exists in a C:\Temp directory. Use Autodesk Inventor to create and save this part file (a sketched circle extruded to form a cylinder) before running this code. The code omits error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

First, create a new assembly document:

```
Dim oApp As Inventor.Application
Set oApp = ThisApplication

Dim oAssyDoc As AssemblyDocument
Set oAssyDoc = oApp.Documents.Add(kAssemblyDocumentObject, _
   oApp.GetTemplateFile(kAssemblyDocumentObject))
```

Inserting component occurrences requires a matrix, even if it does nothing, so create the matrix object.

```
Dim oPositionMatrix As Matrix
Set oPositionMatrix = oApp.TransientGeometry.CreateMatrix
```

Now create the component occurrence by calling the Add method of the ComponentOccurrences collection, referencing the part file previously created manually in Autodesk Inventor.

```
Dim sFileName As String
sFileName = "c:\temp\cylinder.ipt"

Dim oCylinder1 As ComponentOccurrence
Set oCylinder1 = oAssyDoc.ComponentDefinition.Occurrences.Add(sFileName, oPositionMatrix)
```

This created the first occurrence of the cylinder part in the assembly. Now add another, but this time, move it to one side. Otherwise it occupies the same space as the first one and is not visible. Use the matrix object to adjust its insertion position slightly. Here, the matrix also reverses the direction of the cyclinder.

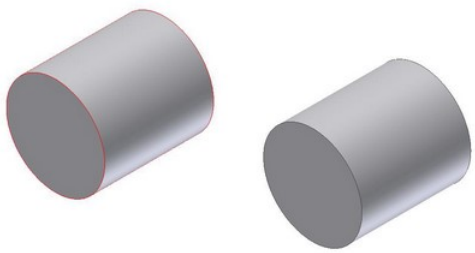
```
Dim oTrans As Vector
Set oTrans = oApp.TransientGeometry.CreateVector(2, 0, -1)
oPositionMatrix.SetTranslation oTrans
```

Now add the second occurrence of the cylinder component.

Note: The reference to the part file is no longer needed when inserting the second cylinder, since there is already an instance of the part in the assembly. Simply reference the same ComponentDefinition, in this case that of the previously inserted ComponentOccurrence.

```
Dim oCylinder2 As ComponentOccurrence
Set oCylinder2 = oAssyDoc.ComponentDefinition.Occurrences.AddByComponentDefinition _
   (oCylinder1.Definition, oPositionMatrix)
```

In Autodesk Inventor, the preceding code results in two cylinders in the assembly, looking something like the following figure.



The intention is to get these cylinders to line up and butt together, and for Autodesk Inventor to ensure they stay that way despite future recomputes of the assembly model. Use assembly constraints to achieve this, in much the same way geometric constraints can define the shape of a sketch. Apply mate constraints to remove two degrees of freedom, aligning the cylinders to each other, and then mating the ends of the cylinders together. The AssemblyComponentDefinition maintains a collection of constraints to apply to the component occurrences, so add the mate constraints to this collection. First, obtain the assembly component definition.

```
Dim oAxisDef As AssemblyComponentDefinition
Set oAxisDef = oApp.ActiveDocument.ComponentDefinition
```

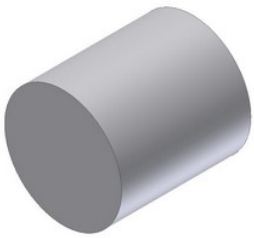
The first constraint will be applied between the curved faces of the two cylinders, to align them. One way to obtain these faces is by iterating through all the faces that make up the surface body of each component occurrence, checking for a cylindrical surface. For more information on faces and surface bodies, refer to the Boundary Representation (BRep) sections of the Autodesk Inventor API documentation.

```
Dim oCylAxis1 As Face
Dim oCylAxis2 As Face
Dim oFace As Face

For Each oFace In oCylinder1.SurfaceBodies(1).Faces
    If oFace.SurfaceType = kCylinderSurface Then
       Set oCylAxis1 = oFace
    End If
Next

For Each oFace In oCylinder2.SurfaceBodies(1).Faces
    If oFace.SurfaceType = kCylinderSurface Then
       Set oCylAxis2 = oFace
    End If
Next
```

So now oCylAxis1 and oCylAxis2 contain the curved cylinder faces, and can be passed to the AddMateConstraint method of the assembly component definition's AssemblyConstraints collection.

```
Dim oConstr As AssemblyConstraint
Set oConstr = oAxisDef.Constraints.AddMateConstraint(oCylAxis1, oCylAxis2, 0, kInferredLine, kInferredLine)
```

The result of this constraint is shown in the following figure. The two cylinders now occupy the same space, and are indistinguishable from each other. Another constraint is required on the end faces to have the two cylinders butt against each other to form one long cylinder.



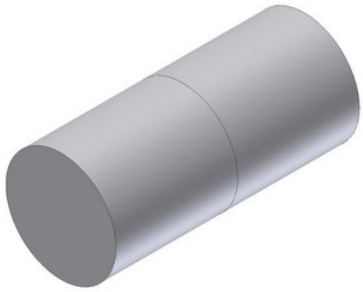
Again, iterate through the faces of the surface bodies, this time looking for planar faces. In this case, it is safe to select the face on the same end of each cylinder as the position of the second cylinder was reversed by its insertion matrix (the two cylinder component occurrences were inserted in opposite directions).

```
Dim oCylFace1 As Face
Dim oCylFace2 As Face

For Each oFace In oCylinder1.SurfaceBodies(1).Faces
    If oFace.SurfaceType = kPlaneSurface Then
        Set oCylFace1 = oFace
    End If
Next

For Each oFace In oCylinder2.SurfaceBodies(1).Faces
    If oFace.SurfaceType = kPlaneSurface Then
       Set oCylFace2 = oFace
    End If
Next
```

Now add a mate constraint for these two faces to the AssemblyConstraints collection.

```
Set oConstr = oAxisDef.Constraints.AddMateConstraint _
   (oCylFace1, oCylFace2, 0, kNoInference, kNoInference)
```

This results in the following figure with the intended relationship between the two cylinders.



The preceding example demonstrates insertion of parts into the assembly environment, and subsequent manipulation of those parts. The resulting assembly can be saved to file, and itself used as a component occurrence in a new assembly with other parts and subassemblies.

Summary

To insert a part into an assembly, a ComponentOccurrence object is added to the ComponentOccurrences collection of the assembly document's AssemblyComponentDefinition object. The position of the occurrence can be manipulated during insertion. Final positioning is typically achieved through assembly constraints, which define how parts and subassemblies relate to each other. Multiple instances of the same ComponentOccurrence should reference the same ComponentDefinition.

Also consider -

Autodesk Inventor supports the concept of patterns, which is a uniform (circular or rectangular) or nonuniform (feature-based) repetition of parts. The API supports such patterns too, though the RectangularOccurrencePattern, CircularOccurrencePattern and FeatureBasedOccurrencePattern objects. These contain collections of OccurrencePatternElement objects that provide access to the list of ComponentOccurrences comprising that pattern.

Parts in the assembly environment are actually considered proxies. This only really matters when working with a part in the assembly context rather than the part definition context. For more information, refer to the Proxies overview.

Bill of Materials

Introduction to the BOM

Autodesk Inventor keeps track of which parts comprise an assembly. This information can be tabulated into a bill of materials report, commonly used in the manufacturing and assembly processes. Minimally, a BOM lists components, quantity, and relevant totals. Additionally, a BOM likely includes part or stock numbers, cut length figures, and so on. Autodesk Inventor can add all this information to a drawing in the form of a table.

The purpose of the BOM API

The API allows the following actions on BOM data.

- Data can be queried and exported.
- Quantity totals can be modified - for example, overriding a value or setting a quantity to be a parameter value.
- The BOM component type (the BOM structure) can be modified.

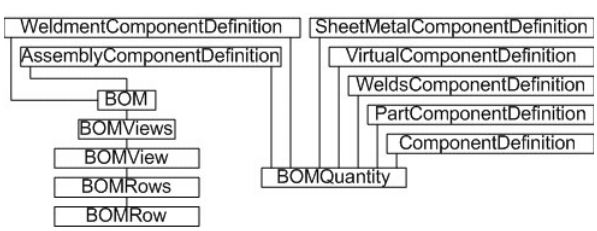
A typical use for the BOM API is to query and export data for a specific use, perhaps to be read by other software downstream.

The BOM Structure defines the status of the component in the BOM. BOM Structure has five basic options:

- Normal
- Phantom
- Reference
- Purchased
- Inseparable

Phantom components exist in the design but are not separate line items in the BOM. Reference components are excluded from the BOM entirely, existing only to augment the content of a design. A Purchased component is considered a single BOM line item, even if it is a subassembly. An Inseparable component is similar to a Purchased component, in that it is considered a single item, except that it is typically fabricated.

BOM Object Model Diagram



Working with the BOM through the API

Autodesk Inventor maintains the BOM internally, updating types and quantities as changes are made to an assembly. These changes may be implemented through the user interface or through the modeling functions of the Autodesk Inventor API. The BOM API can make additional changes to the BOM data, and can query and export that data. Autodesk Inventor makes use of PropertySet objects to store some BOM data for a component. Client code can access this data directly.

Querying the BOM

The following sample code demonstrates use of the API to query an assembly for BOM data. The code assumes that the sample assembly, Arbor_Press.iam, is open in Autodesk Inventor. The code omits error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

First, obtain the BOM object for this assembly's ComponentDefinition.

```
Dim oBOM As BOM
Set oBOM = ThisApplication.ActiveDocument.ComponentDefinition.BOM
```

From the BOM object, get the base BOMView object, named "Structured." A BOM may also contain views where the items have been reordered or renumbered.

```
Dim oBOMView As BOMView
Set oBOMView = oBOM.BOMViews.Item("Structured")
```

Iterate through the rows in the BOM view, obtaining each BOMRow object.

```
Dim i As Long
For i = 1 To oBOMView.BOMRows.Count

  Dim oRow As BOMRow
  Set oRow = oBOMView.BOMRows.Item(i)
```

For some of the BOM data, we'll need to query a property set of the component referenced by each row. The PropertySets collections are accessed through the parent document of the ComponentDefinition, so obtain the ComponentDefinition of the component.

```
  Dim oCompDef As ComponentDefinition
  Set oCompDef = oRow.ComponentDefinitions.Item(1)
```

The required property set is named "Design Tracking Properties." Obtain this from the owning document of the component.

```
  Dim oPropSet As PropertySet
  Set oPropSet = oCompDef.Document.PropertySets.Item("Design Tracking Properties")
```

Now we have all the information necessary to list the BOM content. Obtain variable information such as the item number and quantity from the BOMRow object, and the part number and description from the property set.

```
  Debug.Print "#: "; oRow.ItemNumber; _
              " Quantity:"; oRow.ItemQuantity; _
              "Part: "; oPropSet.Item("Part Number").Value; _
              " Desc: "; oPropSet.Item("Description").Value
Next
```

The preceding code iterates through all the BOMRows in the BOMView object, printing BOM data to the VBA debug window. The output should appear something like the following example.

```
#: 1 Quantity: 1 Part: Arbor Press Desc:
#: 2 Quantity: 1 Part: FACE PLATE Desc:
#: 3 Quantity: 1 Part: PINION SHAFT Desc:
#: 4 Quantity: 1 Part: LEVER ARM Desc:
#: 5 Quantity: 1 Part: THUMB SCREW Desc:
#: 6 Quantity: 1 Part: TABLE PLATE Desc:
#: 7 Quantity: 1 Part: RAM Desc:
#: 8 Quantity: 2 Part: HANDLE CAPipt Desc:
#: 9 Quantity: 1 Part: COLLAR Desc:
#: 10 Quantity: 1 Part: GIB PLATE Desc:
#: 11 Quantity: 1 Part: GROOVE PIN Desc:
#: 12 Quantity: 4 Part: ANSI B18.3 - 1/4 - 20 - 7/8 Desc: Hexagon Socket Head Cap Screw
#: 13 Quantity: 4 Part: ANSI B18.3 - 10-32 UNF x 0.58 Desc: Hexagon Socket Set Screw - Flat Point
#: 14 Quantity: 1 Part: ANSI B18.6.2 - 10-32 UNF - 0.1875 Desc: Slotted Headless Set Screw - Flat Point - UNF (Fine Thread - Inch)
```

Summary

Autodesk Inventor can keep track of components in an assembly for the purposed of producing a bill or materials. Components referenced by the BOM can be assigned specific types indicating whether - and how - they should be incorporated into the BOM. Views of the BOM can be renumbered and reordered. The API allows this data to be queried and exported, either in predefined file formats or by reading and manipulating the BOM data directly.

Also consider

The BOM objects described previously are global in nature. The API also supports the PartsList object in the drawing environment. This can be considered a local snapshot of the parts of the BOM relevant to that drawing's parts list.

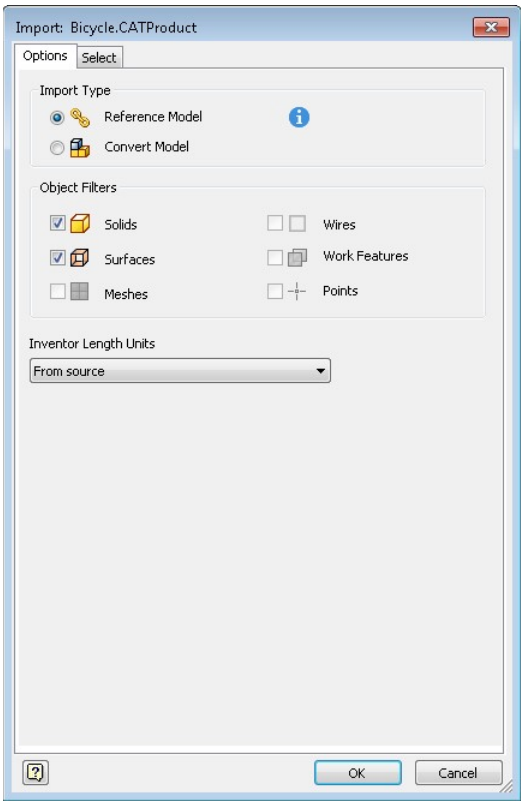
Any CAD

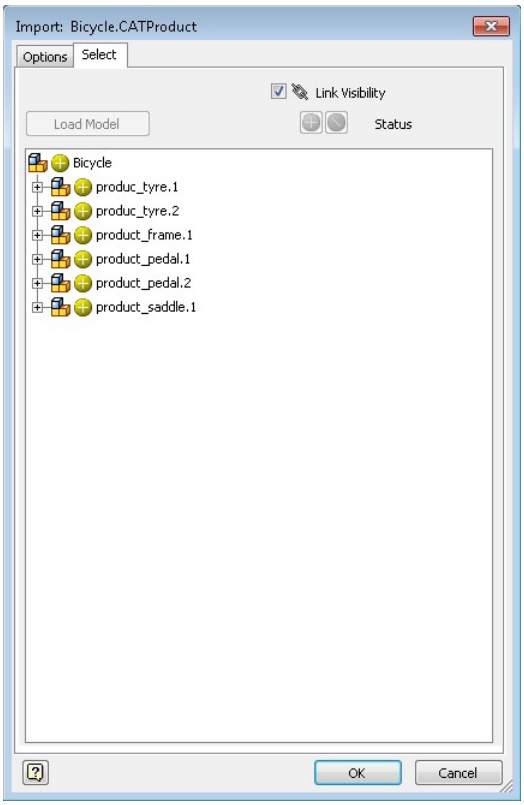
Introduction to AnyCAD

AnyCAD is introduced since Inventor 2016, it not only means to consolidate the import process for any other CAD file formats that Inventor can translate into Inventor data, additionally it also means that the other CAD files can be associatively imported into Inventor. Currently below CAD files can be associatively imported into Inventor:

Alias, CATIA V5, ProE/Creo Patametric, NX, SolidWorks, STEP and AutoCAD

AnyCAD consolidated import process in UI:





In Inventor API, the associatively imported AnyCAD files are defined as ImportedComponent objects. To create an ImportedComponent you should define the ImportedComponentDefinition for it first, use the ImportedComponents.CreateDefinition allows you to create an ImportedComponentDefinition object, please note that currently we have two definition objects for AutoCAD and other CAD file formats respectively: The ImportedDWGComponentDefinition and ImportedGenericComponentDefinition objects are derived from the ImportedComponentDefinition, and the created ImportedComponentDefinition should be either ImportedDWGComponentDefinition or ImportedGenericComponentDefinition, which is determined by the input AnyCAD file format, and the ImportedDWGComponentDefinition or ImportedGenericComponentDefinition have more properties than the base ImportedComponentDefinition object which allow users to set how to import AnyCAD file into Inventor.

Working with AnyCAD through the API

When import a generic AnyCAD file other than AutoCAD DWG file to Inventor part and assembly you can refer to below VBA samples:

Import a generic AnyCAD file to Inventor part:

```
Sub AssociativelyImportAliasToPartSample()
    Dim oDoc As PartDocument
    Set oDoc = ThisApplication.Documents.Add(kPartDocumentObject)

    Dim oPartCompDef As PartComponentDefinition
    Set oPartCompDef = oDoc.ComponentDefinition

    ' Create the ImportedGenericComponentDefinition bases on an Alias file
    Dim oImportedGenericCompDef As ImportedGenericComponentDefinition
    Set oImportedGenericCompDef = oPartCompDef.ReferenceComponents.ImportedComponents.CreateDefinition("C:\ProjectName\iPod.wire")

    ' Set the ReferenceModel to associatively import the Alias file, set this property to False will just convert the
    oImportedGenericCompDef.ReferenceModel = True
    oImportedGenericCompDef.IncludeAll

    ' Import the Alias
    Dim oImportedComp As ImportedComponent
    Set oImportedComp = oPartCompDef.ReferenceComponents.ImportedComponents.Add(oImportedGenericCompDef)
End Sub
```

Import a generic AnyCAD file to Inventor assembly:

```
Sub AssociativelyImportSolidworksToAssemblySample()
    Dim oDoc As AssemblyDocument
    Set oDoc = ThisApplication.Documents.Add(kAssemblyDocumentObject)

    Dim oAssyCompDef As AssemblyComponentDefinition
    Set oAssyCompDef = oDoc.ComponentDefinition

    'Create the ImportedGenericComponentDefinition bases on an Alias file
    Dim oImportedGenericCompDef As ImportedGenericComponentDefinition
    Set oImportedGenericCompDef = oAssyCompDef.ImportedComponents.CreateDefinition("C:\ProjectName\iPod.SLDPRT")

    'Set the ReferenceModel to associatively import the Alias file
    oImportedGenericCompDef.ReferenceModel = True

    'Import the Solidworks to assembly
```

```
    Dim oImportedComp As ImportedComponent
    Set oImportedComp = oAssyCompDef.ImportedComponents.Add(oImportedGenericCompDef)
End Sub
```

When import an AutoCAD DWG file into part you can refer to below VBA sample:

```
Sub AssociativelyImportDWGToPartSample()
    Dim oDoc As PartDocument
    Set oDoc = ThisApplication.Documents.Add(kPartDocumentObject)

    Dim oPartCompDef As PartComponentDefinition
    Set oPartCompDef = oDoc.ComponentDefinition

    'Create the ImportedDWGComponentDefinition bases on an AutoCAD DWG file
    Dim oImportedDWGCompDef As ImportedDWGComponentDefinition
    Set oImportedDWGCompDef = oPartCompDef.ReferenceComponents.ImportedComponents.CreateDefinition("C:\ProjectName\Basic.dwg")

    'Import the AutoCAD DWG
    Dim oImportedComp As ImportedComponent
    Set oImportedComp = oPartCompDef.ReferenceComponents.ImportedComponents.Add(oImportedDWGCompDef)
End Sub
```

More info about AnyCAD

When import generic AnyCAD files into assembly documents, Inventor will generate embedded documents referenced by component occurrences for the imported components, and the ComponentOccurrence.HasAssociativeImportedComponent can be used to determine if a ComponentOccurrence has associative imported component, and then use the ComponentOccurrence.ImportedComponent to get the imported component if it has. Please be aware that if you associatively import AnyCAD file which is an assembly file(like Solidworks assembly), only the top AnyCAD assembly will be treated as ImportedComponent, that means only the top level ComponentOccurrence will have the HasAssociativeImportedComponent return True, and the sub-occurrences won't have equivalent imported components, but you can use the ComponentOccurrence.IsAssociativelyImported to check whether it is created along with importing AnyCAD assembly file, and the ComponentOccurrence.AssociativeForeignFilename returns the referenced AnyCAD file name. For the embedded documents, they are saved in the same file as the embedding document on disk, you can access them via API but because they are resulted from the imported AnyCAD files you should not treat them as normal Inventor documents, and the embedded documents will be updated if their referenced AnyCAD files have changed, the IsEmbeddedDocument can tell you if a part or assembly is embedded document, so you can ignore it because edit or save embedded documents directly are not allowed.

Drawing Views

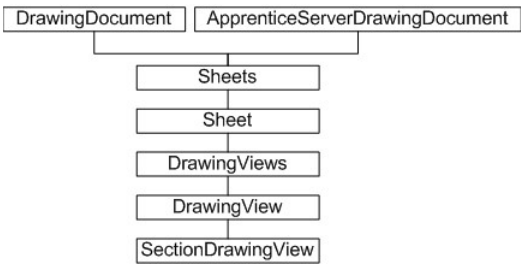
Introduction to drawing views

The Autodesk Inventor user interface allows placement of drawing views on a drawing sheet. These views are 2D representations of the 3D model, and can be orthographic, isometric, or they can have an arbitrary view angle. They can be scaled views to show fine detail, or they can be section views. The API provides the same functionality through the collection of DrawingView objects.

The purpose of the drawing views API

The drawing view API enables code to create, place and manipulate drawing views, including base views, on drawing sheets. View types supported include projected, section, and draft views. Additionally, the API supports the OnViewUpdate event, provided through the DrawingViewEvents object, allowing application code to be notified if an associated draft view is updated due to a change to the model.

Drawing views object model diagram



Working with drawing views through the API

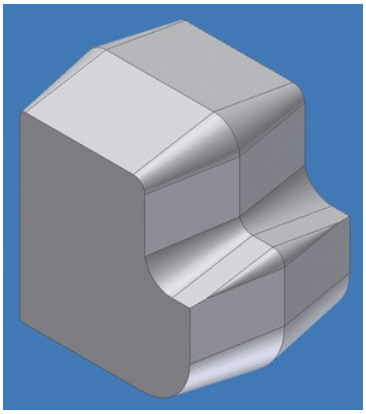
The DrawingViews collection is obtained from the Sheet object. The Sheets collection is obtained from the DrawingDocument object, or if Apprentice Server is being used, from the ApprenticeServerDrawingDocument. The latter hierarchy provides read-only access to the drawing views.

The various types of drawing views are represented by the DrawingView object, with the exception of section views, which are of type SectionDrawingView. Section, draft, and associated draft views can only be created on the active sheet.

The DrawingView and SectionDrawingView objects provide a number of transformation methods designed to translate points between drawing view, model, and sheet spaces. These methods include DrawingViewToModelSpace, DrawingViewToSheetSpace, ModelToDrawingViewSpace, ModelToSheetSpace, SheetToModelSpace, and SheetToDrawingViewSpace.

Creating a drawing view

The following code assumes a drawing document is open in Autodesk Inventor, and that the source part file (part.ipt) is located in the root of the C: drive. This file reference can be changed to suit. Here is the part used for this example.



This sample and the following code omit error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

The first step is to obtain the sheet object, in order to place a view. The location of the view is provided by a 2D point. The following code uses transient geometry to define a 2D point at X=5, Y=5.

```
Dim oDrawingDoc As DrawingDocument
Set oDrawingDoc = ThisApplication.ActiveDocument

Dim oSheet As Sheet
Set oSheet = oDrawingDoc.Sheets.Item(1)

Dim oPoint1 As Point2d
Set oPoint1 = ThisApplication.TransientGeometry.CreatePoint2d(5#, 5#)
```

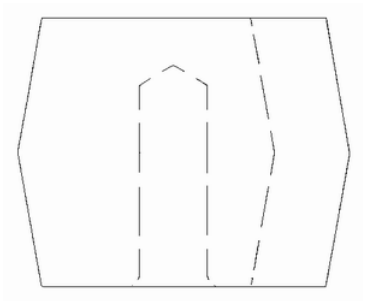
This sample places a regular drawing view of a model part. The following code uses an arbitrary hard-coded part file reference that should be changed as appropriate.

Note: The False argument to the document Open method indicates the part file is to be opened invisibly. In this context, there is no need to see the part in Autodesk Inventor. It is simply the source for the drawing view. Close the part file afterwards.

The new hidden line DrawingView object is added to the DrawingViews collection using a bottom view of the opened part file, at point 5,5, at a scale of 1:1. Immediately afterwards, the part file is closed. It is good practice to keep file references open no longer that necessary.

```
Dim oPartDoc As PartDocument
Set oPartDoc = ThisApplication.Documents.Open("c:\testpart.ipt", False)

Dim oView1 As DrawingView
Set oView1 = oSheet.DrawingViews.AddBaseView(oPartDoc, _
    oPoint1, 1#, kBottomViewOrientation, kHiddenLineDrawingViewStyle)

Call oPartDoc.Close(True)
```

The following is an example of how this appears on the sheet.



Creating a section drawing view

The previous example placed a drawing view of a part. The following code places a new section drawing view through the previously placed view. Section views require that a section line be defined. This is the line through which the part is to be cut. However, unlike section lines created through the user interface, section lines created through the API are not constrained to the model, and therefore do not automatically update to reflect changes to the model.

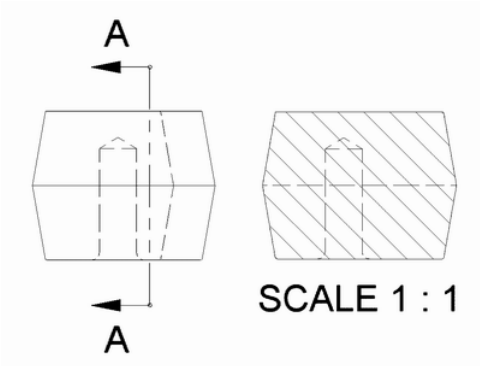
The first task is to indicate the section line, either using existing geometry, or by creating a sketch containing a sketch line, as in the following code.

```
Dim oPoint2 As Point2d
Set oPoint2 = ThisApplication.TransientGeometry.CreatePoint2d(15#, 40#)

Dim oDrawingSketch As DrawingSketch
Set oDrawingSketch = oView1.Sketches.Add

oDrawingSketch.Edit
Dim oSketchLine As SketchLine
Set oSketchLine = oDrawingSketch.SketchLines.AddByTwoPoints(oPoint1, _
    oPoint2)
oDrawingSketch.ExitEdit
```

For simplicity, this code reuses the view placement points to define the ends of the sketch line. In reality, this sketch line would be placed at a specific point relative to the base view. Now to add the section view by calling AddSectionView.

```
Dim oView2 As SectionDrawingView
Set oView2 = oSheet.DrawingViews.AddSectionView(oView1, _
     oDrawingSketch, oPoint2, kHiddenLineRemovedDrawingViewStyle)

oDrawingSketch.Visible = False
```

Note the absence of a part file reference in the preceding code. Unlike base view creation, the previously created view is passed as an argument, together with the sketch line indicating the cut line. The two views now look something like the following illustration.



The arrows and notation in the preceding figures were generated automatically.

Summary

The drawing view API functionality is equivalent to much of the drawing view functionality available through the user interface. It allows for the placement of different types of views on a drawing sheet. These views may be blank draft views, projected views, or section views. The API also provides a means of notification if the underlying model is subject to change.

Also consider

Other drawing view API functionality includes the ability to edit drawing dimensions on a sheet through the DrawingDimension object. Sheet title blocks and templates can be edited through the use of drawing text (the TextBox object and associated styles) and drawing sketches.

Drawing Dimensions

Introduction to Drawing Dimensions

Autodesk Inventor has two different types of dimensions - model and drawing. These can be thought of as controlling and documenting, respectively. Model dimensions are typically parametric constraints intended to control part or feature size. Sketch dimensions are included in this category since they directly affect model size and constraints.

Drawing dimensions simply document a given dimension on a drawing sheet, and have no effect on model size or constraints. However, drawing dimensions can be promoted from a draft view to an underlying sketch, or can be retrieved from a sketch or model.

Drawing dimensions are associative, but not parametric. There are various types of drawing dimensions supported, including general, linear, ordinate, radius, diameter, and angular. Options controlling the appearance of dimensions are specified through dimension styles, enabling support of common dimension standards. Individual settings may be overridden to suite.

A look at the Drawing Dimensions API

The Drawing Dimensions API provides access to all drawing dimensions on a sheet. Objects corresponding to the available dimension types are derived from a common object named DrawingDimension which contains common methods and properties, such as tolerance, precision, text and so on.

Ordinate dimensions differ from other types in that a given sequence of dimension values will reference the same single point of origin. Figure 1 shows the object model diagram.

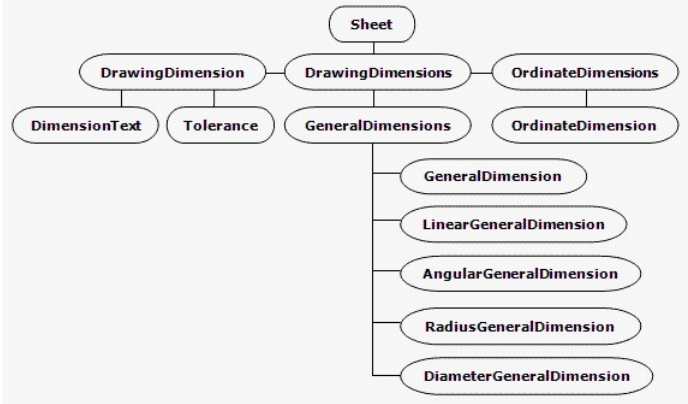


Figure 1

Geometry Intent

Drawing dimension creation methods expect geometry points to be supplied in the form of GeometryIntent objects. To better understand such objects, imagine a leader line and arrow pointing to a user-selected point on a drawing line. Leader line associativity to the selected spot on the drawing line needs to be maintained, but there is no point geometry midway on the drawing line to reference. In this case, a GeometryIntent object encapsulates the intent to reference a location on the drawing line, a certain distance from a particular end.

Similarly, dimensions require GeometryIntent objects because, unlike the Autodesk Inventor modeling environment, the drawing environment contains only 2D lines, arcs and circles - no points. So, a GeometryIntent object for a dimension might reference a particular end of a line or arc, or the center of a circle or arc.

Sample

This sample VBA code listing may be cut and pasted into a module in the Autodesk Inventor VBA editor. Make sure all grayed sections are present. Immediately before each grayed section is text describing that section's purpose and operation - do not copy this text into the editor. The code omits error checking for the sake of clarity and brevity. In your code, always check that return values are of the expected type. Ensure that the Autodesk Inventor Object Library is available in your project; in the Visual Basic Editor, pick Tools > References, and check the appropriate reference.

Creating a Linear Dimension

This code creates a simple linear dimension of a curve on a drawing sheet. It assumes the curve is selected before the code is run. Figure 2 shows two lines selected in the sample Vertical Plate.idw.

Figure 2

Select the highlighted lines, then run the VBA code. The code first obtains the active sheet from the active document.

```
Sub CreateDimension()

    Dim oDoc As DrawingDocument
    Set oDoc = ThisApplication.ActiveDocument

    Dim oSheet As Sheet
    Set oSheet = oDoc.ActiveSheet
```

Next, obtain DrawingCurve objects from the selected lines. DrawingCurve objects are acceptable for creation of GeometryIntent objects.

```
    Dim oCurve1 As DrawingCurve
    Set oCurve1 = oDoc.SelectSet(1).Parent

    Dim oCurve2 As DrawingCurve
    Set oCurve2 = oDoc.SelectSet(2).Parent
```

Now we're ready to create the GeometryIntent objects. Since we're using the DrawingCurve end points, there is no need to specify a point on the geometry, which would be provided by the optional second argument for CreateGeometryIntent.

```
    Dim oIntent1 As GeometryIntent
    Set oIntent1 = oSheet.CreateGeometryIntent(oCurve1)

    Dim oIntent2 As GeometryIntent
    Set oIntent2 = oSheet.CreateGeometryIntent(oCurve2)
```

Create a 2D point for the location of the dimension line.

```
    Dim oPt As Point2d
    Set oPt = ThisApplication.TransientGeometry.CreatePoint2d(15, 15)
```

Create a linear dimension on the sheet, using the two GeometryIntent objects as dimension extension line origin points, and the 2D point for the dimension line location.

```
    Dim oLinDim As LinearGeneralDimension
    Set oLinDim = oSheet.DrawingDimensions.GeneralDimensions.AddLinear(oPt, oIntent1, oIntent2)

End Sub
```

The sample code will produce a result similar to that shown in Figure 3.

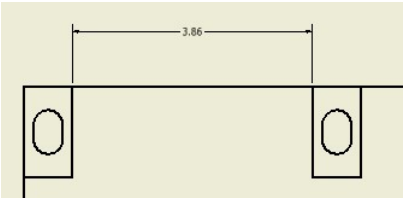


Figure 3

Balloons

Introduction to balloons

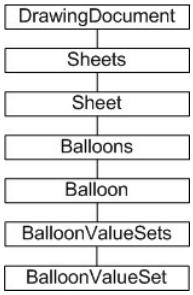
Autodesk Inventor users can add balloon notation to a drawing view or selection of parts automatically. These enumerated balloons are typically associated with a parts list, also generated automatically. Either the balloons or the list can have their values overridden, or new part numbers can be attached to custom parts. Balloons can be of various styles. The balloons and associated parts list can be updated to reflect the current content of the drawing view.

The purpose of balloons

Balloons and their associated parts list are a common means of calling out engineering requirements and specifications for manufacturing parts. Autodesk Inventor automates much of the process, including balloon placement. Balloon placement, type, and content can all be modified through the API.

Note: Balloon notation referencing properties (specified through styles) currently cannot be changed through the API.

Balloons Object Model Diagram



Balloons API structure

Balloons have content in the form of notation. This notation may be a part number, the date the part was created, the mass of the part, the author, and so on. It is typically a part number corresponding to items in a parts list.

The Balloon object references a BalloonValueSets collection comprised of BalloonValueSet objects. These BalloonValueSet objects also have a value and an override value, corresponding to the same functionality through the Autodesk Inventor user interface. The balloon type can be set through the SetBalloonType method of the balloon object. Balloons can reference an existing sketched symbol in place of the standard balloon shapes.

Using the balloon API

The following sample code makes changes to balloons on a drawing sheet, and so assumes an open drawing with balloons attached to parts. The code modifies the type and value of all the balloons. It omits error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

The first step is to obtain the active sheet in the drawing. The sample performs a count of the number of balloons and value sets, overriding the balloon values with these counts, so the balloon count is first initialized.

```
Dim oDrawDoc As DrawingDocument
Set oDrawDoc = ThisApplication.ActiveDocument

Dim oSheet As Sheet
Set oSheet = oDrawDoc.ActiveSheet

Dim BalloonCount As Long
BalloonCount = 1
```

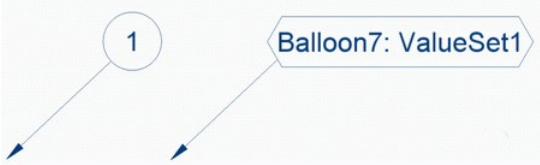
The intention is to change values in all balloons on the sheet, not just in a view. The following code iterates through the balloons collection, obtaining each balloon object in turn. The balloon type is changed to hexagonal, and the balloon placement direction is changed to kBottomDirection.

```
Dim oBalloon As Balloon

For Each oBalloon In oSheet.Balloons
    oBalloon.SetBalloonType (kHexagonBalloonType)
    oBalloon.PlacementDirection = kBottomDirection
```

Typically, a balloon has a single ValueSet object associated with it, but it's safe to iterate through the BalloonValueSets collection object. The code overrides each balloons value set or sets with a string, comprised of the balloon count and value set count values.

```
    Dim ValueSetCount As Long
    ValueSetCount = 1

    For Each oBalloonValueSet In oBalloon.BalloonValueSets
        oBalloonValueSet.OverrideValue = "Balloon" & _
           BalloonCount & ": ValueSet" & ValueSetCount
        ValueSetCount = ValueSetCount + 1
    Next
    BalloonCount = BalloonCount + 1
Next
```

In the following figure, the balloon on the right demonstrates the changes the preceding code applied to the balloon on the left.



Summary

Balloons are typically, though not exclusively, used as notation to identify parts in a drawing, with part numbers corresponding to items in a part list. The balloon API provides the means to modify balloon content, type, and position.

Also consider

Other types of notation available in the drawing sheet environment include drawing dimensions, drawing sketches, drawing text, and custom tables.

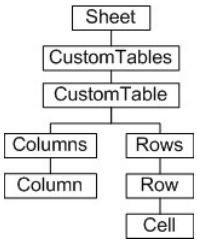
Custom Tables

Introduction to Custom Tables

Tables in Autodesk Inventor are typically associated with automated parts lists and bubble annotation. However, tables can also be considered general annotation, distinct from parts lists. Tables can be created with custom content. Also, custom tables can be assigned colors and specific line weights and text styles, and can be positioned anywhere on a drawing sheet.

Custom tables are designed so the developer can present and maintain tabulated data as annotation in a drawing document.

Custom Tables Object Model Diagram



Working with Custom Tables through the API

Custom tables are defined by specifying a number of rows and columns. Columns have header names which can optionally be part of the table. The following sample code adds a new table to the current sheet of an open drawing. The table will appear as follows.

Employees	
Desk Number	Employee
103	Peter Heald
107	Jamie Foss

Creating a Custom Table

The table is added to the active sheet, so first get this sheet from the active document. Note that this code omits error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

```
Public Sub CreateTable()

    Dim oDrawDoc As DrawingDocument
    Set oDrawDoc = ThisApplication.ActiveDocument

    Dim oSheet As Sheet
    Set oSheet = oDrawDoc.ActiveSheet
```

Set up a string array to contain the column titles (headers). This table has two columns, so the array contains two strings.

```
    Dim oTitles(1 To 2) As String
    oTitles(1) = "Desk Number"
    oTitles(2) = "Employee"
```

Now add the content by setting up an array for the two rows and two columns (2 x 2 = 4, so the array contains four strings). The table cell order is left-to-right, top-to-bottom.

Note: An empty table can be constructed by not passing content to the CustomTables.Add method, but if an array *is* passed, it must contain string entries for *all* cells, otherwise an error occurs.

```
    Dim oContents(1 To 4) As String
    oContents(1) = "103"
    oContents(2) = "Peter Heald"
```

```
    oContents(3) = "107"
    oContents(4) = "Jamie Foss"
```

The column width can be left at its default, to fit the content, but here the width is set a little wider for a neater appearance.

```
    Dim oColumnWidths(1 To 2) As Double
    oColumnWidths(1) = 3
    oColumnWidths(2) = 3
```

For this example the table is placed in an arbitrary position, at x=15 y=15.

```
    Dim InsP As Point2d
    Set InsP = ThisApplication.TransientGeometry.CreatePoint2d(15, 15)
```

Add the table to the sheet through the CustomTables.Add method, specifying a two-row, two-column table named "Employees". Pass the values defined previously.

```
    Dim oCustomTable As CustomTable
    Set oCustomTable = oSheet.CustomTables.Add("Employees", InsP, 2, 2, oTitles, oContents, oColumnWidths)
```

At this point the table appears on the sheet. However, we can continue to make changes to its appearance. The following code changes the justification of the second column to center-justified.

```
    oCustomTable.Columns.Item(2).ValueHorizontalJustification = kAlignTextCenter
```

To make format changes, set up a TableFormat object that can then be applied to the table.

```
    Dim oFormat As TableFormat
    Set oFormat = oSheet.CustomTables.CreateTableFormat
```

Change some properties of the TableFormat object (for example, the color of the inside grid lines, and the line weight of the inside and outside grid lines).

```
    oFormat.InsideLineColor = ThisApplication.TransientObjects.CreateColor(0, 0, 255)
    oFormat.InsideLineWeight = 0.1
    oFormat.OutsideLineWeight = 0.2
```

Apply the new format to the table. The line weights and color are updated.

```
    oCustomTable.OverrideFormat = oFormat
```

The following code demonstrates the ability to access values of individual cells based on their row and column intersection. Note that the column can be specified by its name.

```
    Dim oCell As Cell
    Set oCell = oCustomTable.Rows.Item(2).Item("Employee")

    Debug.Print oCell.Value

End Sub
```

The cell value "Jamie Foss" prints to the VBA "immediate" debug screen, if enabled.

Summary

The Custom Tables API allows tabulated data to be added to drawing sheets, complete with column headers and table formatting. Table grid line colors and weights can be specified, and cell contents can be accessed and modified.

Also consider

Autodesk Inventor has the ability to link to other documents through OLE. Add links to other documents through the Add method of the Document.ReferencedOLEFileDescriptors collection object. This gives developers the ability to add (for example) a Microsoft Excel file link to a drawing document.

Standard drawing sheet constructs such as a border and title block can be added to a sheet through the AddBorder, AddDefaultBorder and AddTitleBlock methods of the Sheet object.

ViewFrames - multiple monitor support

Introduction to ViewFrame

A ViewFrame is a window that can dock views and browser panes in it. Inventor has a built-in view frame which can be determined by ViewFrame.IsDefault, all other view frames can be generated by moving a view tab out of the view frame it docks to. Support multiple view frames allows users to deal with Inventor data in multiple monitors more easily. To dock or undock a view in a view frame is to move its tab(ViewTab), a ViewTab will always be in a ViewTabGroup, below snapshot demonstrates the view frames and view tabs:

When open or create a document, its graphical view will be displayed in the active ViewFrame, the Application.ActiveViewFrame tells which view frame is active. Activate a ViewFrame can be achieved by activating a view in it or dock a view into it. Below is a sample to change the active ViewFrame:

```
Sub ActivateViewFrameSample()
    ' This sample demonstrates how to activate a ViewFrame, close all custom ViewFrame windows before running it.
    ' When you open or new a document, the document's View will be located in the active ViewFrame.

    Dim oDefaultViewFrame As ViewFrame
    Set oDefaultViewFrame = ThisApplication.ViewFrames(1)

    Dim oDoc As PartDocument
    Set oDoc = ThisApplication.Documents.Add(kPartDocumentObject)

    ' This View of the new document is located in the default ViewFrame.
    Dim oView1 As View
    Set oView1 = oDoc.Views(1)

    Dim oViewTab1 As ViewTab
    Set oViewTab1 = oView1.ViewTab

    ' Create a new View for the same document.
    Dim oView2 As View
    Set oView2 = oDoc.Views.Add

    Dim oViewTab2 As ViewTab
    Set oViewTab2 = oView2.ViewTab

    ' Move the second View to generate a new custom ViewFrame, this will also activate the new ViewFrame.
    Dim oViewFrame1 As ViewFrame
    Set oViewFrame1 = oViewTab2.MoveToNewViewFrame(500, 600, 200, 100)
    Debug.Print ThisApplication.ActiveViewFrame Is oViewFrame1

    ' Add a new document, now it will be opened in the active custom ViewFrame.
    Dim oDoc1 As PartDocument
    Set oDoc1 = ThisApplication.Documents.Add(kPartDocumentObject)

    ' Activate the default ViewFrame through activating a View in it.
    oView1.Activate
    Debug.Print ThisApplication.ActiveViewFrame Is oDefaultViewFrame

    ' Now create a new document will be located in the active default ViewFrame.
    Dim oDoc2 As PartDocument
    Set oDoc2 = ThisApplication.Documents.Add(kPartDocumentObject)
End Sub
```

ViewFrame can be resized by changing its Width or Height directly, or using ViewFrame.Move function. The ViewFrame.Move can also change the ViewFrame's position on screen(can be in different monitors that connect with the same computer). The ViewFrame.Arrange can arrange views in it as tiles.

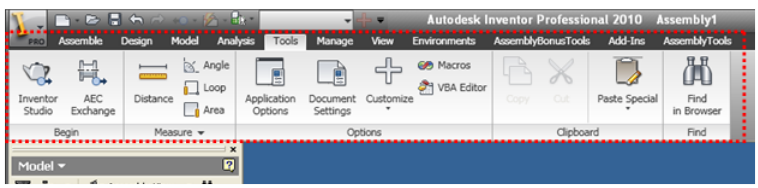
Customizing the Ribbon using the API

Ribbon Terminology and the API

Here's a high level look at the components of the ribbon and the corresponding API objects.

Ribbon

The ribbon is the entire area highlighted below and is represented by the Ribbon API object.

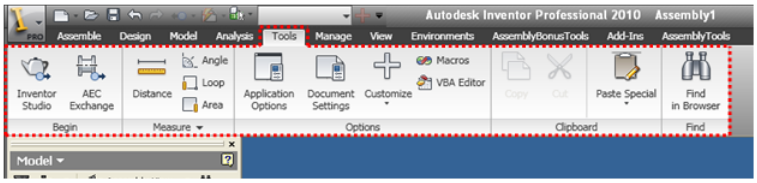


Internally there are seven different ribbons, primarily one for each document type. You can't create or delete ribbons but you can edit the seven existing ribbons by adding and removing elements from them. You typically access a specific ribbon through the API using the ribbon's internal name. The internal names for the ribbons are:

- ZeroDoc (Displayed when there aren't any documents open)
- Part
- Assembly
- Drawing
- Presentation
- iFeatures
- UnknownDocument (Used for notebook and drawing view orientation environments.)

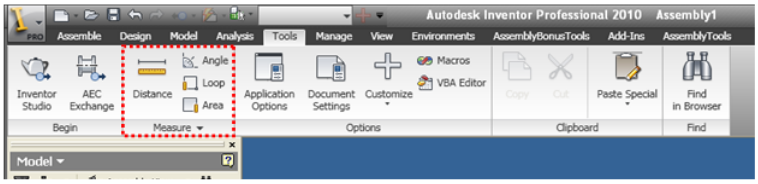
Ribbon Tab

Each ribbon consists of a set of tabs. The Tools tab of the Assembly ribbon is highlighted below. These are represented in the API by the RibbonTab object. There can be any number of tabs, although there's a practical limit to what will fit on the screen. Tabs can be visible or not. Each tab also has an internal name that can be used to find it.



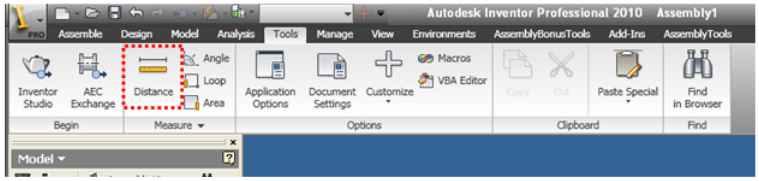
Ribbon Panel

Each tab consists of a set of panels. The Measure panel of the Tools tab is highlighted below. These are represented in the API by the RibbonPanel object. There can be any number of panels in a tab, again with a practical limit, and they can be visible or not. Each panel also has an internal name that can be used to find it.



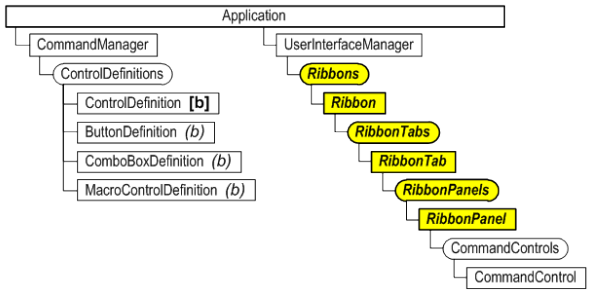
Command Control

Each tab contains a set of controls. The button control for the Distance command is highlighted below. All controls are represented in the API by the CommandControl object. Each control also has an internal name that can be used to find it; however their internal name isn't explicitly defined but is inherited from the ControlDefinition object it references.



Ribbon API Object Hierarchy

Below is the object hierarchy for the portion of Inventor's API that works with the ribbon. The highlighted items are the objects specific to the ribbon. The others are objects that are also used in other areas of the API.



Determining Where to Add Your Buttons

The first step in supporting the ribbon is deciding where you want to insert your add-in's buttons into the ribbon. A suggestion is to try and put yourself in the position of the user of your add-in and imagine where they will most likely look for your commands. For example, if your add-in has a command that draws a custom slot shape in a sketch you would probably want to insert your button on the Draw panel of the Sketch tab of the Part, Assembly, and Drawing ribbons. If your command is unique from the rest of Inventor's commands and doesn't really fit in you might want to create a new tab or a panel to logically separate it from the rest of Inventor's commands. In addition to the ribbon you can also add your commands to the Application and Help menus and the QAT (Quick Access Toolbar). The main objective is to have your button located where users will intuitively look for it.

Another factor that you need to consider when positioning your commands is the overall layout of the ribbon once your command is added. The reality is that there is a limited width to the screen and each button takes some space. Some ribbons are fuller than others so if logically your button could go in more than one location you might want to choose the one that has the most room available.

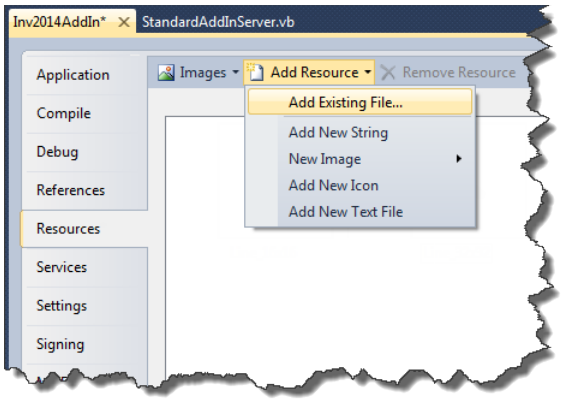
Getting the Internal Names of the Ribbon Components

It was previously mentioned that the various components of the ribbon have internal names that can be used to find specific ribbon components. How do you know what the internal names are? The best answer to this question is one of the sample programs that generates a list of the contents of all of the ribbons. This provides and always up to date listing of the names of all the components in the current configuration of the ribbon.

Creating Your Icons

The ribbon supports two sizes of icons; the small icons are16x16 pixels and the large icons are 32x32 pixels. You have different options for creating your icons. Probably the best format is png because it supports transparency and is supported by most image editing applications. You can also use .ico files, which also support transparency, or even use .bmp files.

Your images can be added as resources to your add-in project using the Resources tab in the Properties dialog for your project, as shown below. Choose the "Add Existing File…" option in the "Add Resource" dropdown to select your files and add them as resources.



Adding Ribbon Support to Your Add-In

The first thing you need to do is create the control definitions for the controls you want to add to the ribbon. Typically these are buttons. Below is some typical code from the Activate method of an add-in that creates the control definitions. It's good to remember that the control definitions aren't the visible controls themselves but define all of the information that's needed besides the location to create a control. The code within the CreateUserInterface function creates the visible controls using the control definitions. The CreateUserInterface function is only called when the FirstTime flag is True. Currently with the ribbon user-interface it will always be true.

For the following code to work you'll need to add references to the .Net stdole and System.Windows.Forms libraries.

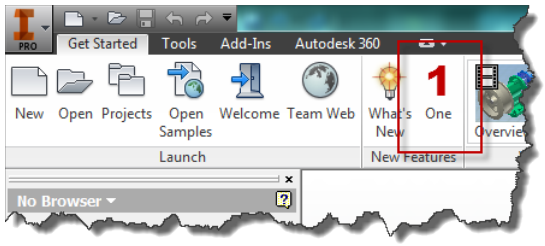
```
Private m_InventorApplication As Inventor.Application = Nothing
Private WithEvents m_buttonDef As ButtonDefinition = Nothing
Private WithEvents m_uiEvents As UserInterfaceEvents = Nothing
Private m_clientID As String = "{311a4c02-49df-4947-a01c-47765ec06b27}"

Public Sub Activate(...) Implements Inventor.ApplicationAddInServer.Activate
    ' Save reference to the Application object in member variable.
    m_inventorApplication = addInSiteObject.Application

    ' Get a reference to the UserInterfaceManager object.
    Dim UIManager As Inventor.UserInterfaceManager = _
                                        m_inventorApplication.UserInterfaceManager

    ' Get a reference to the ControlDefinitions object.
    Dim controlDefs As ControlDefinitions = _
                            m_inventorApplication.CommandManager.ControlDefinitions

    ' Get the images from the resources.  They are stored as .Net images and the
    ' PictureConverter class is used to convert them to IPictureDisp objects, which
    ' the Inventor API requires.
    Dim smallPicture As stdole.IPictureDisp = _
            PictureConverter.ImageToPictureDisp(My.Resources.MySmallImage)

    Dim largePicture As stdole.IPictureDisp = _
            PictureConverter.ImageToPictureDisp(My.Resources.MyLargeImage)

    ' Create the button definition.
    m_buttonDef = controlDefs.AddButtonDefinition("One", "UIRibbonSampleOne", _
                                        CommandTypesEnum.kNonShapeEditCmdType, _
                                        m_clientID, , , smallPicture, largePicture)

    ' Call the function to add information to the user-interface.
    If firstTime Then
        CreateUserInterface()
    End If

    ' Connect to UI events to be able to handle a UI reset.
    m_uiEvents = m_InventorApplication.UserInterfaceManager.UserInterfaceEvents
End Sub


' Creates the add-in's UI in the ribbon.
Private Sub CreateUserInterface()
    ' Get a reference to the UserInterfaceManager object.
    Dim UIManager As Inventor.UserInterfaceManager = _
```

```
                                        m_inventorApplication.UserInterfaceManager

    ' Get the zero doc ribbon.
    Dim zeroRibbon As Inventor.Ribbon = UIManager.Ribbons.Item("ZeroDoc")

    ' Get the getting started tab.
    Dim startedTab As Inventor.RibbonTab = zeroRibbon.RibbonTabs.Item("id_GetStarted")

    ' Get the new features panel.
    Dim newFeaturesPanel As Inventor.RibbonPanel
    newFeaturesPanel = startedTab.RibbonPanels.Item("id_Panel_GetStartedWhatsNew")

    ' Add a button to the panel, using the previously created button definition.
    newFeaturesPanel.CommandControls.AddButton(m_buttonDef, True)
End Sub
```

Outside of the StandardAddInServer class, add the class below. This class wraps some functionality available in one of the .Net classes that's not typically exposed that allows you to convert a .Net Image object to an IPictureDisp object, which is what Inventor requires when working with images.

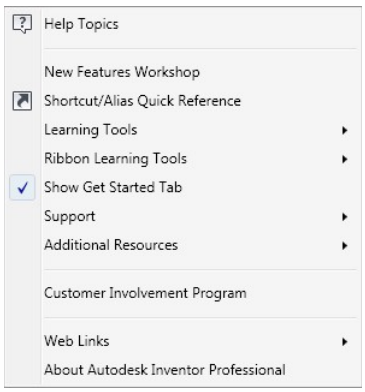
```
<System.ComponentModel.DesignerCategory("")> _
Friend Class PictureConverter
    Inherits System.Windows.Forms.AxHost

    Private Sub New()
        MyBase.New(String.Empty)
    End Sub

    Public Shared Function ImageToPictureDisp( _
                            ByVal image As System.Drawing.Image) As stdole.IPictureDisp
        Return CType(GetIPictureDispFromPicture(image), stdole.IPictureDisp)
    End Function
End Class
```

The result of loading this add-in is shown below.



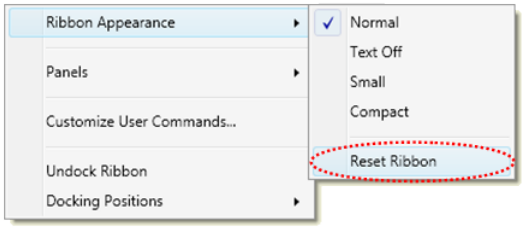
Separators

Separators are also supported by the ribbon although they're not as common since the panels serve the same purpose of grouping commands. They're most common in the Application and Help menus. The picture below shows the help menu in the ribbon with its three separators. To create a separator you use the AddSeparator method of the CommandControls object.



Handling Resets in the Ribbon Interface

The user can reset the ribbon interface to restore it to its original state. This has the side effect of removing any customization that add-ins have done. Most users will expect a reset to take Inventor back to its initial interface plus any customization that add-ins have done to add their buttons. To accomplish this, an add-in needs to react to a reset by recreating whatever customization it did. An add-in does this by listening to events. In the ribbon interface the user can use the Reset Ribbon command to reset the entire ribbon interface back to its initial state. The Reset Ribbon command is invoked through the ribbon's context menu as shown below.



For an add-in to handle the Reset Ribbon command it needs to listen to the OnResetRibbonInterface event. All you need to do in response to this event is exactly what you did when the add-in was executed for the first time. The code below illustrates this by calling the same function that was called in the Activate method of the add-in.

```
Private Sub m_uiEvents_OnResetRibbonInterface(...) Handles m_uiEvents.OnResetRibbonInterface
    CreateUserInterface()
End Sub
```

User Interaction

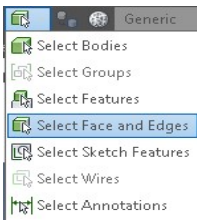
Introduction

An important requirement of many programs is the ability to interact with the end user. There are several types of user interaction that your program might need to do. The first, which is independent of Autodesk Inventor, is interaction through a dialog that you create. For example, an NC Application might display a dialog to allow the user to define tool information. This dialog is created and controlled using whatever dialog creation tools are provided by the programming language you've chosen. This section discusses the other types of user interaction that are Autodesk Inventor specific, including selecting entities, mouse input, displaying messages, and highlighting graphics.

One of the most common requirements regarding user interaction is the ability to have the user select an entity. Autodesk Inventor supports two techniques for entity selection: the select set and interactive selection. Each method is useful for certain cases and in many programs both methods will be used. Both methods are described in more detail in the rest of this section.

The Select Set

Using the select set is the easiest but also the least flexible method of selection. The select set is the current set of entities the user has selected using Autodesk Inventor's Select command. The Select command is the command that is active in Autodesk Inventor when no other command is running. It can be explicitly started by the end-user by clicking the "Select" button on the Command Bar, as shown below. The drop-down list for the Select command allows the end-user to set the priority filter for what type of entities will be selected first.



Use of the Select command by the user is entirely independent of the API. As a programmer, you're able to use the API to look at current contents of the select set to see what the user has already selected. Commands that use the select set work in an object-action manner. This means that the user first selects appropriate objects and then runs the command that utilizes what's in the select set. Some of Autodesk Inventor's commands behave like this. The Delete command is one example. You first select the entities you want to delete and then run the Delete command. The Fillet command is another example. If you have any edges selected when you run the Fillet command the fillet dialog will initialize with those edges already selected.

Use of the select set from the programmer's point of view is extremely easy since you're not involved in the selection process itself. During the selection process it's left up to the user to understand what needs to be selected before running your command or program. Your program just looks at the results, checks to see if any valid entities have been selected, and uses them if they are valid. The SelectSet object is available from all documents via the SelectSet property. The sample below illustrates a program that checks to see if a single face has been selected and displays its surface area.

```
Public Sub ShowSurfaceArea()
    ' Set a reference to the select set of the active document.
    Dim oSelectSet As SelectSet
    Set oSelectSet = ThisApplication.ActiveDocument.SelectSet

    ' Check to make sure a single item was selected.
    If oSelectSet.Count = 1 Then
        ' Check to make sure a face was selected.
        If TypeOf oSelectSet.Item(1) Is Face Then
            ' Set a reference to the selected face.
            Dim oFace As Face
            Set oFace = oSelectSet.Item(1)

            ' Display the area of the selected face.
            MsgBox "Surface area: " & oFace.Evaluator.Area & " cm^2"
            Exit Sub
        Else
            MsgBox "You must select a single face."
            Exit Sub
```

```
        End If
    Else
        MsgBox "You must select a single face."
        Exit Sub
    End If
End Sub
```

In addition to providing access to the objects the user has selected, the SelectSet object also supports methods that allow you to add and remove objects from the select set.

Besides being the method of choice for programs that need to have object-action behavior, the select set is often used for other programs because it's relatively easy to implement. However, you're trading ease of use on the programming side for a usually harder to use interface for the end-user. For example, in the previous sample the select set isn't necessarily the best method for having the end-user select a face, since they have to know before they run your command what's needed. Because there is no control, end-users can also select invalid entities. Using interactive selection, which is discussed next, you can control the selection so only valid items are selected. Even though the select set isn't the optimal choice in many cases, it is useful in many cases and its speed of implementation can often offset its limitations. It is extremely useful when writing small test programs that need to obtain a specific entity.

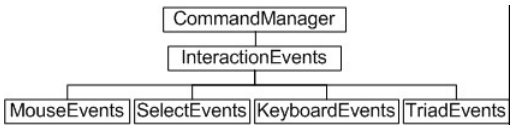
Interactive Selection

Many commands are easier to use if the selection process is more controlled than what is possible when using the select set. Autodesk Inventor supports another method of entity selection that provides you with extensive control over the selection process. This capability is exposed through the InteractionEvents object. This object not only supports selection, but also mouse and keyboard events. We'll focus this discussion initially on the selection capabilities of the InteractionEvents object.

What makes the interactive selection capability so powerful is that it allows you to participate in the selection process. It does this by providing a series of events to notify you about what's currently happening and allows you to control what the user sees. If you look at the behavior of Autodesk Inventor commands you'll notice that they each have unique selection behavior that helps the user in selecting only those entities appropriate for the current task. For example, when the Fillet command is initially invoked you can only select edges of the model; other entities (faces, work features, etc.) are not selectable. Another thing to notice about the Fillet command is that when selecting edges, any edges that are tangentially connected to the selected edge are also automatically selected.

Here's a brief overview of the objects and the steps required to make use of the interaction events functionality. Following the overview, we'll look at an example using the InteractionEvents object that duplicates the Fillet command behavior described earlier.

The InteractionEvents portion of the object hierarchy is shown below. For selection, the objects used are the InteractionEvents and SelectEvents objects.



This section does not cover TriadEvents. Please refer to the TriadEvents overview.

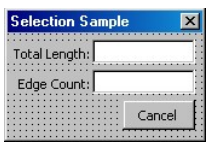
The following is a brief overview of the steps to use the interactive selection functionality.

- Create an InteractionEvents object
- Define its behavior by setting properties
- Connect to events supported by the InteractionEvents object
- Connect to events supported by the associated SelectEvents object
- Start the interaction process and respond to the events.

Let's look at the steps involved in implementing edge-picking behavior that is similar to that used in the Fillet command. This simple command will allow you to prompt the user to select edges and will show the length of the edge as it is selected. The first step is to create an InteractionEvents object using the CreateInteractionEvents method of the CommandManager object. The next step is to set up the various objects by connecting to the events of interest and set the various properties to get the desired behavior. There are events on the InteractionEvents object and also for the SelectEvents, MouseEvents, and KeyboardEvents objects that are obtained from the InteractionEvents object.

Once the behavior has been defined using the events and methods of the various objects you launch the selection process by calling the Start method of the InteractionEvents object. An important concept to understand is that when the InteractionEvents object is started it causes the same side effect as Autodesk Inventor commands do: it will terminate the command that is currently running. This also implies that if an Autodesk Inventor command is started when the InteractionEvents object is running, the InteractionEvents object will be terminated. The exception to this is when view commands are run. They don't terminate the current command but temporarily suspend it until the view command is finished. We'll see later how the InteractionEvents object provides the information you need to handle these situations correctly.

For this example, we'll assume you're using VBA, although the implementation in VB would be almost identical. To create a running version of this example, create a new form module within any VBA project, as shown below.



The form consists of five controls: two text boxes, two labels, and a command control. The form is named frmSelection. The text box for the length is named txtLength, the text box for the edge count is named txtEdgeCount, and the command control is named cmdCancel. The names of the label controls don't matter. The following are the global declarations within the form module and the code from the Initialize event of the form that obtains the necessary objects and sets them up for the selection process.

```
Private WithEvents oInteraction As InteractionEvents
Private WithEvents oSelect As SelectEvents

Private Sub UserForm_Initialize()
    ' Create a new InteractionEvents object.
    Set oInteraction = ThisApplication.CommandManager.CreateInteractionEvents

    ' Set the prompt.
    oInteraction.StatusBarText = "Select an edge."

    ' Connect to the associated select events.
    Set oSelect = oInteraction.SelectEvents

    ' Define that all part edges should be selectable.
    oSelect.AddSelectionFilter kPartEdgeFilter

    ' Enable single selection.
    oSelect.SingleSelectEnabled = True

    ' Start the selection process.
    oInteraction.Start
End Sub
```

Notice that the global variables for the InteractionEvents and SelectEvents objects use the WithEvents keyword. This allows us to set up event handlers that will receive the events associated with these objects. Variables that are declared using the WithEvents keyword now show up in the Object List (the pull-down at the top-left of the code window). When you select one of the objects in this list, the associated events are listed in the Events List (the pull-down at the top-right of the code window).

In this sample, the Initialize event of the form is used to trigger the setup and start of the selection process. Two of the more interesting lines of code are the ones calling the AddSelectionFilter and Start methods. The AddSelectionFilter method can be called multiple times, specifying a single filter each time. Each filter specifies a type of entity that you want to allow for selection. The filters are pre-defined by Autodesk Inventor and are fairly broad in the types of entities they describe. As we'll see later you can use the events fired by the SelectEvents object to perform runtime filtering using any criteria you choose. Setting the SingleSelectEnabled property sets the behavior so that only a single entity can be selected at a time. If an entity is already selected it will be unselected when you select another entity. Calling the Start method causes any current Autodesk Inventor command to be aborted and your selection within Autodesk Inventor to begin.

To use this code, we need the form to be displayed in a modeless manner, so you can leave the form displayed and still interact with Autodesk Inventor to do the selection. The following function should be added to a standard code module to display the form. It's this function that you'll run to start this sample program.

```
Public Sub SelectionSample()
    frmSelection.Show vbModeless
End Sub
```

You can run the program now by executing the SelectionSample sub. First, make sure you have a part document open that contains a model. When you run the SelectionSample sub, the current Autodesk Inventor command will be aborted and you will be able to select edges of the model. Any other entities, such as faces, work geometry, sketches, etc., are not selectable. Because of the SingleSelectEnabled property, as you continue to select edges the previously selected edge is unselected. This demonstrates that you can control the filtering and initiate the selection process, but it isn't very useful because we're not getting or doing anything with the selected entities. We'll add code now to get the entities just selected and display their length. To do this we need to implement OnSelect event of the SelectEvents object, as shown below.

```
Private Sub oSelect_OnSelect(ByVal JustSelectedEntities As ObjectsEnumerator, _
                             ByVal SelectionDevice As SelectionDeviceEnum, _
                             ByVal ModelPosition As Point, _
                             ByVal ViewPosition As Point2d, _
                             ByVal View As View)
    ' Calculate the length of the edge(s) selected.
    Dim i As Long
    Dim dLength As Double
    For i = 1 To JustSelectedEntities.Count
        ' Since we set the filter to only select edges it's safe to assign
        ' the returned entities to an Edge object.
        Dim oEdge As Edge
        Set oEdge = JustSelectedEntities.Item(i)

        ' Determine the length of the current edge.
        Dim dMin As Double
        Dim dMax As Double
        Call oEdge.Evaluator.GetParamExtents(dMin, dMax)

        Dim dSingleLength As Double
        Call oEdge.Evaluator.GetLengthAtParam(dMin, dMax, dSingleLength)

        ' Add up the length of all the edges in this set.
        dLength = dLength + dSingleLength
    Next

    ' Display the length and number of the edges.
    txtLength.Text = Format(dLength, "0.0000 cm")
    txtEdgeCount.Text = JustSelectedEntities.Count
End Sub
```

Autodesk Inventor fires the OnSelect whenever the user selects an entity. The entity selected is provided in the JustSelectedEntities argument. This argument is an ObjectEnumerator but in our sample the returned ObjectsEnumerator object will always contain a single entity. We'll see in a minute a case when this contains multiple entities. The other arguments provide additional information about the selection. If you run the program now you should see the length displayed for the single selected edge.

Now, let's add an enhancement to the program so that edges that are tangentially connected will be selected as one. This demonstrates a powerful feature of the SelectEvents, which is the ability to programmatically control the entities available for selection. This is how you can define your own filters and group entities together for selection. The primary component in this is the OnPreSelect event of the SelectEvents object.

When selecting entities in Autodesk Inventor, entities that are valid for selection change to the highlight color as the mouse passes over them to indicate they're available to be selected. When the user actually selects an entity, it changes to the select color. The OnPreSelect event is fired whenever the user moves the mouse over an entity that meets the filter criteria we defined using the AddSelectionFilter method. The important point here is that the entity has not been highlighted yet. You now have the ability to examine the entity and determine if it should be made available for selection or not. In addition to determining if the current entity is selectable, you can add additional entities to the selection so the entire group will highlight in the pre-select color and be selected should the user click the mouse.

The following code takes the input edge and checks to see if there are any tangentially connected edges. If so, it adds these edges to the set to be highlighted. When you run the sample after adding this code, any tangentially connected edges will highlight and select together. In this case, the ObjectCollection passed in through the JustSelectedEntities argument of the OnSelect event will contain all of the edges selected.

```
Private Sub oSelect_OnPreSelect(PreSelectEntity As Object, _
                                DoHighlight As Boolean, _
                                MorePreSelectEntities As ObjectCollection, _
                                ByVal SelectionDevice As SelectionDeviceEnum, _
                                ByVal ModelPosition As Point, _
                                ByVal ViewPosition As Point2d, _
                                ByVal View As View)
    ' Set a reference to the object the mouse is currently over.
    ' We know it's an edge because of the filtering previously defined.
    Dim oEdge As Edge
    Set oEdge = PreSelectEntity

    ' Determine if there are any tangentially connected edges.
    Dim oEdges As EdgeCollection
    Set oEdges = oEdge.TangentiallyConnectedEdges
    If oEdges.Count > 1 Then
        ' Build up the object collection containing the additional edges.
        Set MorePreSelectEntities = _
                        ThisApplication.TransientObjects.CreateObjectCollection
        Dim i As Long
        For i = 1 To oEdges.Count
            If Not oEdges.Item(i) Is PreSelectEntity Then
                MorePreSelectEntities.Add oEdges.Item(i)
            End If
        Next
    End If
End Sub
```

The basic behavior of the SelectEvents object is to remain in selection mode, allowing the user to continue selecting entities until it is explicitly stopped. It can be stopped by your program telling it to stop, or by the user aborting it by pressing the escape key or running another Autodesk Inventor command. For example, if you have a command that requires the user to select faces and then performs some operation with all of the selected faces, you might start your command with the select enabled and filtered to select only faces. Your dialog can contain some method for users to specify when they're finished selecting faces so you can perform the desired operation.

In our example, the Cancel button is the signal to stop selection. In this sample, in addition to stopping the selection it also terminates the command and dismisses the dialog. Another action that can cause our command to stop is when the user runs another command or presses the escape key. In this case Autodesk Inventor fires the OnTerminate event to notify you that your interaction event is being terminated. Both of these are handled in the code below.

```
Private Sub cmdCancel_Click()
    ' Stop the selection and release any global references.
    oInteraction.Stop
    Set oSelect = Nothing
    Set oInteraction = Nothing

    ' Unload the form.
    Unload Me
End Sub

Private Sub oInteraction_OnTerminate()
    ' Release any global references.
    Set oSelect = Nothing
    Set oInteraction = Nothing

    ' Unload the form.
    Unload Me
End Sub
```

In the previous discussion we've seen how the SelectEvents object supports the selection of a single entity at a time when the SingleSelectEnabled property is set to True. There are also many cases when you want the user to select multiple entities. Setting the SingleSelectEnabled to False will allow this. There are two things to be aware of in this case. First, you may want to implement the OnUnselect event to handle the case where the user unselects a previously selected entity. Second, the JustSelectedEntities argument of the OnSelect event will only contain the entity(s) just selected. To obtain all of the entities currently selected you can use the SelectedEntities property of the SelectEvents object.

```
    ' Clear the current selection.
    oSelect.ResetSelections
```

There are also several other events supported by the SelectEvents object that weren't used in this sample. The OnPreSelectMouseMove fires events as the user moves the mouse over an entity that is valid for selection. The primary use of this event is to capture the current position of the mouse relative to the selected entity as the mouse moves across it. For example, a Finite Element Modeling system might have a command that allows the user to position a load on a face. Using this event you can track the mouse and dynamically show the load on the face as it follows the mouse.

The OnStopPreSelect event works in conjunction with the OnPreSelectMouseMove. This event notifies you when the user moves the mouse off off the model and into space. This way you know when to stop displaying any preview graphics.

The OnUnSelect event notifies you when the user has unselected an entity that was already selected. They can do this by holding down the Shift or Control key while selecting entities.

The InteractionEvents object also supports some events that we didn't use here. They are the OnActivate, OnSuspend, and OnResume events. The OnActivate event is fired whenever the Start method of the InteractionEvents object if called. This is useful in cases where you may be storing information about the selection. The OnActivate event is a convenient location to initialize your private set of data. The OnSuspend and OnResume events are used in conjunction to notify you when a stackable Autodesk Inventor command has been started. Stackable commands are commands that don't require your command to terminate, but can temporarily take over and then allow your command to resume. Commands of this type do not change the contents of the file. The most they can do is change the view orientation. Currently the only stackable commands are the various view commands: Zoom All, Pan, Rotate, etc. The OnSuspend event notifies you that a stackable command has been executed. You might choose to hide your dialog at his point. The OnResume notifies you that the stackable command is finished and you are now back in control.

So far we've seen the power of the SelectEvents, but we've also seen that it's not as easy to implement as the select set. A common task in many simple programs is to have the user select a single object. The code below wraps the functionality of the InteractionEvents and SelectEvents within a class module to make this fairly easy to implement within any VBA program. The sample below reproduces the face area sample that was used to illustrate the select set.

Below is the primary code. This can exist within a standard code module, a form, or another class. The only selection-related code here is the declaration and creation of an object of the clsSelect class and the call of this object's Pick method. The Pick method takes a single argument that defines the filter and returns the selected face. For the user this is easier to use than the previous select set sample because it enforces that only faces are selected.

```
Public Sub ShowSurfaceArea2()
    ' Declare a variable and create a new instance of the select class.
    Dim oSelect As New clsSelect

    ' Call the Pick method of the clsSelect object and set
    ' the filter to pick any face.
    Dim oFace As Face
    Set oFace = oSelect.Pick(kPartFaceFilter)

    ' Check to make sure a face was selected.
    If Not oFace Is Nothing Then
        ' Display the area of the selected face.
        MsgBox "Surface area: " & oFace.Evaluator.Area & " cm^2"
    End If
End Sub
```

The code below does all of the work. To use this create a Class module named clsSelect and add the code to it.

```
' Declare the event objects
Private WithEvents oInteraction As InteractionEvents
Private WithEvents oSelect As SelectEvents

' Declare a flag that's used to determine when selection stops.
Private bStillSelecting As Boolean

Public Function Pick(filter As SelectionFilterEnum) As Object
    ' Initialize flag.
    bStillSelecting = True

    ' Create an InteractionEvents object.
    Set oInteraction = ThisApplication.CommandManager.CreateInteractionEvents

    ' Define that we want select events rather than mouse events.
    oInteraction.SelectionActive = True

    ' Set a reference to the select events.
    Set oSelect = oInteraction.SelectEvents

    ' Set the filter using the value passed in.
    oSelect.AddSelectionFilter filter

    ' The InteractionEvents object.
    oInteraction.Start

    ' Loop until a selection is made.
    Do While bStillSelecting
        DoEvents
    Loop

    ' Get the selected item.  If more than one thing was selected,
    ' just get the first item and ignore the rest.
    Dim oSelectedEnts As ObjectsEnumerator
    Set oSelectedEnts = oSelect.SelectedEntities
    If oSelectedEnts.Count > 0 Then
        Set Pick = oSelectedEnts.Item(1)
    Else
        Set Pick = Nothing
    End If

    ' Stop the InteractionEvents object.
    oInteraction.Stop

    ' Clean up.
    Set oSelect = Nothing
    Set oInteraction = Nothing
End Function


Private Sub oInteraction_OnTerminate()
    ' Set the flag to indicate we're done.
    bStillSelecting = False
End Sub


Private Sub oSelect_OnSelect(ByVal JustSelectedEntities As ObjectsEnumerator, _
                             ByVal SelectionDevice As SelectionDeviceEnum, _
                             ByVal ModelPosition As Point, _
                             ByVal ViewPosition As Point2d, _
                             ByVal View As View)
    ' Set the flag to indicate we're done.
    bStillSelecting = False
End Sub
```

Mouse and Keyboard Events

Mouse Events

As mentioned earlier, the InteractionEvents object provides access to mouse and keyboard events in addition to the select events. Listening to mouse events is very similar to listening to select events but instead of using the SelectEvents object and its related events you use the MouseEvents object and its events.

The events supported by the MouseEvents object are fairly straightforward and are very similar to the mouse events available for VB/VBA forms. Using these events you can receive notification that mouse moves or a button is clicked and the coordinates, both model and view, where this occurred.

Some properties of the MouseEvents object probably deserve a little more description. The MouseMoveEnabled property allows you to control whether the OnMouseMove event is fired or not. In some cases there may be certain times when you want the OnMouseMove event. When you don't need it, you can turn it off. Because this particular event is fired so frequently, turning it off when it's not needed can improve the performance of your application.

The PointInferenceEnabled and PointInferences properties of the MouseEvents object are used within the sketch environment. If you set the PointInferenceEnabled property to True, then as you move the mouse it will infer its position relative to existing entities in the sketch. This is the same type of inference you see when you create sketch entities within Autodesk Inventor. As you move the mouse so it is close to the end point of an entity, the cursor indicates this inference. As you move so the cursor is close to horizontal or vertical to a point or end of an entity it infers this relationship. The point inference of the MouseEvents object performs this same inference and then allows you to find out what inferences were implied by examining the contents of the PointInferenceEnumerator returned by the PointInferences property.

Keyboard Events

The KeyboardEvents object is also obtained from the InteractionEvents object. Keyboard events can be listened to in conjunction with the mouse and select events. They're very similar to the keyboard events for VB/VBA forms and controls. The KeyboardEvents object supports the OnKeyDown, OnKeyUp, and OnKeyPress events and provides information about which key the event occurred for.

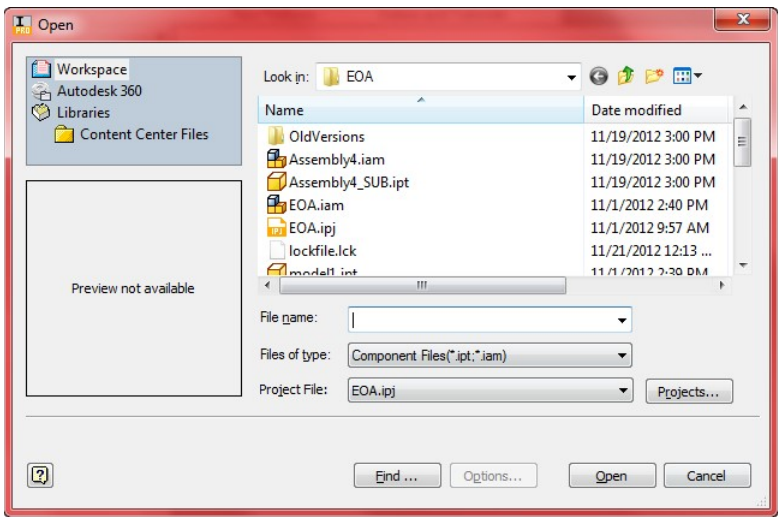
Status Bar Text

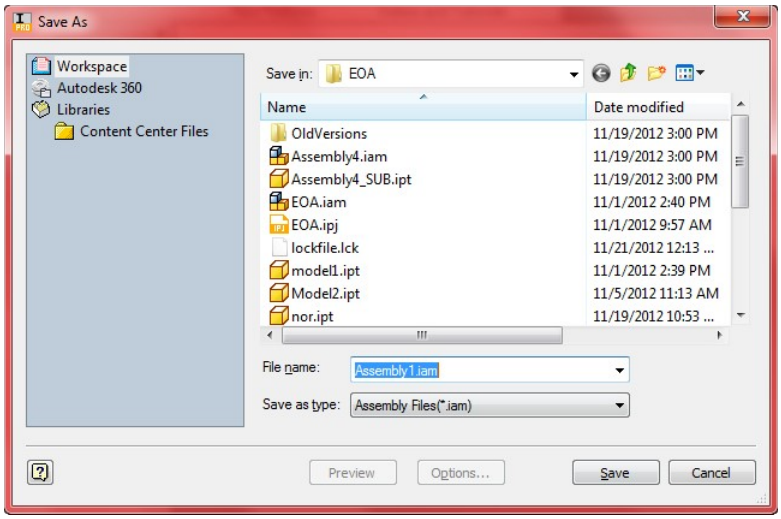
The final user interaction capability in Autodesk Inventor is the ability to display prompts to the user in the status bar area. This is the area at the bottom of the Autodesk Inventor main window and the message area on the command bar. Support for this is provided by the StatusBarText property of the Application and the StatusBarText property of the InteractionEvents object. Both of these are read-write properties that allow you to get and set the text currently shown in the status bar. The StatusBarText property on the Application object can be accessed and set at anytime. However, you don't have any control over how long the text remains displayed. Usually as the user moves the mouse across the screen Inventor displays messages in this area and they would overwrite your message. The StatusBarText property of the Application object is typically used to display the status of a process that is running so the end-user isn't interacting with Inventor.

The StatusBarText property of the InteractionEvents object solves this problem. The text assigned to this property will be displayed within the status bar while your InteractionEvents object is running. The InteractionEvents object will maintain the display of the text even and the user moves the mouse around to other areas of the screen. In this case, the status bar text is typically used to provide instructions to the end user.

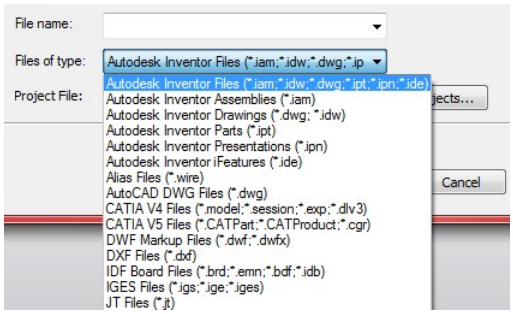
File Dialogs

The File Dialogs portion of the API provides the functionality for developers to reuse the standard Autodesk Inventor Open and Save As dialogs as shown below. Using the API, the developer can define some of the behavior of the dialog and then displays the dialog to the user. The user interacts with the dialog to specify a file name to open or save. The developer is provided the selected file. The use of this API does not actually perform the Open or Save but only obtains a file name from the user. The developer can then use this file name in any way. This functionality is very similar to the Microsoft common file control.





The Files of type portion of the dialog is controllable through the API to allow the developer to set the filter for specific types of files.



Triad Events

Introduction to TriadEvents - the 3D move and rotate tool

The Autodesk Inventor user interface has a 3D move and rotate tool, available when you create or edit a grounded work point in a part or 3D sketch. A triad symbol is displayed. Select or drag a triad segment to indicate the type of transform required, or click it to open a dialog box. In the dialog box, you can enter coordinates to precisely position a grounded work point. In addition to entering coordinates, you can drag the triad along the X, Y, and Z coordinates to dynamically update the dialog box. Movement is not limited to the part axes. The 3D Move/Rotate tool can be rotated about any axis to realign the direction of subsequent moves.

The Autodesk Inventor API supports the 3D Move/Rotate tool through the TriadEvents object.

The purpose of TriadEvents

The ability of the developer to control the triad affords many advantages to the user. An example is the routing of piping. As each segment is placed, the triad can be positioned at the end of the pipe so the user can indicate, graphically but precisely, the axis of the next segment. The user can do this using the triad tool - something already familiar through the user interface. In addition to programmatically positioning the triad, the developer can receive notification of a number of triad events - for example, if the user interactively moves the triad.

TriadEvents Object Model Diagram

Working with the TriadEvents API

The TriadEvents object is used in a similar manner to other interaction events such as mouse and keyboard events. It is obtained through the TriadEvents property of the InteractionEvents object.

Selection and other interaction events cannot be active while triad events are enabled. Start the InteractionEvents object and set its InteractionDisabled property to True to start receiving triad events.

You can position the triad by calling the Reposition method of the TriadEvent object.

You can register a number of triad events, including the following:

OnActivate - Fires when the triad is activated.
OnEndSequence - Fires when the user ends a move sequence of the triad.

OnStartSequence - Fires when the user starts a move sequence of the triad.
OnMove - Fires when the triad moves as a result of a drag, reposition or if the user enters values for translation and rotation.
OnMoveTriadOnlyToggle - Fires when the Move Triad Only option is toggled.
OnStartMove - Fires when the triad begins to move as a result of a drag, reposition or if the user enters values for translation and rotation.
OnEndMove - Fires when the user ends an intermediate move of the triad.
OnTerminate - Fires when the triad is terminated.
OnSegmentSelectionChange - Fires every time a segment of a triad is selected.

Using the TriadEvents object

This example provides skeletal code demonstrating how to set up triad interaction events. It enables the triad and implements the triad move event. The code omits error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

Start a new VBA project and add a blank form (here named UserForm1). Add the following code to the form, starting with declarations of the interaction and triad event objects:

```
Private WithEvents oInteraction As InteractionEvents
Private WithEvents oTriadEvents As TriadEvents
```

Add a subroutine to be called when the form is initialized. Obtain the InteractionEvents object from the CommandManager object, and turn of selection and interaction. This means that the triad can still be manipulated by the user.

```
Sub UserForm_Initialize()
  Set oInteraction = ThisApplication.CommandManager.CreateInteractionEvents
  oInteraction.SelectionActive = False
  oInteraction.InteractionDisabled = True
```

Obtain the TriadEvents object from the InteractionEvents object, and then set its Repeat and Enabled properties to True. The Repeat property indicates that triad events should continue after a move sequence completes.

```
Set oTriadEvents = oInteraction.TriadEvents
  oTriadEvents.Repeat = True
  oTriadEvents.Enabled = True
```

Now start interaction events, and call DoEvents so the process can be terminated when required.

```
  oInteraction.Start

  While UserForm1.Enabled = True
    DoEvents
  Wend
End Sub
```

Add another subroutine, this time the OnMove event of the TriadEvents object. When the user moves the triad through the user interface, this code is called. This code prints a message, but other more significant actions can take place depending on context.

```
Private Sub oTriadEvents_OnMove(ByVal SelectedTriadSegment _
            As TriadSegmentEnum, _
            ByVal ShiftKeys As ShiftStateEnum, _
            ByVal CoordinateSystem As Matrix, _
            ByVal Context As NameValueMap, _
            HandlingCode As HandlingCodeEnum)

  Debug.Print "3D move / rotate tool has been moved."
End Sub
```

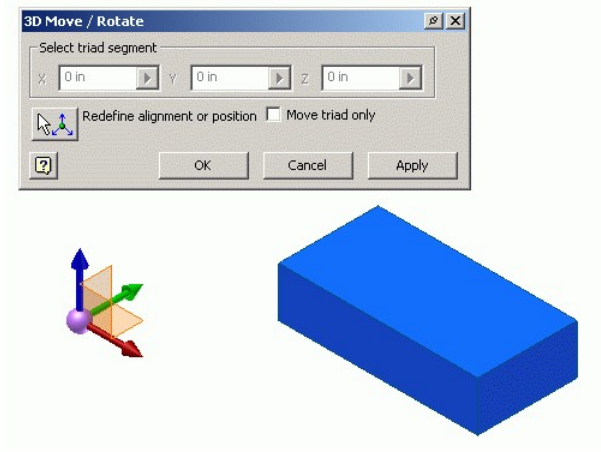
The last subroutine to add to the form cleans up once the form is closed, and it stops the InteractionEvents object.

```
Private Sub UserForm1_Terminate()
  oInteraction.Stop
  Set oTriadEvents = Nothing
  Set oInteraction = Nothing
End Sub
```

Finally, add the following code to a code module (not the form). To run the sample code, run this TriadEventsTest subroutine.

```
Public Sub TriadEventsTest()
    UserForm1.Show vbModeless
End Sub
```

When this sample is run in an open part document, the 3D Move/Rotate triad tool and dialog is displayed as follows:



When the triad tool is moved by the user, the OnMove event is called and the code prints a short message to the VBA debug screen (the 'immeadiate' window, if open).

Summary

The TriadEvents object and API provide a convenient means of obtaining location and transformation data in a fashion already familiar to the user. The triad can be activated and controlled through the API, and also provides a set of events to respond to changes to the triad made through the user interface.

Also consider

The InteractionEvents object provides selection, mouse and keyboard event callbacks, but also exposes the InteractionGraphics API. Like the ClientGraphics API, this API can be used to display transient geometry as visual cues. Used with TriadEvents, a lot of contextual information can be provided to the user in the form of visual cues and guidance.

Browser

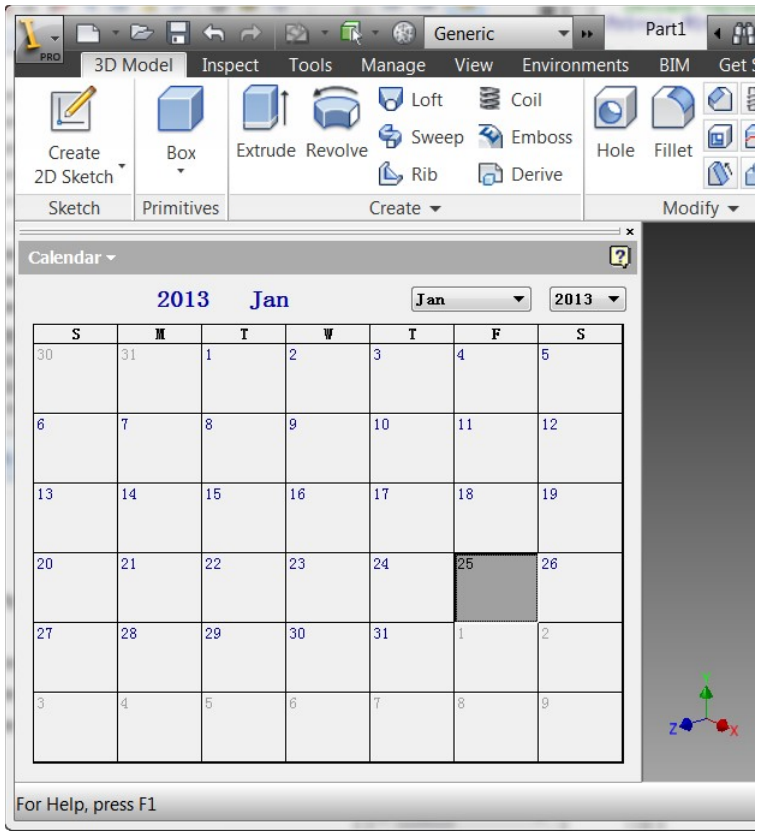
Browser Integration

The browser in Autodesk Inventor is a critical element of the user interface. Autodesk Inventor uses it to display a hierarchical representation of the contents of the current document. While this information is useful, there are cases when it is desirable to use this portion of the screen to display other types of information. For example, a finite element application might choose to show a representation of the constraints and loads currently defined on the model.

The Autodesk Inventor API allows you to define and interact with the contents of the browser window. The API supports the ability to create additional panes within the browser. You can also add a tree hierarchy of iconic nodes - see the browser nodes overview.

Only one pane is visible at a time, but the user can switch between panes and you can activate different panes using the API. To create new panes, Autodesk Inventor uses an approach that is both very simple and extremely powerful. The additional panes that you can add are just ActiveX control containers. From the API you create a new pane and specify which ActiveX control to display within the pane. What makes this powerful is the fact that an ActiveX control can contain any type of information and can even consist of other ActiveX controls. There are a huge number of existing ActiveX controls and it is relatively easy to construct new custom ActiveX controls.

Below is some sample code that illustrates the process of creating and interacting with a new browser pane. This sample creates a new pane and places Microsoft's ActiveX calendar control within it. (This control was chosen for the sample because it's relatively simple and should already be available on your system.) It responds to events from the control by displaying the current date as the user clicks on new days within the control. The picture below displays the result after the pane has been created.



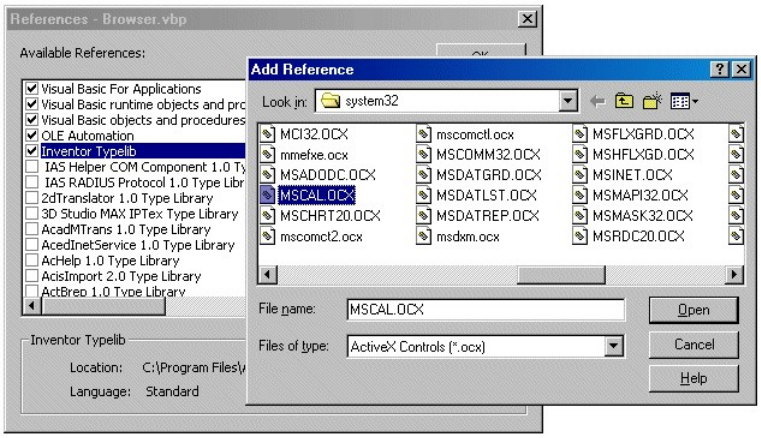
For this sample, all of the following code is contained within a form module. This is not intended to show the only approach to creating a browser pane. Often this would be done in a class module so you can have a single object that handles all of the browser pane information that is unique to each document. This sample was written this way to make it easier to focus on the browser pane portion of the code without the complication of other issues. To make the sample, create a new ActiveX EXE project and copy this code into the form module.

```
' Declare variable for the calendar control.
Private WithEvents oCal As MSACAL.Calendar

' Declare global variables for the Inventor document and Application.
Private WithEvents oDocEvents As DocumentEvents
Private oPane As BrowserPane
Private oApp As Inventor.Application

Private Sub Form_Load()
    ' Set a reference to a running instance of Inventor.
    ' This expects Inventor to be running.
    Set oApp = GetObject(, "Inventor.Application")

    ' Get the active document.  This assumes a document is open.
    Dim oDoc As Inventor.Document
    Set oDoc = oApp.ActiveDocument

    ' Connect to the documents events.  Used to
    ' listen for when the document is closed.
    Set oDocEvents = oDoc.DocumentEvents

    ' Create a new browser pane using the Microsoft calendar control.
    Set oPane = oDoc.BrowserPanes.Add("Calendar", "MSCAL.Calendar")

    ' Set a reference to the calendar control
    ' that was created on the pane.
    Set oCal = oPane.Control

    ' Set the calendar to today's date.
    oCal.Today

    ' Make the new pane the active pane.
    oPane.Activate
End Sub
```

The first line is declaring a variable to be the type of the control. This is declared using the WithEvents keyword, so you'll be able to capture events from the control. There's one thing to be aware of at this point: in order to declare a variable of this type you must reference the control into your project. You typically think of referencing controls using the Components command within Visual Basic. However, because of what appears to be a bug in Visual Basic, if you reference the control this way, you'll get a "Run-time error 13: Type mismatch" error on the line:

```
Set oCal = oPane.Control
```

To reference the control so you can use it in a browser pane you must use the References command. The control won't be displayed in the list so you need to click the Browse?button on the References dialog. The control used in this sample is found in the System32 directory. By default, .ocx files are not displayed in the file list, so you need to select "ActiveX Controls (*.ocx)" in the "Files of type" list. This control is "MSCAL.OCX."



Let's look at some of the more interesting portions of the code. The first few lines handle connecting to Autodesk Inventor, getting the active document, and connecting to the document's events. None of this is specific to creating a new browser pane. The next few lines are where it gets interesting. The BrowserPanes.Add method creates the new pane. It only has two arguments. The first is the name of the pane. This is the name that the user sees in the list when selecting panes and what is displayed at the top of the browser when that pane is active. You can see in the previous picture that "Calendar" is displayed as the browser's title. The second argument specifies which ActiveX control to place into the new pane. The control can be specified using either the ProgID, as is done in this sample, or the GUID of the control. If a GUID is specified it is as a string with the enclosing braces. For example, "{8E27C92B-1264-101C-8A2F-040224009C02}" would also work in this case.

Just creating the pane and adding the ActiveX control is usually not enough. Typically you'll need to interact with the control to populate data within it and keep the information up-to-date. The next line uses the Control property of the BrowserPane object to set a reference to the ActiveX control that was added to the pane. You now have complete access to the control through whatever methods, properties, and events it exposes. In the sample it calls the Today method of the control to set the calendar to today's date. Finally, it activates that pane to make it visible to the user.

The variable that is used to reference the control was declared using the WithEvents keyword to enable receiving events from the control. The code below is executed whenever the user clicks a date on the calendar. If the user selects January 1, 2000, it will cause the form to be unloaded. The code for the Form_Unload event is listed later.

```
' Event fired when the calendar is clicked
Private Sub oCal_Click()
    ' Display the new date in the Inventor status bar.
    oApp.StatusBarText = "Selected new date: " & _
                         Format(oCal.Value, "mmmm dd, yyyy")

    ' Check for January 1, 2000 and unload the form.
    If oCal.Value = #1/1/2000# Then
        Unload Me
    End If
End Sub
```

The code below is executed when the document is closed. It causes the form to be unloaded.

```
Private Sub oDocEvents_OnClose( _
                        ByVal BeforeOrAfter As Inventor.EventTimingEnum, _
                        ByVal Context As Inventor.NameValueMap, _
                        HandlingCode As Inventor.HandlingCodeEnum)
    If BeforeOrAfter = kBefore Then
        Unload Me
    End If
End Sub
```

The code below is executed when the form is unloaded. It releases all references and deletes the pane.

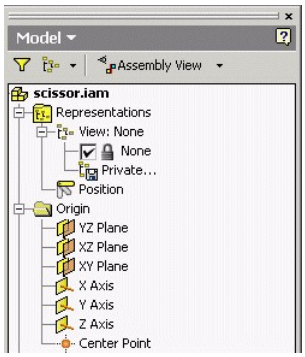
```
Private Sub Form_Unload(Cancel As Integer)
    Set oCal = Nothing
    Set oDocEvents = Nothing
    Set oApp = Nothing
    oPane.Delete
End Sub
```

The browser is a document-centric object. The contents of the browser are unique for each document. When adding panes, they must be added to each document you want the pane to be visible within. Each pane will have its own instance of the ActiveX control, so each one must be managed independently. The browser information is not persisted between sessions by Autodesk Inventor. If you need the data between sessions you'll need to persist the data yourself, create a new pane, and restore the data the next time the document is opened.

Browser Nodes

Introduction to Browser Nodes

A significant element of the Autodesk Inventor user interface is the browser. In a part file, this displays the names of features, sketches, work features and so on. In an assembly file, it also shows part or subassembly occurrences and assembly features. It provides a convenient and intuitive means of traversing and selecting objects and features in the model hierarchy. Typically, the status of each object or feature is indicated by a graphical icon at the appropriate tree node. The following figure represents a browser tree view of a model assembly.

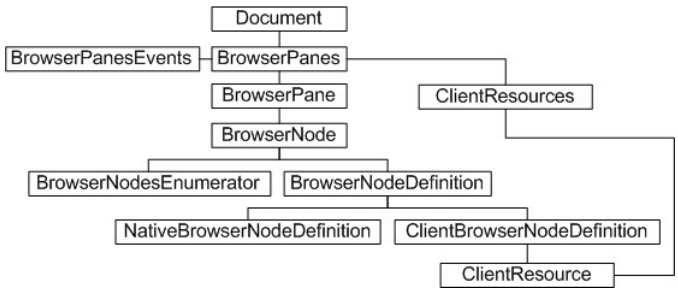


The purpose of browser node customization

The Autodesk Inventor API allows customization of the browser pane. This can mean adding a new pane, perhaps containing an ActiveX control. This topic is covered in a separate overview - see the Browser pane overview.

The API also enables iteration of existing browser tree nodes and the addition of browser panes containing custom tree structures representing your own data, complete with your own graphical icons. For example, you can represent a file system, your organization's management hierarchy, or some discipline-specific data such as HVAC throughput.

Browser API Object Model diagram



Iterating browser nodes through the API

The following code iterates through the browser model tree of an open part or assembly document, printing the node labels to the VBA debug screen (the "Immediate" window in VBA, if enabled). This code omits error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

This code defines two routines. The first one obtains the top-level objects and then the second routine is called recursively (it repeatedly calls itself) to iterate through the entire browser tree structure. Add this code to an *ApplicationProject* code module in VBA.

First, obtain the *Document* object.

```
Private Sub QueryModelTree()
    Dim Doc As Document

    If (ThisApplication.Documents.Count <> 0) Then
        Set Doc = ThisApplication.ActiveDocument
    Else
        MsgBox "There are no open documents!"
        Exit Sub
    End If
```

Now obtain the top node for the browser pane of the model tab.

```
    Dim oTopNode As BrowserNode
    Set oTopNode = Doc.BrowserPanes("Model").TopNode
```

From the top node, call the routine named "recurse", which prints the node definition label and moves on to the next node, if any.

```
    Call recurse(oTopNode)
End Sub
```

The second routine calls itself for each node in the collection of browser nodes, printing the node definition label of each node.

```
Sub recurse(node As BrowserNode)
    If (node.Visible = True) Then
        Debug.Print node.BrowserNodeDefinition.Label
        Dim bn As BrowserNode
        For Each bn In node.BrowserNodes
            Call recurse(bn)
        Next
    End If
End Sub
```

For the assembly tree denoted by the first figure, this results in a report similar to the following example.

```
scissor.iam
Representations
View: None
None
Private...
Position
Origin
YZ Plane
XZ Plane
XY Plane
X Axis
Y Axis
Z Axis
Center Point?
```

and so on.

Creating a custom pane and adding browser nodes

The preceding code simply iterated through an existing tree node hierarchy.

The following code adds a new pane tab and then adds several levels of tree nodes. This results in a new tab on the browser pane in the user interface, in addition to the standard model, library and representations tabs. Add this code to the *ThisDocument* module in VBA.

First, obtain the *BrowserPanes* collection for the current document. As noted in the object model diagram, new panes are added to this collection, and also this object provides access to the client node resources through which new icons may be added.

```
Sub AddNodes()

Dim oPanes As BrowserPanes
Set oPanes = ThisDocument.BrowserPanes

Dim oRscs As ClientNodeResources
Set oRscs = oPanes.ClientNodeResources
```

Create a standard Microsoft Windows *IPictureDisp* referencing an icon (.bmp) bitmap file. Change the file referenced here as appropriate - here the code references *test.bmp*. This is the icon that will be displayed at this node. Add the *IPictureDisp* to the client node resource. (If an invalid icon is supplied, the icon will default to the standard "Notepad" icon.)

```
Dim oIcon As IPictureDisp
Set oIcon = LoadPicture("c:\temp\test.bmp")

Dim oRsc As ClientNodeResource
Set oRsc = oRscs.Add("Test", 1, oIcon)
```

Before adding a new pane tab to the panes collection, define the top node the pane will contain. Pass this node when creating the new pane "My Pane" through the *AddTreeBrowserPane* method.

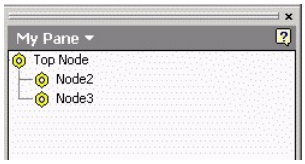
```
Dim oDef As BrowserNodeDefinition
Set oDef = oPanes.CreateBrowserNodeDefinition("Top Node", 3, oRsc)

Dim oPane As BrowserPane
Set oPane = oPanes.AddTreeBrowserPane("My Pane", "MyGUID", oDef)
```

Add two child nodes to the tree, labeled Node 2 and Node 3.

```
Dim oDef1 As BrowserNodeDefinition
Dim oNode1 As BrowserNode

Set oDef1 = oPanes.CreateBrowserNodeDefinition("Node2", 5, oRsc)
Set oNode1 = oPane.TopNode.AddChild(oDef1)

Dim oDef2 As BrowserNodeDefinition
Dim oNode2 As BrowserNode

Set oDef2 = oPanes.CreateBrowserNodeDefinition("Node3", 6, oRsc)
```

```
Set oNode1 = oPane.TopNode.AddChild(oDef2)

End Sub
```

The preceding routine results in a new pane and tree structure as follows (substituting your icon).



Browser nodes can be reordered. The *BrowserPane.Reorder* method allows one or more nodes to be repositioned within the tree hierarchy. (Note: this is not possible with the assembly pane.)

If a custom pane has browser nodes that are conceptually connected to objects (sketches, faces, constraints, and so on) in the model, this relationship can be created and maintained through the use of reference keys. The *ClientBrowserNodeDefinition* object supports the *GetReferenceKey* method.

By default, the browser pane is saved with the document. This means it is the responsibility of the application to either remove its browser UI, or reconnect and synchronize with the browser when the document is next present.

Events

The *BrowserPanesEvents* property of the *BrowserPanes* collection object returns the *BrowserPanesEvents* object, supporting browser node event notification. This allows applications to emulate Autodesk Inventor's behavior by reacting to user manipulation of the tree nodes. Events supported include node label editing, node activation, and object highlighting. These events are in addition to those supported by the *BrowserPane* object.

Summary

The browser customization API can be divided into two areas: adding an ActiveX control to a browser pane, or adding a hierarchical tree node structure to a browser pane. The latter allows an application to represent its data or constructs in a graphical manner already familiar to the user, complete with custom icons and labels. Events allow the application to be notified when the user selects a tree node.

Also consider

For complex user interactions, standard or custom ActiveX controls can be added to browser panes. The browser panes collection object acts as an ActiveX container.

Environments

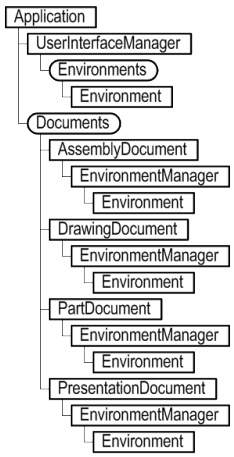
Introduction to Environments

To the Autodesk Inventor user, an environment, such as the assembly environment or the sketch environment, simply means that tools are made available to accomplish the task at hand. Menus and panel bars appropriate to the workflow are presented to the user. For example, the sketch environment provides commands to create sketch lines, circles and so on, but not commands for placing or copying components which are actions specific to the assembly environment.

The purpose of Environments

If all Autodesk Inventor commands were present and available on the menus and panel bar all the time, the user interface would be very cluttered and confusing. Environments provide a means to present a focused subset of commands. Taking this a step further, a client application may define custom environments in addition to Autodesk Inventor's set of built-in environments. The application may define the environment as transient, or it may be persistent across sessions. Additionally, a given environment may have a number of edit-targets, any of which may have focus. For example, an application may define an FEA environment. The results of FEA computation models may result in a number of different solution results, each represented by a node on the browser.

Environments Object Model Diagram

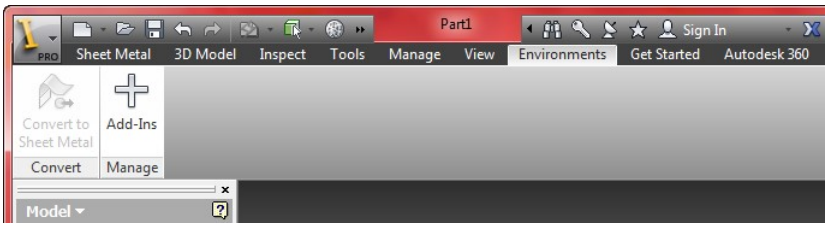


Working with Environments through the API

Creating an environment

Every document has a built-in base environment. A client application may create a new custom environment by adding to the Environments collection, and will typically copy some components used in another environment, such as command bars, to the new one. Command bars can inherit their properties from the parent environment if the InheritAll property of the CommandBarList is set to true.

The samples that follow show how to add new environments to Autodesk Inventor. The figure below shows the initial view of the Applications menu and the panel bar, with no additional environments or edit targets added. The example code will add three new environments. Lastly, it will create an edit target within one of these environments.



The first step is to get the Environments collection from the UserInterfaceManager. From this, obtain a built-in environment to use as the basis for the new environment. This sample and the following code omit error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

```
Sub AddParallelEnv1()
    Dim oEnvs As Environments
    Set oEnvs = ThisApplication.UserInterfaceManager.Environments

    Dim oEnv As Environment
    Set oEnv = oEnvs("MBxSheetMetalEnvironment")
```

Next, add a new environment to the environments collection. Here, we use a display name of "FEA" - this is the name that will be displayed in the Applications menu. The string "FEA Env's internal name" is this the internal name for the new environment - its equivalent to the internal names of built-in environments, such as "DLxDrawingEnvironment". (A ClientId is not supplied so this environment is not persisted.)

```
    Dim oNewEnv As Environment
    Set oNewEnv = Nothing
    Set oNewEnv = oEnvs.Add("FEA", "FEA Env's internal name")
```

This example will use the same default command bar as the source environment, and will populate the new environment's panel bar with some command bars. It will also copy the command bars from the context menu.

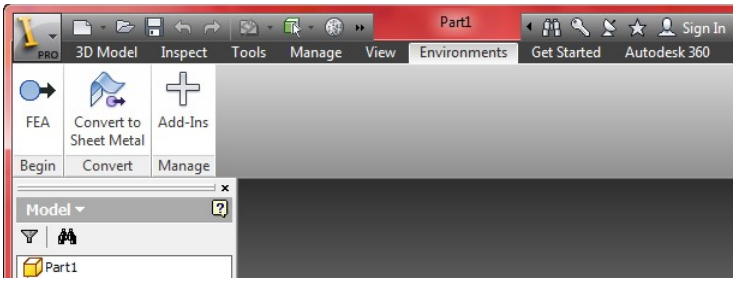
```
    oNewEnv.DefaultRibbonTab = oEnv.DefaultRibbonTab
    Dim sTabs(0 To 1) As String
    sTabs(0) = "id_TabSheetMetal"
    sTabs(1) = "id_TabFlatPattern"

    oNewEnv.AdditionalVisibleRibbonTabs = sTabs
    oNewEnv.InheritAllRibbonTabs = True
```

In this case the new environment will inherit all the command bars from the parent environment.

```
    oNewEnv.PanelBar.CommandBarList.InheritAll = True
    oNewEnv.ContextMenuList.InheritAll = True
```

All that remains is to add the new environment to the list of parallel environments (thus adding the environment to the list of available environments, as indicated by the Applications menu shown in the figure below.

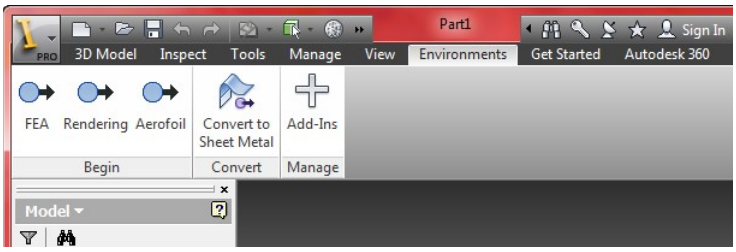
```
    Dim oParEnvs As EnvironmentList
    Set oParEnvs = ThisApplication.UserInterfaceManager.ParallelEnvironments
    Call oParEnvs.Add(oNewEnv)
End Sub
```



Next, add another two environments named "Rendering" and "Aerofoil", based on the assembly and weldment environments respectively. This code is much the same as that in the previous section.

```
Sub AddParallelEnv2()
    Dim oEnvs As Environments
    Set oEnvs = ThisApplication.UserInterfaceManager.Environments

    Dim oEnv As Environment
    Set oEnv = oEnvs("AMxAssemblyEnvironment")

    Dim oNewEnv As Environment
    Set oNewEnv = Nothing

    Set oNewEnv = oEnvs.Add("Rendering", "Rendering Env's internal name")

    Dim sTabs(0 To 1) As String
    sTabs(0) = "id_TabAssemble"
    sTabs(1) = "id_TabDesign"

    oNewEnv.DefaultRibbonTab = oEnv.DefaultRibbonTab
    oNewEnv.InheritAllRibbonTabs = False

    Dim oParEnvs As EnvironmentList
    Set oParEnvs = ThisApplication.UserInterfaceManager.ParallelEnvironments

    Call oParEnvs.Add(oNewEnv)
End Sub


Sub AddParallelEnv3()
    Dim oEnvs As Environments
    Set oEnvs = ThisApplication.UserInterfaceManager.Environments

    Dim oEnv As Environment
    Set oEnv = oEnvs("AMxWeldmentEnvironment")

    Dim oNewEnv As Environment
    Set oNewEnv = oEnvs.Add("Aerofoil", "Aerofoil Env's internal name")

    oNewEnv.DefaultRibbonTab = oEnv.DefaultRibbonTab
    oNewEnv.InheritAllRibbonTabs = True

    Dim oParEnvs As EnvironmentList
    Set oParEnvs = ThisApplication.UserInterfaceManager.ParallelEnvironments

    Call oParEnvs.Add(oNewEnv)
End Sub
```

The preceding code will add the new FEA, Rendering and Aerofoil environments to Autodesk Inventor's Applications menu, as shown below. Each environment has its own set of command bars, but so far none has a specific edit target defined.

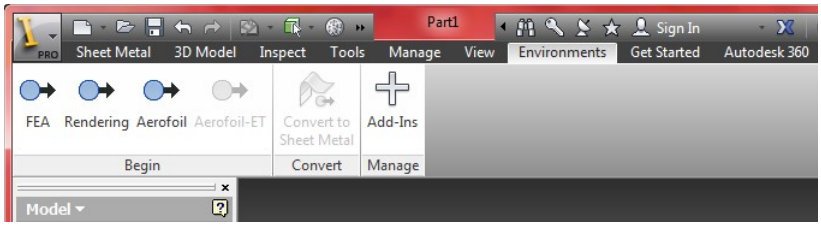


Now to add an edit target to one of these new environments. This indicates a specific editing requirement. An example in Autodesk Inventor is the sketch editing facility within the Part environment. The code is much the same as previously except now the SetCurrentEnvironment method of the EnvironmentManager object is used - this sets the environment for this document but it is not persisted.

```
Sub ActivateET()
    Dim oEnvs As Environments
    Set oEnvs = ThisApplication.UserInterfaceManager.Environments

    Dim oEnv As Environment
    Set oEnv = oEnvs("MBxSheetMetalEnvironment")

    Dim oPartDoc As PartDocument
    Set oPartDoc = ThisApplication.ActiveDocument

    Dim oNewEnv As Environment
```

```
    Set oNewEnv = oEnvs.Add("Aerofoil-ET", "ET Env's internal name")

    oNewEnv.DefaultRibbonTab = oEnv.DefaultRibbonTab
    oNewEnv.InheritAllRibbonTabs = True

    Call oPartDoc.EnvironmentManager.SetCurrentEnvironment(oNewEnv, "ET's Cookie")
End Sub
```

The result of this code is indicated in the following figure. Note that the Applications menu now has the new Aerofoil-ET edit target checked instead of the Part edit target. Note too the change to the panel bar.



Although the previous edit target is not persisted in the sense that it will not be active when the document is next loaded, you can return to this edit target by calling the SetCurrentEnvironment method again.

```
Sub ActivateETAgain()
    Dim oEnvs As Environments
    Set oEnvs = ThisApplication.UserInterfaceManager.Environments

    Dim oPartDoc As PartDocument
    Set oPartDoc = ThisApplication.ActiveDocument

    Dim oNewEnv As Environment
    Set oNewEnv = Nothing

    Set oNewEnv = oEnvs("ET Env's internal name")

    Call oPartDoc.EnvironmentManager.SetCurrentEnvironment(oNewEnv, "ET's Cookie")
End Sub
```

However, an edit target can be made persistent by setting the OverrideEnvironment property of the EnvironmentManager object. Thus, when a document is loaded, the specified edit target will be present.

```
Sub TestOverrideEnv()
    Dim oEnvs As Environments
    Set oEnvs = ThisApplication.UserInterfaceManager.Environments

    Dim oEnv As Environment
    Set oEnv = oEnvs("MBxSheetMetalEnvironment")

    Dim oPartDoc As PartDocument
    Set oPartDoc = ThisApplication.ActiveDocument

    Dim oNewEnv As Environment
    Set oNewEnv = oEnvs.Add("OverrideTmp", "OverrideTmp Env's internal name")

    oNewEnv.DefaultRibbonTab = oEnv.DefaultRibbonTab
    oNewEnv.InheritAllRibbonTabs = True

    Dim oEnvMgr As EnvironmentManager
    Set oEnvMgr = oPartDoc.EnvironmentManager

    oEnvMgr.OverrideEnvironment = oNewEnv
End Sub
```

Summary

The combination of custom environments, and the ability to define transient or overridden edit targets within those environments, allows client applications to define a user interface specific to their purpose, with full control over what is presented to the user. They can behave in the same manner as built-in Autodesk Inventor environments, with environments and edit targets pushed onto the environment stack, and with Autodesk Inventor's Return button traversing back through the stack as expected.

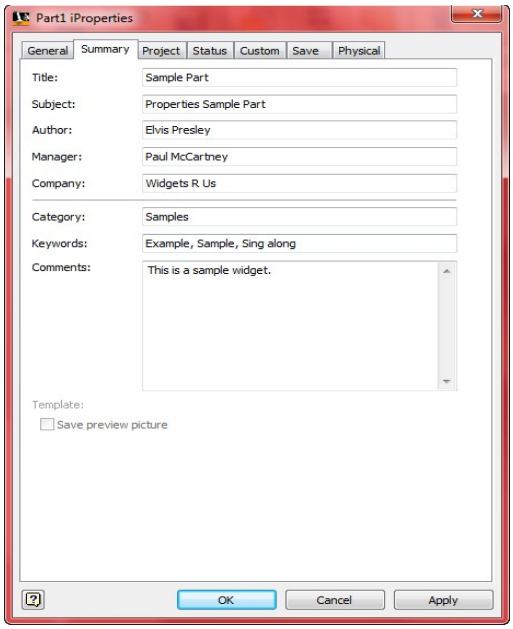
Also consider

For additional control over what buttons are enabled in specific scenarios, use the DisabledCommandList object. This is available both at the document and environment level, and is complementary to the Enabled property of the ControlDefinition object. Commands placed in the DisabledCommandList object at the environment level will be grayed-out within that environment.

Document Properties

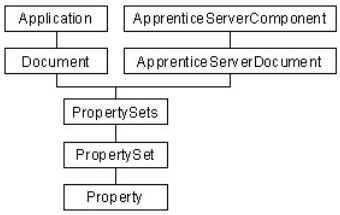
Introduction

Autodesk Inventor has a set of properties associated with every document. The properties for a file can be viewed and set from inside Autodesk Inventor using the "File | Properties" command. The dialog displayed for this command is shown below. You can also access the properties using Windows Explorer by right-clicking on the file and selecting the "Properties" option. Autodesk Inventor's Design Assistant also provides access to document properties. You also have full access to the property information using the API.



Accessing Properties Using the API

Using the API, you have complete read and write access to the document properties using both Autodesk Inventor and Apprentice. In fact more functionality is exposed through the API than through the various property-related user interfaces. The diagram below shows the object hierarchy for properties.



From the hierarchy diagram you can see that properties are accessible from Autodesk Inventor and Apprentice. You can also see the structure in which the properties are stored. The various properties are grouped within "property sets." From the Document object (or the ApprenticeServerDocument object when using Apprentice) you can obtain the PropertySets collection object. The PropertySets collection object provides access to all of the property sets in a document. The PropertySet object represents each property set and each property set provides access to the properties it owns.

The sample below illustrates obtaining the PropertySets collection object using Apprentice.

```
' Declare the Apprentice object
Dim oApprentice As New ApprenticeServerComponent

' Open a document using Apprentice
Dim oApprenticeDoc As ApprenticeServerDocument
Set oApprenticeDoc = oApprentice.Open("C:\Test\part.ipt")

' Obtain the PropertySets collection
Dim oPropsets As PropertySets
Set oPropsets = oApprenticeDoc.PropertySets
```

The sample below illustrates obtaining the PropertySets collection object using Autodesk Inventor.

```
' Declare the Application object
Dim oApplication As Inventor.Application

' Obtain the Inventor Application object.
' This assumes Inventor is already running.
Set oApplication = GetObject(, "Inventor.Application")

' Set a reference to the active document.
' This assumes a document is open.
Dim oDoc As Document
Set oDoc = oApplication.ActiveDocument
```

```
' Obtain the PropertySets collection object
Dim oPropsets As PropertySets
Set oPropsets = oDoc.PropertySets
```

The PropertySets collection object supports methods to iterate through and access all the PropertySets in the collection. Each property set has two identifiers: the display name and the internal name. The display name is a string that helps to identify the type of information stored by the properties within the property set. Usually the display names are unique but this is not guaranteed. The internal name is another identifier of a property set and is guaranteed to be unique. The internal name is actually a GUID converted to a string. The following code can be added to either of the samples above to show the display name and internal name of every property set in a document.

```
' Iterate through all the PropertySets one by one using for loop
Dim oPropSet As PropertySet
For Each oPropSet in oPropsets
    ' Obtain the DisplayName of the PropertySet
    Debug.Print "Display name: " & oPropSet.DisplayName

    ' Obtain the InternalName of the PropertySet
    Debug.Print "Internal name: " & oPropSet.InternalName

    ' Write a blank line to separate each pair.
    Debug.Print
Next
```

When a new Autodesk Inventor document is created, Inventor adds a standard set of property sets. The following are the display names, internal names, and brief description of all of the pre-defined property sets. Prior to Autodesk Inventor 6 some of the Autodesk Inventor properties used standard property sets as defined by Microsoft. Starting with Autodesk Inventor 6 all property sets used are now exclusive to Autodesk Inventor. This change was made to solve problems that occurred when moving files from one language to another. The current set of property sets and their internal names are shown below.

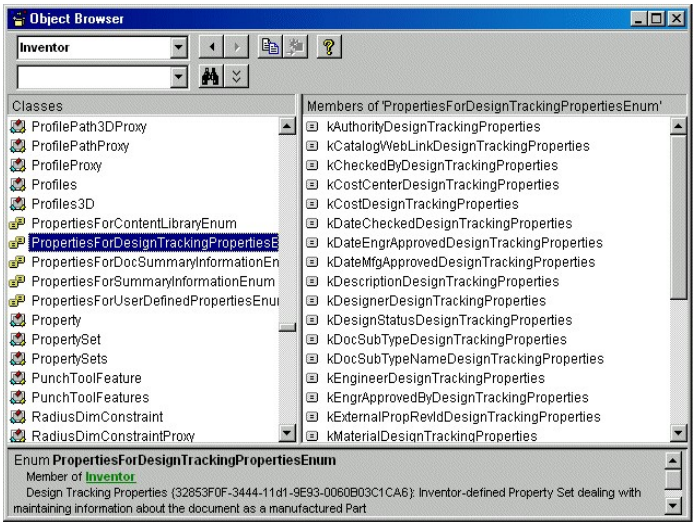
Summary Information, {F29F85E0-4FF9-1068-AB91-08002B27B3D9}
Document Summary Information, {D5CDD502-2E9C-101B-9397-08002B2CF9AE}
Design Tracking Properties, {32853F0F-3444-11D1-9E93-0060B03C1CA6}
User Defined Properties, {D5CDD505-2E9C-101B-9397-08002B2CF9AE}

The properties sample delivered in the SDK demonstrates the reading and writing the properties and also serves as a tool to view the properties of a document. The sample is available in the "SDK\Samples\VB.NET\Standalone Applications\ApprenticeServer\Properties" directory.

If you want to access a particular property set and know either its display name or internal name you can access it directly using the Item property of the PropertySets object. The alternative is to iterate through the collection looking for a particular property set. You can use the display name or internal name but because the display name is not guaranteed to be unique the internal name will be more reliable. Also if your application will be used in other countries, you should use the internal name because it will always remain the same and will not be affected by localization changes made. The sample below illustrates obtaining a specific property set using the display name.

```
' Get a reference to the "Design Tracking Properties" PropertySet
Dim oPropSet As PropertySet
Set oPropSet = oPropsets.Item("Design Tracking Properties")
```

Using the internal name to obtain a specific property set is done in the same way, but first you need to know what the internal name is. The internal names for the Autodesk Inventor-created property sets are listed above. You can also find the internal names using the Object Browser. If you select the enumerator for the property set you're interested in, the internal name is displayed as part of the description at the bottom of the browser as shown below.



The sample below shows how to access a property set using the internal name.

```
' Get a reference to the "Design Tracking Properties" PropertySet
Dim oPropSet As PropertySet
Set oPropSet = oPropsets.Item("{32853F0F-3444-11d1-9E93-0060B03C1CA6}")
```

Once you have obtained a property set you have access to the properties it contains. A property is basically just a name and a value. Each property is identified by a Property ID (a Long). The ID value is guaranteed to be unique among all properties within a specific property set. The value of a property is a Variant so it can contain almost any type of value: Integer, Double, String, Dates, and arrays of these types. The thumbnail property also returns an IPictureDisp object that can be used by standard picture controls.

Just as the PropertySets object is a collection of different PropertySet objects, the PropertySet object in turn is a collection of different Property objects and similar to the PropertySets collection object, the PropertySet object supports methods to iterate through and access all the properties in a particular set. It also provides direct access to a particular property if its name or property ID is known. The following code segment shows how to access all the properties in a PropertySet.

```
Dim oPropSet As PropertySet
For Each oPropSet In oPropSets
    ' Iterate through all the Properties in the current set.
    Dim oProp As Property
    For Each oProp in oPropset
        ' Obtain the Name of the Property
        Dim Name As String
        Name = oProp.Name

        ' Obtain the Value of the Property
        Dim Value As Variant
        Value = oProp.Value

        ' Obtain the PropertyId of the Property
        Dim PropertyId As Long
        PropertyId = oProp.PropId
    Next
Next
```

Properties can also be accessed directly if their names (called property IDs) are known. All of the Autodesk Inventor-native and Microsoft-defined properties have their property IDs specified in the corresponding enum. As shown in the previous diagram of the object browser, each property has a corresponding enum value. This is its PropertyID. The sample below directly accesses a specific property using the internal name of the property set and the unique ID of the property.

```
' Access the value of the "Cost" Property of
' "Design Tracking Properties" PropertySet
Dim Value As Variant
Value = oPropsets.Item("{32853F0F-3444-11d1-9E93-0060B03C1CA6}").ItemByPropId(kCostDesignTrackingProperties).Value
```

Creating, Changing and Deleting Properties

Creating new PropertySets and Properties

Autodesk Inventor allows users to create their own properties, which are added to the "Custom Properties" collection. These properties are added to the "User Defined Properties" PropertySet. The API also supports the creation and edit of custom properties. In addition it also supports the creation of totally new PropertySets and Properties.

The steps involved in creating a new collection of properties are:

1. Add a new PropertySet object to the PropertySets collection. The PropertySets collection object supports the Add method, which allows for the addition of a new PropertySet to the collection.
2. Add new properties to the newly created PropertySet object. The PropertySet object supports the Add method, which allows for addition of new properties to the PropertySet.

```
'declare new PropertySet object to be added
Dim oNewPropertySet As PropertySet

On Error Resume Next
' Try to obtain the PropertySet to see if it already exists
Set oNewPropertySet = oPropsets.Item("New PropertySet")

' If PropertySet does not exist then add the new PropertySet
If Err Then
        ' Add the new PropertySet
        Set oNewPropertySet = oPropsets.Add("New PropertySet")

        ' Adding the new Property to the new PropertySet
        Call oNewPropertySet.Add("A Value", "New Property", 2)
End If
```

Display names and internal names for property sets must be unique with respect to all other property sets in the document. In the sample above, no internal name is specified, which causes Autodesk Inventor to create one. When creating properties, the name, display name, and ID must be unique with respect to other properties in that property set. The ID must be greater than 1, (the ID of 1 is reserved by Microsoft) and less than 255.

The above sample demonstrates how to create a new property set and add a property to that set. Autodesk Inventor permits adding properties to new property sets and the "Custom" property sets. Adding properties to the standard property sets is blocked.

Changing the Names and Values of Properties

The Autodesk Inventor API allows you to change both the names and values of properties that have been created by third parties, but only the value of pre-defined properties can be changed. For example, the PropertySet object's DisplayName property can be changed for user-created PropertySet objects but not for pre-defined PropertySets. The InternalName property cannot be reset for any of the PropertySet objects. For the Property object, the Name and Value can be changed for properties created by third parties but not for pre-defined properties. You can tell if a property is read-only or not by referring to the helpstring associated with the corresponding enum for the property. The PropID is read-only and cannot be changed for any of the properties.

The name of a property can be changed by using the Name property of the Property object. The value of a property can be changed using the Value property of the Property object. However, for pre-defined properties you need to be careful to use the correct type when setting the value. For example, let's consider changing the value of the "Date Checked" property (of the "Design Tracking Properties" PropertySet). This needs to be set using a Variant of Date type, otherwise Autodesk Inventor functions that read this value may not work correctly. The sample below illustrates setting this property to a specific date.

```
' Get a reference to the "Date Checked" property.
Dim oProp As Property
Set oProp = oPropSets.Item("{32853F0F-3444-11d1-9E93-0060B03C1CA6}").ItemByPropId(kDateCheckedDesignTrackingProperties)

' Define a Date variable with the desired date.
Dim NewDate As Date
NewDate = "Jan 1, 2000"

' Assign the date to the property.
oProp.Value = NewDate
```

The following code could cause problems in Autodesk Inventor because a String rather than a Date type of variable is assigned to the property.

```
' Get a reference to the "Date Checked" property.
Dim oProp As Property
Set oProp = oPropSets.Item("{32853F0F-3444-11d1-9E93-0060B03C1CA6}").ItemByPropId(kDateCheckedDesignTrackingProperties)

' Assign the date to the property.
oProp.Value = "Jan 1, 2000"
```

A Date property has the check box that can be checked/unchecked in UI, when set a Date property with a valid Date variable(should be later than Jan 1, 1601) then the check box is checked, when set a Date property using a Date variable with value of Jan 1, 1601 will clear the checked status and set the Property.Expression to empty string.

Deleting Properties Created by Third Parties

The PropertySet and Property objects both support Delete methods, which can be used to delete PropertySets and Properties respectively. Deleting PropertySets and Properties is not supported for pre-defined property sets and properties. Calling the Delete method for pre-defined properties will result in an error.

The following example shows how to delete all the properties in the "New PropertySet" that we created earlier by first deleting all the properties in the collection and then deleting the PropertySet itself. (It's not required to delete the contents of the PropertySet before deleting the PropertySet.)

```
' Declare the PropertySet object
Dim oPropSet As PropertySet

' Try to obtain the PropertySet object by using its name
Set oPropSet = oPropsets.Item("New PropertySet")

' Iterate through all the Properties and delete them
Dim oProp As Property
For Each oProp In oPropSet
    ' Delete the Property object
    oProp.Delete
Next

' Delete the PropertySet
oPropSet.Delete
```

Saving Changes to Properties

When working with Inventor, any changes made to properties are saved when the document is saved by the user. When using Apprentice you can save any property changes by saving the entire document or you can use the FlushToFile method of the PropertySets object. Using the FlushtoFile is much more efficient than saving the entire document. FlushToFile is not supported by Autodesk Inventor.

Attributes

Introduction to Attributes and AttributeSets.

The Autodesk Inventor API provides the means to append data, in the form of named data or a name-value pair, to objects in Autodesk Inventor. Attributes are how the programmer stores nongraphical data on Autodesk Inventor objects, and for this data to be maintained by Autodesk Inventor. Attribute data is also accessible using Apprentice Server. Attribute data is persistent by default. However, if the Transient property is set to true, the data is to be maintained during the current Autodesk Inventor session only.

You can append the following types of data: integer, double, string, byte array, or any combination of these. Applications differentiate their attributes from those of other applications through the use of AttributeSets. An AttributeSet can contain any number of Attributes, and an object can contain any number of AttributeSets, dependant only on system resources. Most persistent objects (SketchLine, Constraint, Parameter, and so on) in Autodesk Inventor support Attributes. Brep objects support attributes in the current session, with attributes maintained across parametric changes and model re-computations.

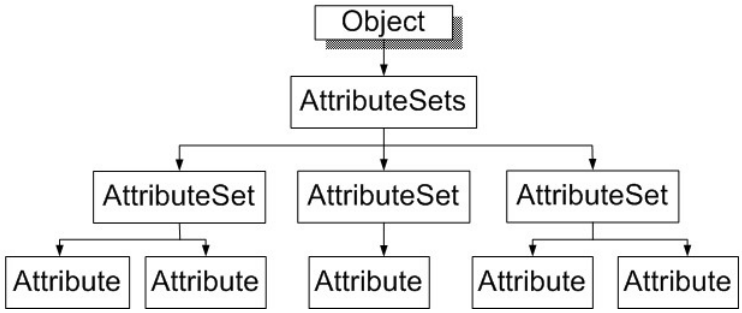
There is no direct User Interface equivalent to Attributes. Similar API functionality exists in AutoCAD through Extended Entity Data, otherwise known as XED or xdata.

The purpose of Attributes

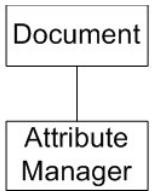
Attributes are commonly used to attach application-specific data to objects within Autodesk Inventor. A typical example is in referencing records in an external database, such as Microsoft Access or Oracle. An attribute appended to a part occurrence may contain a key value. An application can use this value to identify a unique database record (perhaps using ODBC or OLEDB). Thus Autodesk Inventor maintains an attribute value that is a persistent link to further data in an external database. This database can be used for Bill of Materials reports, data updates, and so on.

It is useful to be able to locate Autodesk Inventor objects containing attributes of a certain value. Use the AttributeManager object to accomplished this. The following object model diagrams show that this object is separate from the Attribute and AttributeSets objects. The AttributeManager object is obtained from the document object (DrawingDocument, PartDocument, AssemblyDocument, ApprenticeServerDocument and so on.)

AttributeSet and Attribute Object Model Diagram



AttributeManager Object Model Diagram



Accessing Attributes through the API

To take full advantage of attributes, it is necessary to add, modify and erase Attributes and AttributeSets programmatically. The AttributeSets object also supports the DataIO object, allowing attribute data to be written out in XML form.

Note: Since Autodesk Inventor 9, Attributes and AttributeSets also support the Transient property. If this property is set to true, or if the AttributeSet is created using the AddTransient method of AttributeSets, the attribute data is not persisted beyond the current session. This is useful if you want to do some short-term data manipulation but do not wish to 'dirty' the document. For example, when working with Brep objects at the session level.

Creating an AttributesSet

The first step in appending data to an Autodesk Inventor object is to create and add a new AttributeSet object to the AttributeSets collection. This is obtained through the AttributeSets property of the object to which the attributes are to be attached. Initially, this AttributeSet can be empty of Attributes. So, assuming an assembly document contains a single part occurrence oMyPartOccurrence:

```
Dim oAttribSets As AttributeSets
Set oAttribSets = oMyPartOccurrence.AttributeSets

Dim oAttribSet As AttributeSet

On Error Resume Next
Set oAttribSet = oAttribSets.Add("MyAttribSet")
If Err Then
  Set oAttribSet = oAttribSets.Item("MyAttribSet")
End If
```

The preceding sample code obtains the AttributeSets collection object from the part occurrence, and then adds a new AttributeSet object named "MyAttribSet" to the collection, if one doesn't already exist.

Creating Attributes and adding or modifying values

Next, declare and add an Attribute to the AttributeSet object using the add method of the AttributeSet object.

```
Dim oAttrib As Inventor.Attribute
If oAttribSet.Count = 0 Then
  Set oAttrib = oAttribSet.Add("MyAttrib", kStringType, "Some Text")
End If
```

This code adds a single attribute of type String, containing the text "Some Text", after first checking that the AttributeSet doesn't already contain attributes. An alternative is to check if existing attributes have the same name, and if not, add the new attribute anyway since there is no naming conflict. To modify the value of an existing attribute, update its Value property.

Retrieving attributes values

When obtaining the value of an existing attribute, you usually know its name. The Item method of both AttributeSets allow the use of a name string to identify the required object. Alternatively, the Item method accepts a numeric index value, allowing iteration of all available AttributeSets. In this example, we use the previously named attributes.

```
Dim oAttribSets As AttributeSets
Set oAttribSets = oMyPart.AttributeSets

Dim oAttrib As Inventor.Attribute
Set oAttrib = oAttribSets.Item("MyAttribSet").Item("MyAttrib")
Debug.Print oAttrib.Value
```

For clarity and brevity, the previous example doesn't perform any error checking. You could use the NameIsUsed property (see the next section) to check whether the attribute exists. Given the existence of the attribute 'MyAttrib' this code prints out its value, "Some Text".

Deleting Attributes

Deleting attributes is straightforward, except that it is always wise to check for the existence of attributes contained in an AttributeSet before deleting that AttributeSet. If the AttributeSet contains attributes, delete those attributes before deleting the AttributeSet. It is the programmer's responsibility to ensure it's safe to delete attributes and AttributeSets. There is no Recycle Bin for attributes, although you can undo the last operation. If orphaned attributes exist due to the owning object being deleted or inaccessible, such attributes should be purged using the PurgeAttributeSets method of the AttributeManager.

The following code uses the delete method of the Attribute object, followed by the delete method of the parent AttributeSet object.

```
Dim oAttribSets As AttributeSets
Set oAttribSets = oMyPart.AttributeSets

If oAttribSets.NameIsUsed("MyAttribSet") Then
  Set oAttrib = oAttribSets.Item("MyAttribSet").Item("MyAttrib")
  oAttrib.Delete
  oAttribSets.Item("MyAttribSet").Delete
End If
```

Again, the previous code uses minimal error checking, though it does demonstrate the use of the NameIsUsed property to verify the existence of the AttributeSet.

Searching for Attributes using the AttributeManager object

An assembly may contain hundreds or thousands of objects, be they occurrences, dimensions, constraints, or any other persistent object. If all these objects contain attributes as described in the preceding sections, it would be a very slow and tedious process to search each object for a given AttributeSet or Attribute. Brep objects are another example, with a potentially large number of objects to iteration through. Thankfully, Autodesk Inventor provides an API to help - the AttributeManager object. You can obtain this object from the Document object, and from derived documents such as AssemblyDocument. The AttributeManager object provides a number of methods. For example, to find if an AttributeSet, Attribute or Attribute value exists, or to return objects based on specific attribute values.

Note: A helpful feature of the AttributeManager object is that wildcard characters are legal when used to specify AttributeSet or Attribute names. So, it is possible to perform searches for objects using partial name or value strings.

AttributeManager methods such as OpenAttributeSets return an AttributeSetsEnumerator object. This is a list of AttributeSets. The OpenAttributeSets method requires a list of objects for which you need AttributeSets. The AttributeSetsEnumerator returned has its AttributeSets ordered in the same way as your list of objects, therefore you can easily match which AttributeSet relates to which object by their numerical index. This technique can help to improve performance with large numbers of attributes.

The following code assumes a PartDocument object containing a single part, which in turn has faces to which attributes are attached. This example uses OpenAttributeSets to find AttributeSet "Test" on these face objects, and prints out the value of the first attribute in the AttributeSet. This would be much quicker than opening each face.

```
Dim oDoc As PartDocument
Set oDoc = ThisApplication.ActiveDocument

Dim oFaceColl As ObjectCollection
Set oFaceColl = ThisApplication.TransientObjects.CreateObjectCollection
For Each oFace In oDoc.ComponentDefinition.SurfaceBodies.Item(1).Faces
  oFaceColl.Add oFace
Next

Dim oSet As AttributeSet
Dim oSetEnum As AttributeSetsEnumerator

Set oSetEnum= oDoc.AttributeManager.OpenAttributeSets(oFaceColl,"Test")
For Each oSet In oSetEnum
  If oSet.Count > 0 Then
    Debug.Print oSet.Item(1).Value
```

```
  End If
Next
```

Other methods supported by the AttributeManager object include FindObjects. As its name suggests, this method returns a collection of objects corresponding to specified AttributeSet or Attribute names, or attribute values.

Performance

Conceptually, attributes are appended to persistent objects in Autodesk Inventor. In reality, the link is rather more tenuous. Autodesk Inventor is a parametric modeler; a uniform system for parts and assemblies. Objects in Inventor easily change, whether driven by parameters or as a result of a modeling operation such as the addition of a hole. So a persistent object such as a face is not necessarily as persistent or consistently identifiable as it first appears.

Autodesk Inventor has a complex search algorithm to enable it to consistently identify objects throughout their lifetime. This mechanism is employed by ReferenceKeys, a concept similar to handles in AutoCAD. This internal ReferenceKeys mechanism provides a reliable and robust link between an AttributeSet and the object to which it is appended. However, the link is potentially expensive in terms of processing time, particularly when you are working with large numbers of Attributes. There are several ways to mitigate the effects of this performance issue.

- Autodesk Inventor builds a cache of object-to-AttributeSet links as these links are referenced. This cache greatly reduces access time for subsequent references. You can force Autodesk Inventor to build the entire cache, for all objects, by calling the FindObjects method with no arguments. Then perform your query. Note though that continued modeling operations will destroy the cache.
- When working with large numbers of Attributes, enclose your Attribute editing in a Transaction or ChangeProcessor. This tells Autodesk Inventor not to use its own automatic transaction and undo mechanism, greatly improving performance.

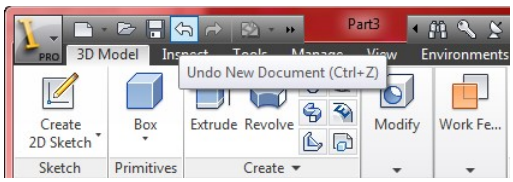
Also consider

Attributes are often used in conjunction with ReferenceKeys. Use Referencekeys to obtain a persistent identifier for a given Autodesk Inventor object. This identifier is valid from session to session. For example, an application can establish a relationship between objects in an assembly by storing a ReferenceKey for one object in a persistent attribute on another.

Transactions

Introduction to Transactions

The transaction functionality is the means by which Inventor keeps track of the timeline of changes made during the course of an Inventor session. Transactions allow the user to perform Undo and Redo operations in Inventor. The menu below shows the user interface command to undo the creation of a new document.



What Are Transactions?

All commands that modify data in Inventor are transactable. This means that Inventor model data created or modified during the course of a command can be undone and redone. When a command changes data in the database, it gathers some information from the user and begins a transaction. Only then does the command attempt to change the data in the database; if all the changes succeed, it asks the transaction manager to commit the changes. If the command fails to change the database successfully, it asks the transaction manager to abort the transaction and displays an error message to the user.

Inventor maintains two lists of transactions to manage undo and redo operations. The lists are called the committed transaction list and the undone transaction list. While a transaction is being recorded it is said to be in the "uncommitted" state. When the recording of the transaction is complete it is said to be in the "committed" state. If the user cancels the recording of the transaction, the transaction is "aborted." As soon as the transaction is committed, it is added to the committed list.

A successful transaction can be undone; an undone transaction can be redone. If the user undoes a committed transaction, the transaction is moved from the committed list to the undone list. If the user redoes an undone transaction it is once again moved from the undone list to the committed list. Inventor limits the number of transactions allowed in the committed list by a registry entry called "UndoLevels." The default value for "UndoLevels" is 10. Once the number of transactions exceeds the undo level, Inventor deletes the earliest transaction in the committed transaction list. When a transaction is committed, any undone transactions are no longer available to be redone (the undone list is emptied). Another important thing to note is that transactions are not persisted and are valid only for one Inventor session. This means that when a document closes, all the transactions in both the committed and the undone lists will be deleted.

Transactions within Inventor are quite different from transactions in some other systems (such as AutoCAD), so it is important to understand these differences before designing your application so you can take them into account. A critical thing to understand about Inventor's transactions is that the transaction stack has an application scope. In other words, all of the transactable actions performed within Inventor are saved in a single transaction list, regardless of which document the action was performed within. The API itself can be misleading in this regard in that it requires you to supply a document when starting a transaction, though the document name is not actually used internally.

There are two things to be aware of that result from Inventor's transactions having an application scope. The first is that transactions cannot be undone or redone with respect to a particular document. For example, suppose you have Part1 open and you draw two lines, then open Part2 and draw a circle. You now activate Part1 and perform an undo operation, expecting the last line you drew in Part1 to be undone. Instead the circle in Part2 will be undone since it was the last transaction in the transaction list.

The second issue of having a single transaction list for the entire application is that the entire list is sometimes deleted. This happens whenever a document is closed (it doesn't matter if the document is visible or not). This results in the user losing the ability to undo any previous actions.

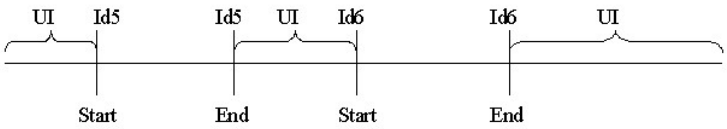
What is Transacted?

The basic principle is to transact changes to data, but not UI changes or view changes. Thus undoing a command that added a feature will remove the added feature. Operations such as adding or removing a toolbar, changing the viewing position, changing the zoom factor or shading the model are not transacted. Changing these things does not create a transaction, so the change cannot be undone. This can sometimes lead to unexpected results. For example, you could extrude a long part, pan to view the end of the part, do an undo and find yourself looking at nothing.

Document Opens are transacting events. This means if you open a document and then do an undo, the document is undone from existence. An immediate redo would bring back the document. Document closes do not transact. When a document is closed, all of its memory is 'blown away' and is completely removed from the transaction chain.

Note that whether something is transacted is strictly dictated by whether it uses Inventor's transactable memory or not. This does not determine whether the object is persisted; persisted information can be allocated inside or outside of Inventor's transactable memory.

Identified and Unidentified Transactions



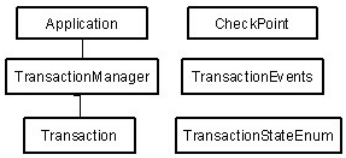
All changes to transacting memory transact. However, many of the changes to this data fall outside commands that start transactions. For example, after a command has completed, views of parts need to be repainted on the computer screen. Windows puts a paint request in a queue, and the repainting happens later, when the transaction has been committed.

Although logically painting a part on the screen does not update the part, graphics information is recalculated and stored in the Scene Graph. These memory changes need to be transacted (so they can be undone). These changes are stored as part of an unidentified transaction.

An unidentified transaction is, for all intents and purposes, a normal transaction that the user isn't aware exists. Undoing or redoing a command always undoes an 'identified' transaction, but it may also undo or redo unidentified transactions. It is important to note that as far as Inventor is concerned, there is always a transaction in progress. If the user does not start an "identified" transaction, Inventor will create an "unidentified" transaction. As the figure above shows, the transaction timeline for Inventor is a continuum, beginning with the start of the Inventor session and ending when the session ends. The timeline shows the identified transactions interspersed with unidentified transactions (labeled "UI" in the figure). The figure may seem to imply that identified and unidentified transactions always occur in pairs, but this is not always the case. If an identified transaction aborts, it will revert the timeline back to the end of the last unidentified transaction. Consequently another unidentified transaction will ensue.

Unidentified transactions are hidden from the API. The API for transactions presents only the identified transaction view. The only way to observe an unidentified trisection via the API would be to watch the CurrentTransaction property of the TransactionManager, when no command is in progress. For example, start a new sketch and observe TransactionManager::CurrentTransaction. It would report that the DisplayName of the transaction is "Unidentified transaction."

Transactions in the API



The transaction API gives you the ability to leverage Inventor's transaction scheme to create custom commands or operations that work like Inventor's commands. The following topics describe the various facilities offered by the transaction API and how they should be used.

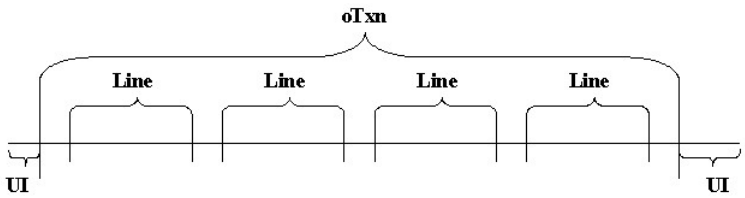
Using Transactions

The most basic use of the transaction API is to wrap a set of transactable operations in a transaction. An example would be a client who wishes to publish a command that creates a sketch rectangle by connecting four sketch lines. Such a command would take two input points from the user and create four sketch lines. From Inventor's perspective this command should behave like the creation of a rectangle and not the creation of four lines; i.e. an Undo operation after the command has been executed should undo the creation of the entire rectangle rather than one of its lines. The sample program below demonstrates the use of transaction API to create such a command:

```
' Get a reference to the active document.
' This can be an Assembly or Part document.
Dim oDoc As Document
Set oDoc = ThisApplication.ActiveDocument

Dim oCmpDef As PartComponentDefinition
Set oCmpDef = oDoc.ComponentDefinition

Dim oSketch As PlanarSketch
Set oSketch = oCmpDef.Sketches(1)

Dim oTG As TransientGeometry
Set oTG = ThisApplication.TransientGeometry
```

```
' Get the transaction manager from the application
Dim oTxnMgr as TransactionManager
Set oTxnMgr = ThisApplication.TransactionManager

' Start a regular transaction
Dim oTxn1 As Transaction
Set oTxn = oTxnMgr.StartTransaction(oDoc, "My Rectangle Command")

' Draw four sketch lines
Dim oLine As SketchLine
Set oLine = oSketch.SketchLines.AddByTwoPoints(oTG.CreatePoint2d(0, 0), oTG.CreatePoint2d(1, 0))
Set oLine = oSketch.SketchLines.AddByTwoPoints(oLine.EndSketchPoint, oTG.CreatePoint2d(1, 2))
Set oLine = oSketch.SketchLines.AddByTwoPoints(oLine.EndSketchPoint, oTG.CreatePoint2d(0, 2))
Set oLine = oSketch.SketchLines.AddByTwoPoints(oLine.EndSketchPoint, oTG.CreatePoint2d(0, 0))

oTxn.End
```

Creating the "My Rectangle Command" transaction wraps the four sketch lines in a transaction. The transaction timeline for this command is illustrated by the figure below:



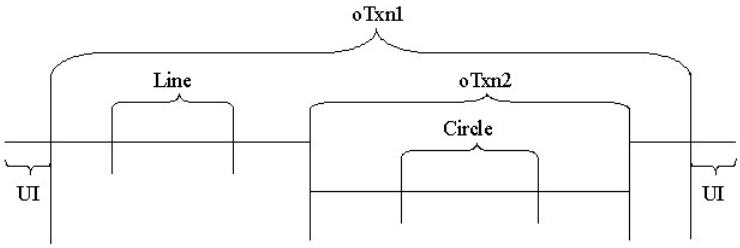
Each AddByTwoPoints method creates a Line transaction in the scope of oTxn. Abort of oTxn (calling Abort before End) at any point will revert oTxn back to the beginning. If any of the Line transactions abort due to an error condition, the transaction timeline will revert to the beginning of the Line.

Parent and Child Transactions

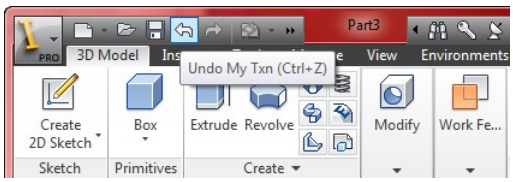
A transaction that is started using the TransactionManager::StartTransaction method can either be a parent or a child transaction. A transaction that is started within the scope of another transaction becomes the child of that transaction. Consider the sample program below:

```
' Get a reference to the active document.
' This can be an Assembly or Part document.
Dim oDoc As Document
Set oDoc = ThisApplication.ActiveDocument

Dim oCmpDef As PartComponentDefinition
Set oCmpDef = oDoc.ComponentDefinition

Dim oSketch As PlanarSketch
Set oSketch = oCmpDef.Sketches(1)

Dim oTG As TransientGeometry
Set oTG = ThisApplication.TransientGeometry

' Get the transaction manager from the application
Dim oTxnMgr as TransactionManager
Set oTxnMgr = ThisApplication.TransactionManager

' Nesting regular transactions
' Start a regular transaction
Dim oTxn1 As Transaction
Set oTxn1 = oTxnMgr.StartTransaction(oDoc, "My Txn")

    ' Draw a sketch line
    Dim oLine As SketchLine
    Set oLine = oSketch.SketchLines.AddByTwoPoints(oTG.CreatePoint2d(0, 0), oTG.CreatePoint2d(1, 0))

    ' Start a nested transaction
    Dim oTxn2 As Transaction
    Set oTxn2 = oTxnMgr.StartTransaction(oDoc, "My child Txn")

    ' Draw a circle
    Dim oCircle As SketchCircle
    Set oCircle = oSketch.SketchCircles.AddByCenterRadius(oLine.EndSketchPoint, 3)

    oTxn2.End

oTxn1.End
```

The above code starts an identified transaction (oTxn1), creates a sketch line (a transaction), starts another transaction (oTxn2), creates a sketch circle (a transaction) and commits oTxn1 and oTxn2. The transaction timeline looks like this:



What this indicates is that oTxn has three child transactions: Line, oTxn2 and Circle. Transaction oTxn2 has one child: Circle. The fact that oTxn2 wraps the Circle transaction means that an abort of Circle will revert the timeline to the beginning of oTxn2. Abort of Line will revert the timeline to the beginning of oTxn. The child transactions, however, do not affect the Undo operation. In the example above, the only transaction visible to Inventor is the top-level transaction oTxn. In this example, Inventor's Undo user interface will look like the figure below.



Start, End, and Abort Transactions

Each start transaction should be paired with an end or abort transaction. Inventor becomes unstable if the StartTransaction is not paired with a corresponding End or Abort. If any of the API calls fail, i.e. return a bad error code, then the command should handle the error and take care to call EndTransaction. Applying this rule consistently is particularly critical, since even a simple API call can fail rather unexpectedly. In the event that the error is not recoverable, the command should abort the transaction. Aborting the transaction means that the transaction was cancelled prior to completion. The sample below indicates how an error condition should be handled within the scope of a transaction.

```
' Get a reference to the active document.
' This can be an Assembly or Part document.
Dim oDoc As Document
Set oDoc = ThisApplication.ActiveDocument

' Get the transaction manager from the application
Dim oTxnMgr as TransactionManager
Set oTxnMgr = ThisApplication.TransactionManager

' Start a transaction
Dim oTxn As Transaction
Set oTxn = oTxnMgr.StartTransaction(oDoc, "My Txn")

' Perform an operation that you wish to transact

' If the error from the operation is not recoverable, abort the Txn
If Err Then
    MsgBox "Unrecoverable error occurred during the operation"
    oTxn.Abort
    Exit Sub
End If

' End the transaction
oTxn.End
```

Checkpoints

Checkpoints are bookmarks for rollback operations within a transaction. Consider the sample program below. A transaction is initiated and this transaction performs two operations; create a complex profile and extrude the profile. A checkpoint has been placed before the extrude operation. What this means is that if the extrude command fails, the transaction can be rolled back to the checkpoint. By enabling such a rollback, the user is given an opportunity to recover from an error. In this case if the extrude operation failed due to a bad profile, an option can be presented to the user to modify the profile and try the extrude operation again.

A side note: Inventor implements checkpoints as child transactions. The GoToCheckpoint operation essentially aborts the checkpoint transaction.

```
' Get a reference to the active document.
' This can be an Assembly or Part document.
Dim oDoc As Document
Set oDoc = ThisApplication.ActiveDocument

' Get the transaction manager from the application
Dim oTxnMgr as TransactionManager
Set oTxnMgr = ThisApplication.TransactionManager

' Start a regular transaction
Dim oTxn1 As Transaction
Set oTxn1 = oTxnMgr.StartTransaction(oDoc, "Checkpoint Txn")

' ****************************************
' Perform the creation of extrude profile
' ****************************************

' Create a checkpoint before the extrude operation
```

```
Dim oChkPt as CheckPoint
Set oChkPt = oTxnMgr.SetCheckPoint

' ****************************************
' Extrude the newly created profile
' ****************************************

' Handle any error condition from the extrude command
If Err Then
    MsgBox "Extrude operation failed. Modify profile ?"
    oTxnMgr.GoToCheckPoint oChkPt
End If
```

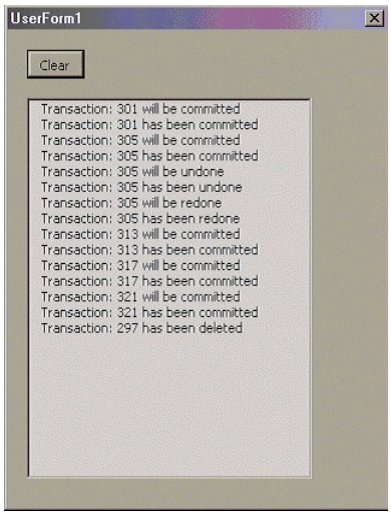
Transaction Events

The Transaction API allows the client to receive notifications for transaction-related events. The TransactionEvents object sends notifications for commit, undo, redo, abort and delete of a transaction. All of the notifications with the exception of the delete are sent before and after the actual event takes place in Inventor. Notification for the deletion of a transaction is sent only after the transaction has been deleted.

The following sample program demonstrates the use of transaction events:

```
Private WithEvents oTxnEvents As TransactionEvents

Private Sub clear_Click()
    List.Clear
End Sub

Private Sub UserForm_Initialize()
    Dim oTxnMgr As TransactionManager
    Set oTxnMgr = ThisApplication.TransactionManager

    Set oTxnEvents = oTxnMgr.TransactionEvents
End Sub

Private Sub oTxnEvents_OnAbort(ByVal TransactionObject As Transaction, ByVal Context As NameValueMap, ByVal BeforeOrAfter As EventTimi

    If BeforeOrAfter = kBefore Then
        List.AddItem "Transaction: " & TransactionObject.Id & " will be aborted"
    ElseIf BeforeOrAfter = kAfter Then
        List.AddItem "Transaction: " & TransactionObject.Id & " has been aborted"
    End If

End Sub

Private Sub oTxnEvents_OnCommit(ByVal TransactionObject As Transaction, ByVal Context As NameValueMap, ByVal BeforeOrAfter As EventTim

    If BeforeOrAfter = kBefore Then
        List.AddItem "Transaction: " & TransactionObject.Id & " will be committed"
    ElseIf BeforeOrAfter = kAfter Then
        List.AddItem "Transaction: " & TransactionObject.Id & " has been committed"
    End If

End Sub

Private Sub oTxnEvents_OnRedo(ByVal TransactionObject As Transaction, ByVal Context As NameValueMap, ByVal BeforeOrAfter As EventTimin

    If BeforeOrAfter = kBefore Then
        List.AddItem "Transaction: " & TransactionObject.Id & " will be redone"
    ElseIf BeforeOrAfter = kAfter Then
        List.AddItem "Transaction: " & TransactionObject.Id & " has been redone"
    End If

End Sub

Private Sub oTxnEvents_OnUndo(ByVal TransactionObject As Transaction, ByVal Context As NameValueMap, ByVal BeforeOrAfter As EventTimin

    If BeforeOrAfter = kBefore Then
        List.AddItem "Transaction: " & TransactionObject.Id & " will be undone"
    ElseIf BeforeOrAfter = kAfter Then
        List.AddItem "Transaction: " & TransactionObject.Id & " has been undone"
    End If

End Sub

Private Sub oTxnEvents_OnDelete(ByVal TransactionObject As Transaction, ByVal Context As NameValueMap, ByVal BeforeOrAfter As EventTim

    If BeforeOrAfter = kAfter Then
        List.AddItem "Transaction: " & TransactionObject.Id & " has been deleted"
    End If

End Sub
```

This program creates a form, shown below, to monitor the transaction activity in Inventor. The transaction activity is reported using the transaction ID. The output was generated by creating several sketch lines and doing undo and redo on the sketch lines.



Dos and Don'ts

- An Undo or Redo should be performed only when there is no transaction in progress (during an unidentified transaction).
- Do not wrap any Inventor commands within transactions. Do not wrap any user interaction events within transactions. A good rule would be that it is OK to create modal dialogs within a transaction but not modeless dialogs.
- The transaction event handling code should not perform any operation that uses transactable memory. For example, it is illegal to create a sketch line in the event handling code for the commit of a transaction.
- Do not close any document inside a transaction, even the active one. The exception to this rule is that it is legal to close a document in a transaction when the document was opened in that exact same transaction.

```
Dim oTxnMgr As TransactionManager
Set oTxnMgr = ThisApplication.TransactionManager

Dim oDoc As Document
Set oDoc = ThisApplication.ActiveDocument

Dim oTxn As Transaction
Set oTxn = oTxnMgr.StartTransaction(oDoc, "Txn1")

' ************************************
' Invalid operation
' ************************************
oDoc.Close

oTxn.End
```

Change Processor

Introduction to the change processor

Autodesk Inventor users can undo actions. They can undo edits that have changed the document. They can redo their undo - in other words, they can recommit changes they had previously undone. These actions are evident in the user interface. The Edit menu indicates the last command that can be undone, as well as any command that can be redone.

For developers using the Autodesk Inventor API, the change processor can be used to achieve the same thing.

The purpose of the change processor

Developers typically use the API transaction mechanism (TransactionManager and associated objects) to group actions atomically, and to commit or roll back those actions. This works well in the context of the application, but it is not well integrated into the user interface, and is distinct from the Autodesk Inventor internal transaction mechanism.

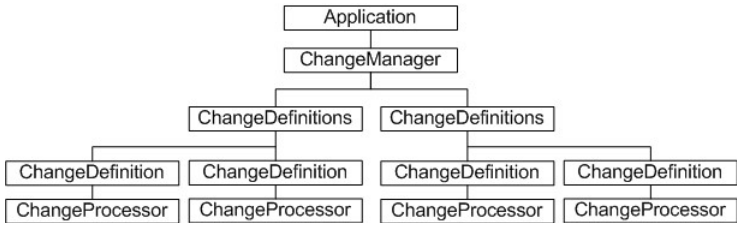
Ideally, a mechanism should exist to allow Autodesk Inventor to take care of transacting developer-defined actions, and to integrate any roll-back and commit actions into its user interface. This is the purpose of the change processor.

An action encapsulated in a change processor automatically benefits from Autodesk Inventor's transaction mechanism as a named action. It also automatically becomes part of the Autodesk Inventor undo and redo sequence, with the named action visible through the user interface.

The change processor object also has methods for writing and reading script strings. Developers can use these strings to persist application-specific actions.

The change processor is the preferred transaction mechanism in most cases, when compared to transactions administered through the transaction manager API.

The change processor object model diagram



Working with the change processor API

The change processor relies on event callbacks. The developer code to be transacted (the drawing of sketches, creating features and so on) is started as a result of a call to the execute method of the change processor. The change processor object is created by the CreateChangeProcessor method of the ChangeDefinition object, which is the named atomic unit subject to Autodesk Inventor's undo and redo mechanism.

There can be any number of change definition objects, contained by the ChangeDefinitions collection object. In addition, the ChangeManager object can create any number of ChangeDefinitions collections. Each ChangeDefinitions object is associated with a unique string for the purposes of identification. This string can be anything, but most often will be the unique CLSID of the application.

The code to be transacted is executed by calling the execute method of the ChangeProcessor encapsulating that code. There can be only one ChangeProcessor object created by the ChangeDefinition object, but that ChangeProcessor object can be deleted and recreated. As an example, consider code to be executed as the result of a button press. The button may never be pressed, so the ChangeProcessor need not be created until it is, and can be deleted afterwards. The ChangeDefinition is always there, ready to create the ChangeProcessor. This is why the code to be transacted is executed as an event callback. The ChangeProcessor is created on the fly, and its execute method is called explicitly.

A ChangeProcessor in action

The following sample first defines a class comprised of three subroutines and some global data. This class is later instantiated and responds to button and change processor events.

This sample omits error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

Create a new VBA project and add a new class module named clsCreateLine. Add the following code to the general declarations section of the new class module. This code defines the change definition, change processor, and button definition as global objects able to respond to events.

```
Option Explicit
Private WithEvents mobjChangeDef As ChangeDefinition
Private WithEvents mobjChangeProcessor As ChangeProcessor
Private WithEvents mobjButtonDefinitionEvents As ButtonDefinition
```

Now add the first of the three subroutines to the class. This one is called when the class is instantiated and is where initialization code should be located. The handler for the button is created here.

```
Private Sub Class_Initialize()
    Dim objCommandMgr As CommandManager
    Set objCommandMgr = ThisApplication.CommandManager

    Dim colControlDefs As ControlDefinitions
    Set colControlDefs = objCommandMgr.ControlDefinitions

    Dim objButtonDef As ButtonDefinition
    Set objButtonDef = colControlDefs.Item("CreateLine")

    Set mobjButtonDefinitionEvents = objButtonDef
End Sub
```

The second subroutine to add to the class defines what happens when the button is pressed. This is where the change processor is created from the existing ChangeDefinition object, and its execute method is called.

```
Private Sub mobjButtonDefinitionEvents_OnExecute(ByVal Context As NameValueMap)

    Dim objChangeMgr As ChangeManager
    Set objChangeMgr = ThisApplication.ChangeManager

    Dim colChangeDefs As ChangeDefinitions

    Set colChangeDefs = objChangeMgr.Item("My_unique_CP_ID")
    Set mobjChangeDef = colChangeDefs.Item("CreateLine")
    Set mobjChangeProcessor = mobjChangeDef.CreateChangeProcessor

    Dim objActiveDoc As Document
    Set objActiveDoc = ThisApplication.ActiveDocument

    mobjChangeProcessor.Execute objActiveDoc
End Sub
```

The third and last subroutine to add to the clsCreateLine class definition defines what happens when the execute event of the change processor is called. This is the transacted code. The following example adds a sketch line to the open part document.

```
Private Sub mobjChangeProcessor_OnExecute(ByVal Document As Document, ByVal Context As NameValueMap, Succeeded As Boolean)
    Dim objPartDoc As PartDocument
    Set objPartDoc = Document

    Dim objPartCompDef As PartComponentDefinition
    Set objPartCompDef = objPartDoc.ComponentDefinition

    Dim objTransGeom As TransientGeometry
```

```
    Set objTransGeom = objPartDoc.Parent.TransientGeometry

    Dim colLine As SketchLine
    Set colLine = objPartCompDef.Sketches(1).SketchLines.AddByTwoPoints _
        (objTransGeom.CreatePoint2d(0, 0), objTransGeom.CreatePoint2d(4, 0))
End Sub
```

This completes the class definition. Now add the code that instantiates this class. Add a new code module to the modules section of your VBA project. First, add a global declaration of the new class to the general declarations section, as follows.

```
Option Explicit
Public mobjCreateLine As clsCreateLine
```

Next, add the public subroutine that will be called to run this application. This does three things. It calls the subroutine to set up the ChangeDefinition object, and it calls the subroutine to add a button to the command panel. Both subroutines are defined in this module. Lastly, it instantiates the clsCreateLine class defined previously.

```
Sub CreateLineUsingChangeProcessor()
    Call AddCreateLineChangeDef
    Call AddLineButton
    Set mobjCreateLine = New clsCreateLine
End Sub
```

Now add the subroutine that creates the ChangeDefinition object. This object is created through the Add method if the ChangeDefinitions object, which is created through the Add method of the ChangeManager object. A unique ID is associated with the ChangeDefinitions object, to aid in identification elsewhere in the application. Similarly, the ChangeDefinition is named too, however this information is used in the user interface. The 'Create Line' text in the following code will be displayed in the Autodesk Inventor Edit menu as the last command that can be undone. If undone, it will display as the command that can be redone.

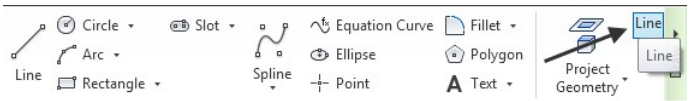
```
Private Sub AddCreateLineChangeDef()
    Dim objChangeMgr As ChangeManager
    Set objChangeMgr = ThisApplication.ChangeManager

    Dim colChangeDefs As ChangeDefinitions
    Set colChangeDefs = objChangeMgr.Add("My_unique_CP_ID")

    Dim objDhangeDef As ChangeDefinition
    Set objDhangeDef = colChangeDefs.Add("CreateLine", "Create Line")
End Sub
```

Finally, add the subroutine that adds the button to the command panel. This uses the standard user interface customization API. User interface customization is covered elsewhere in the Autodesk Inventor programming Help.

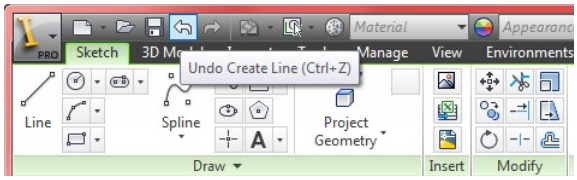
```
Private Sub AddLineButton()
    Dim objCommandMgr As CommandManager
    Set objCommandMgr = ThisApplication.CommandManager

    Dim colControlDefs As ControlDefinitions
    Set colControlDefs = objCommandMgr.ControlDefinitions

    Dim objButtonDef As ButtonDefinition
    Set objButtonDef = colControlDefs.AddButtonDefinition("Line", _
        "CreateLine", kShapeEditCmdType, "", "Create Line", "Line")

    Call ThisApplication.UserInterfaceManager.Ribbons("Part").RibbonTabs("id_TabSketch").RibbonPanels("id_PanelP_2DSketchDraw").Comman

    objButtonDef.Enabled = True
End Sub
```

Once all the preceding code is added, the VBA project browser should appear as follows.

Start a new part document in Autodesk Inventor, and run the CreateLineUsingChangeProcessor subroutine. A new Line button is added to the command panel as follows:



Pressing this button causes a sketch line to be created in the part document. Now look at the Edit menu. Notice the ChangeProcessor named Create Line is available for undo. Pressing the Undo button causes the sketch line to disappear, and the Create Line ChangeProcessor is available for redo. The menu appears as follows:



Summary

Use the change processor API to take advantage of Autodesk Inventor's own transaction and undo/redo mechanism. It is no longer necessary to explicitly define the start, end, and nesting of transactions. Developer code can be executed as an atomic named unit, and can be undone and redone through the Autodesk Inventor user interface.

Also consider

For cases where a fine degree of granular control is required, the transaction API may still be appropriate. The developer can nest transactions in a specific manner and add checkpoints for partial rollback.

Apprentice Server

Using Apprentice Server

The simple way to think of Apprentice is that it's a smaller version of Autodesk Inventor that does not have a user interface. Without a user interface the only way to access its functionality is by using its API. Apprentice is actually an ActiveX component. It runs within the process space of the client that's using it. For example, if you write a Visual Basic program that displays information about the contents of an assembly, Apprentice will be running within your VB program's process space. Apprentice can be very efficient for certain applications because it's smaller than the complete Autodesk Inventor application and because it runs in the same process as your application.

Note: Apprentice is not supported within Inventor itself. Do not run it in an Inventor addin nor in a VBA macro.

The API exposed by Apprentice is a subset of the complete Autodesk Inventor API. Apprentice provides access to file references, assembly structure, B-Rep, geometry, attributes, render styles, and document properties. Access to the assembly structure, B-Rep, geometry, attributes and render styles is read-only. Access to file references and document properties is read and write. In past releases of the product, a type library specifically for Apprentice was delivered with both Autodesk Inventor and Apprentice. This type library contained the limited set of objects supported only by Apprentice. Now, however, since the Autodesk Inventor type library contains all of the Apprentice functionality, this should be used in an Apprentice context. A version of the old Apprentice type library is still supplied for legacy reasons, but this will likely be discontinued in future releases.

Some common types of applications that will make use of Apprentice are larger standalone applications that want to be able to use Autodesk Inventor files. For example, an existing NC application could use Apprentice to directly read in an Autodesk Inventor part or assembly. If it were to use Autodesk Inventor for this it would need to start Autodesk Inventor first, use Autodesk Inventor's API to open the desired document, and then use Autodesk Inventor's API to query for necessary information. There's a lot of unnecessary overhead in having to start Autodesk Inventor and a significant performance penalty caused by all of the API calls going between two processes. These problems are solved with Apprentice since it is very lightweight and runs in the process space of the application.

Another feature of Apprentice is the cost. Apprentice is freely available and is distributed as part of Design Tracking, which is available for download from the Autodesk website. Much of Design Tracking actually uses Apprentice.

A simpler example of the use of Apprentice is a small application that updates the cost property of an Autodesk Inventor part document based on the current cost that is stored in a business database. In this case Apprentice is being used to write the document properties of a part. The utility can quickly go through a set of Autodesk Inventor part documents and set the cost property with a value it obtained from a database.

Most of the features that Apprentice supports are identical to those in Autodesk Inventor. For example, to traverse an assembly you would perform the traversal the same way in Apprentice as you would in Autodesk Inventor. The basic traversal function can be written to be used for both cases. Because most of the API supported by Apprentice is the same as Autodesk Inventor's, this section on Apprentice will focus on the differences between the two.

The primary differences between Autodesk Inventor and Apprentice are in the Application and Document objects. The objects that represent the Application and Document are completely different in Autodesk Inventor and Apprentice. The Apprentice Application object is called ApprenticeServerComponent. It supports a much more limited API than the Inventor Application object. In Apprentice there isn't a Documents collection.

When ApprenticeServer.Open opens a document, a reference to that document is held by the ApprenticeServer component in order to be returned from the ApprenticeServer.Document. This ‘active top/last opened’ document is also considered the document used as the root of the save for the FileSaveAs object. A document is closed when either ApprenticeServerDocument.Close is called, or the document’s reference count fully drops to zero. Keep in mind that the reference held by the ApprenticeServer (if it was the last document to be opened) also counts as a reference. This ApprenticeServer reference will be released either when a different document is opened, or ApprenticeServer.Close is called, or when the ApprenticeServer’s reference count fully drops to zero and it is destroyed.

The ApprenticeServerComponent supports a few methods and properties that are unique to Apprentice. These are the Open and Close methods, which are used to open and close documents within Apprentice; DisplayAffinity, which is used to optimize the behavior of Apprentice for viewer applications; MinimizeFileSize, which compresses files by removing versions, and FileSaveAs, which we'll discuss in more detail later in this section.

The document objects used within Apprentice are different from the document objects used in Autodesk Inventor. In Inventor there are the PartDocument, AssemblyDocument, DrawingDocument, and PresentationDocument objects. In Apprentice, the ApprenticeServerDocument object represents the part, assembly, and presentation documents and the ApprenticeServerDrawingDocument represents the drawing document. The code below illustrates using Apprentice to open a document.

```
Private Sub TestApprentice()
    ' Create a new instance of Apprentice.
    Dim oApprentice As New ApprenticeServerComponent
```

```
    ' Open a document.
    Dim oDoc As ApprenticeServerDocument
    Set oDoc = oApprentice.Open("C:\Temp\Assembly1.iam")
End Sub
```

The "New" keyword is used in the declaration of the variable for the ApprenticeServerComponent. This creates a new instance of an object of ApprenticeServerComponent type. Apprentice isn't actually loaded at this point, but is loaded the first time the variable is used--in this case, when the Open method of Apprentice is called.

Even though there are differences between the application and document objects of Autodesk Inventor and Apprentice, once you get past these top-level objects, the objects below them in the hierarchy are the same. For example, a function that extracts information from the B-Rep or from an assembly can be used with Autodesk Inventor or Apprentice without any changes. The function below will display an assembly tree given a ComponentOccurrences object regardless of whether it is used in Apprentice or Autodesk Inventor.

```
Private Sub GetComponents(InCollection As ComponentOccurrences, _
Level As Long)
    ' Iterate through the components in the current collection.
    Dim oCompOccurrence As ComponentOccurrence
    For Each oCompOccurrence In InCollection
        ' Display information about the current component.
        Debug.Print Space(Level * 3) & oCompOccurrence.Name

        ' Recursively call this function for the suboccurrences
        ' of the current component.
        Call GetComponents(oCompOccurrence.SubOccurrences, Level + 1)
    Next
End Sub
```

Here's a modified version of the earlier sample that connects to Apprentice and opens a document. It's been updated to check for the document type and then call the GetComponents function above.

```
Private Sub TestApprentice()
    ' Create a new instance of Apprentice.
    Dim oApprentice As New ApprenticeServerComponent

    ' Open a document.
    Dim oDoc As ApprenticeServerDocument
    Set oDoc = oApprentice.Open("C:\Temp\Assembly1.iam")

    ' Display this document's name.
    Debug.Print oDoc.DisplayName

    ' Check to make sure the document is an assembly.
    If oDoc.DocumentType = kAssemblyDocument Then
        ' Show the occurrence tree.
        Call GetComponents(oDoc.ComponentDefinition.Occurrences, 1)
    End If
End Sub
```

Because Apprentice provides read-only access to some of the information in an Autodesk Inventor document, some of the properties on objects may fail when you attempt to set them. For example, the Name property of the ComponentOccurrence object behaves as read-only in Apprentice, whereas in Autodesk Inventor it is read-write. The B-Rep portion of the API behaves identically in Autodesk Inventor and Apprentice since it provides query-only access in both cases.

There are a few things that can only be done using Apprentice. These are all things that cause problems when the file is open in Inventor but can be easily accomplished when accessed through Apprentice. One of the most important of these is the ability to change file references. This is a critical part of Apprentice that Design Assistant uses. For example, let's say you have an assembly that contains Part1 and Part2. Part1 needs to be revised so you make a copy of the file, give it a new name, make the changes to the part, and save it. When you open the assembly it's still referencing the old file. The file reference portion of the API allows you to change the reference within the assembly so that it's pointing to the file that represents the revised version of Part1.

The FileSaveAs object, which is only available in Apprentice, provides functionality that lets you save components of an assembly to different filenames and update the assembly's references to point to the new files.

Using the FileSaveAs object you can save the document you currently have open. If you make any changes to the document you must use the FileSaveAs object to save those changes, otherwise it's equivalent to exiting the file without saving. The sample below opens an assembly, looks for a reference to a specific file, changes the reference to point to another file, and then saves the change.

```
Private Sub ChangeReferenceSample()
    Dim oApprentice As New ApprenticeServerComponent

    ' Open a document.
    Dim oDoc As ApprenticeServerDocument
    Set oDoc = oApprentice.Open("C:\Temp\Assembly1.iam")

    ' Iterate through the references looking for a
    ' reference to a specific file.
    Dim oRefFileDesc As ReferencedFileDescriptor
    For Each oRefFileDesc In oDoc.ReferencedFileDescriptors
        If oRefFileDesc.FullFileName = "C:\Temp\OldPart.ipt" Then
            ' Replace the reference.
            Call oRefFileDesc.PutLogicalFileNameUsingFull( _
                                        "C:\Temp\NewPart.ipt")
            Exit For
        End If
    Next

    ' Set a reference to the FileSaveAs object.
    Dim oFileSaveAs As FileSaveAs
```

```
    Set oFileSaveAs = oApprentice.FileSaveAs

    ' Save the assembly.
    Call oFileSaveAs.AddFileToSave(oDoc, oDoc.FullFileName)
    Call oFileSaveAs.ExecuteSave
End Sub
```

If you're using Apprentice to access the document properties, and that's the only edit you've made to the document, you don't need to use the FileSaveAs object to save the file, but can use the FlushToFile method of the PropertySets object to save the property changes. This is much more efficient because it only writes the properties to the document and doesn't write the entire document.

Please note: Saving of files in Apprentice is not allowed on files that require migration (any file that has not already been migrated to the same version of the Apprentice server). The current document's NeedsMigrating property must return False before the FileSaveAs object can successfully be returned.

How to get Apprentice Server installed on machine?

To access Apprentice API you should have Apprentice Server installed on your machine, there are three ways to have Apprentice Server installed on your machine:

1. Install Inventor. When you have Inventor installed, you have Apprentice Server installed along with it.
2. Install Apprentice Server standlone installer(Since Inventor Apprentice Server 2022). The Autodesk website will contain download link for Inventor Apprentice Server standlone installer since Inventor Apprentice Server 2022.
3. Install Inventor View. When you have Inventor View installed, you have Apprentice Server installed along with it.

Client Views

Introduction to client views

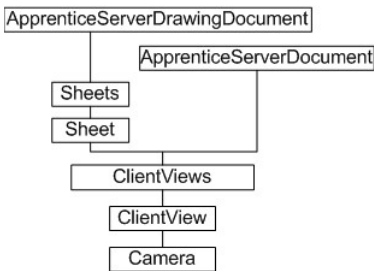
A subset of the Autodesk Inventor API, with a few additions, is provided for developers wishing to work with Autodesk Inventor documents outside of the Autodesk Inventor environment. This API is provided through Apprentice Server - an ActiveX server that can be incorporated into other applications. Apprentice Server has no user interface, and most of its API is read-only.

One part of the Apprentice Server API that has no direct equivalent in Autodesk Inventor is the ClientView object. When writing an application outside of Autodesk Inventor, the ClientView object provides a convenient means of displaying an Autodesk Inventor document in a similar manner to Autodesk Inventor itself.

The purpose of client views

The ClientView object allows the developer to associate an Autodesk Inventor document with a window handle in their standalone application, and to manipulate the view of the document through the use of the Camera object. The view orientation can be modified, as can the perspective, extents, and mouse-driven applied transformation. A client view can provide a visually appealing graphic with minimal effort on the part of the developer.

ClientView object model diagram



Working with client views through the API

Creation of a new ClientView object is only possible through Apprentice Server. It is not currently possible to add a new ClientView object within the Autodesk Inventor environment. For example, using VBA to call the Add method of a ClientView object obtained through the Sheet object of a Drawing document will fail.

The ClientView object is always associated with a window handle (hWnd). The ClientViews collection object provides two methods for creating a new ClientView object; Add and AddBySubset. The latter method is likely to be depreciated. The Add method associates a document client view with a given window handle.

Setting up a client view in Apprentice Server

The following example uses Visual Basic, though any fully ActiveX-compliant environment should work. The code omits error checking for brevity and clarity. Always check that return values are of the expected type and in the expected range.

In Visual Basic, add a PictureBox control to a new form. Keep the default control name Picture1. Double-click the control to enter the code module, and add the following code in the declarations section:

```
Dim oSvr As ApprenticeServerComponent
Dim oAppDoc As ApprenticeServerDocument
Dim oClientView As ClientView
```

At this point, Visual Basic has no idea what an ApprenticeServerComponent is. A reference to the Apprentice Server type library is needed. In the Visual Basic user interface, select Project, then References. In the list of available references, put a check against the item labeled Autodesk Inventor's Apprentice Object Library. This points to the RxApprentice.tlb type library.

Continue adding code. In the Form_Load sub, add the following:

```
Private Sub Form_Load()
  Dim filename As String
  filename = "C:\Autodesk\Inventor 10\Tutorial Files\analyze-2.iam"
  Set oSvr = New ApprenticeServerComponent
  Set oAppDoc = oSvr.Open(filename)
  Set oClientView = oAppDoc.ClientViews.Add(Picture1.hWnd)
End Sub
```

The preceding code references an assembly file. Change this reference to suit. An instance of the ApprenticeServerComponent is created, and the assembly document is opened by Apprentice Server. The last line adds a new ClientView object to the document's ClientViews collection, and references the window handle (hWnd) of the Picture1 control.

In the Picture1_Click sub, add the following code:

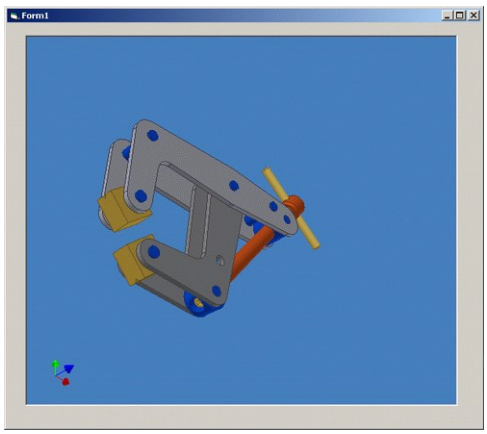
```
Private Sub Picture1_Click()
  Dim oCamera As Camera
  Set oCamera = oClientView.Camera
  oCamera.ViewOrientationType = kIsoBottomLeftViewOrientation
  oCamera.Perspective = True
  oCamera.Apply
  Me.Refresh
End Sub
```

The preceding code responds to a click event on the Picture1 control, setting various camera properties before refreshing the view. The ISO orientation type is set to bottom left, and perspective viewing is enabled.

Lastly, add the following code to the Form_Paint event. This is to ensure the view is maintained if the window is changed in any way. The Update method takes a Boolean argument; true if the view is dynamically updated, to improve performance.

```
Private Sub Form_Paint()
  oClientView.Update (False)
End Sub
```

The following figure shows the results of running the sample code and clicking on the picture control:



ClientViews, the ClientViews ActiveX control, and Autodesk Inventor Read-only mode

The ClientView object in the Apprentice Server API provides a convenient means of displaying Autodesk Inventor documents outside of Autodesk Inventor, but it does require Apprentice Server. For how to get Apprentice Server, please refer to: **Apprentice Server**.

Another option available to the developer is a ClientView OCX control. This control can be embedded in applications or HTML pages. It does not require Autodesk Inventor or Apprentice Server, and provides a similar level of functionality as the ClientView API. The control is named InventorViewerCtrl.ocx, and is located in Autodesk Inventor's Bin subdirectory. Its ProgID is Inventor.ViewControl, and it exposes a number of properties, including the following:

AboutBox : Causes the about box associated with the control to be displayed.
ActiveViewingCommand : Gets and sets the currently active command within the view control (ViewingCommandEnum).
ApprenticeServerDocument : Gets the document the view was created from.
ClientView : Returns the ClientView object that is currently being used by the viewer control.
DisplayMode : Gets or sets the display mode (DisplayModeEnum).
Filename : Gets or sets the name of an Assembly/Part/Drawing file to open.
HideToolbar : Controls whether the toolbar is displayed when the end-user moves the mouse over the viewer control.

Interactive : Gets or sets a value that determines whether the control can be manipulated interactively.
Perspective : Gets or sets whether the camera is in perspective mode.
SheetIndex : Gets or sets the index of the current sheet. Only for drawing files.
ViewOrientationType : Gets the view orientation type (ViewOrientationTypeEnum).

A full installation of Autodesk Inventor will also install Autodesk Inventor Read-only mode. This is a user application for viewing Autodesk Inventor files. Autodesk Inventor Read-only mode is distinct from the OCX control, and has no API.

Summary

An application developer wishing to use an Autodesk Inventor API to display Autodesk Inventor documents in their application has two options. The Apprentice Server API provides the ClientView object, enabling Autodesk Inventor files to be viewed and manipulated in a client window.

If no other Apprentice Server functionality is required, the second option is to use the InventorViewerCtrl control, which can be embedded into applications and web pages without the need for either Autodesk Inventor or Apprentice Server.

A user application for viewing Autodesk Inventor files is installed with Autodesk Inventor. It is named Autodesk Inventor Read-only Mode, and is accessible though the Microsoft Windows Start menu.

Also consider

Autodesk has viewers for many of its products, but an increasingly popular format is DWF (Drawing Web Format). Autodesk Inventor can export DWF files for publishing.

Client Graphics

Introduction to Client Graphics

The Autodesk Inventor API provides the means to place certain graphic primitives on components or in drawing views. These primitives (points, lines, triangles, text) collectively referred to as ClientGraphics, will be maintained and transformed by Autodesk Inventor for the duration of the session only. ClientGraphics are transient in nature, unlike parts and assemblies. AutoCAD users will note a similar intent to the "grdraw" and "grvecs" AutoLISP functions, which draw display-only vectors in an AutoCAD viewport. However, ClientGraphics are not destroyed by a screen refresh.

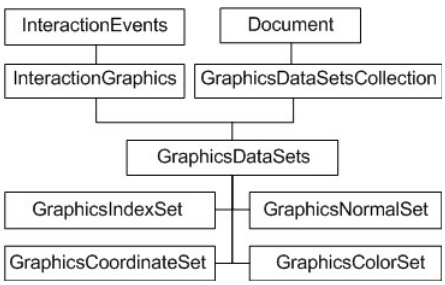
An interesting feature of ClientGraphics is that they will be transformed along with everything else during view changes, zooms, pans, and so on. However, it is possible to specify that they be front facing. So that as the viewpoint changes, the graphics location appears to change too but its planar appearance does not - important for text to remain readable.

The purpose of Client Graphics

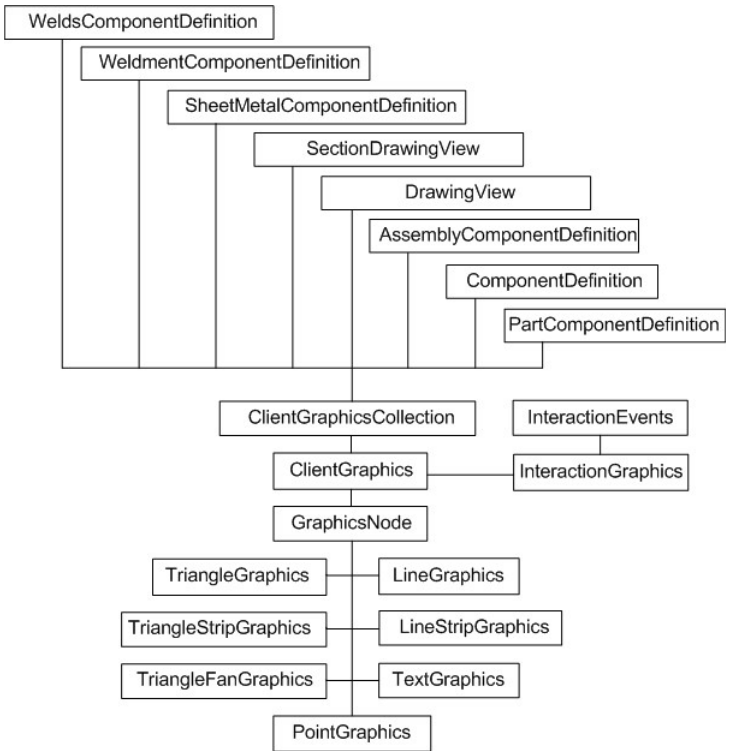
Client Graphics allow the developer to provide visual cues. Autodesk Inventor does this itself in many commands. When the user extrudes a sketch, a preview of the extruded part is displayed, proving a visual cue to how the part would appear (see InteractionGraphics, below). A developer can provide similar visual cues for their own application. A common example is a cutting application, where it is desirable to show the cutting path prior to executing the cut.

Autodesk Inventor recently introduced a new object - InteractionGraphics. This operates in a similar manner to regular ClientGraphics, except that it's in the context of InteractionEvents only. ClientGraphics via InteractionGraphics are much faster and so are well suited to real-time feedback during a command. InteractionGraphics are automatically removed once the associated InteractionEvents object stops.

Custom Graphics Object Model Diagram - Data Sets



Custom Graphics Object Model Diagram - Client Graphics



Working with Custom Graphics via the API

Creating Client Graphics can be considered a two stage process. There are two separate object models, indicated by the preceding diagrams. The first stage is to set up the data - most importantly, the list of coordinates your graphics will be based upon. The second stage is to set up the graphics primitives that will use that data.

This separation of data from graphics allows a single set of data to be referenced by many primitives. In addition, it allows Autodesk Inventor to easily transform graphics by transforming a single set of data only. All graphics referencing that data will be updated automatically. A way is needed then to map a subset of the point data to your required graphics primitives. This is done through the GraphicsIndexSet object. This contains an indexed list of indices to a data set, essentially creating a new list of coordinates to be passed to the graphics set.

Creating a data set

As indicated by the object diagram, the GraphicsDataSetsCollection object is owned by the document. For the purposes of this code sample, the document may be either a part document or an assembly document. The first step is to add a new GraphicsDataSets object, here named 'CG_Test', to the GraphicsDataSetsCollection. From there you can create the GraphicsCoordinateSet object.

```
Dim oDoc As Document
Set oDoc = ThisApplication.ActiveDocument

Dim oDataSets As GraphicsDataSets
Set oDataSets = oDoc.GraphicsDataSetsCollection.Add("CG_Test")

Dim oCoordSet As GraphicsCoordinateSet
Set oCoordSet = oDataSets.CreateCoordinateSet(1)
```

This code and the code that follows does not perform error checking, for the sake of clarity and brevity. Your code should always check that return values are as expected, and that you are not attempting to create objects that already exist.

Now create a one dimensional array, and add your list of coordinates. Here they are added individually for clarity; in reality you would likely read them from file, generate them mathematically, or infer from other geometry. Once all the points have been added to the array, pass the array to the GraphicsCoordinateSet object.

Creating graphics primitives

As indicated by the object model diagram, ClientGraphics can be associated with a number of persistent objects in Autodesk Inventor (ComponentDefinition, DrawingView, and so on). This sample code uses a ComponentDefinition.

```
Dim oCompDef As ComponentDefinition
Set oCompDef = oDoc.ComponentDefinition

Dim oClientGraphics As ClientGraphics
Set oClientGraphics = oCompDef.ClientGraphicsCollection.Add("CG_Test")
```

The preceding code obtains the ComponentDefinition (in this case from the document object). Then a new ClientGraphics object is added, also named 'CG_Test', to its ClientGraphicsCollection.

Autodesk Inventor client graphics uses the concept of nodes. A node is a logical grouping of graphics, typically grouped for the purpose of selection or transformation.

The next step is to create a new node, and determine what primitives to add to it. A number of different graphics primitives are supported. Referring to the object model diagram, you can see that these include lines, strips of lines, triangles, strips of triangles, and so on. Grouping of primitives into strips allows for more efficient use of point data where line endpoints are coincident. However, for demonstration purposes, this example will create a triangle of three lines as LineGraphics, rather than creating a single TriangleGraphics object. The following code creates a new node and adds a LineGraphics object to it.

```
Dim oLineNode As GraphicsNode
Set oLineNode = oClientGraphics.AddNode(1)

Dim oLineSet As LineGraphics
Set oLineSet = oLineNode.AddLineGraphics
```

Now to define the relationship between the point data (the GraphicsCoordinateSet) and the points required as endpoints of lines. To do this, create a GraphicsIndexSet object and add the indices of each X Y Z point required.

Note: The points are added to the GraphicsIndexSet object as indices to the original data set, not as coordinates. In this example, the GraphicsCoordinateSet data set only contains three points, but to create a LineSet of three lines requires four points. To create a triangle, the last point is the same as the starting point.

To define a triangle, the GraphicsIndexSet must reference the first coordinate in the data set twice, so the GraphicsIndexSet will contain four points. The following code adds all four points to the GraphicsIndexSet, as indices to the coordinate points in the GraphicsCoordinateSet object.

```
Dim oIndex As GraphicsIndexSet
Set oIndex = oDataSets.CreateIndexSet(1)

Call oIndex.Add(1, 1)  ' Line from point 1
Call oIndex.Add(2, 2)  ' to point 2
Call oIndex.Add(3, 3)  ' to point 3

Call oIndex.Add(4, 1)  ' to point 1
```

When creating graphics, typically some control over color is required. The object model diagram refers to the GraphicsColorSet object. Use this to assign a color to a group of graphics primitives.

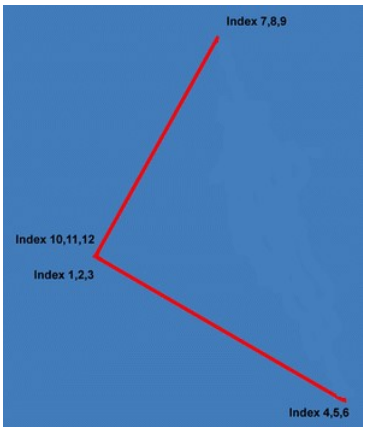
The GraphicsColorSet is an indexed list of color definitions. Add a color definition to the GraphicsColorSet object, and pass the object to the LineSet object in order to determine the color of the resulting graphics. The example below assigns the RGB values for the color red (255,0,0)

```
Dim oColorSet As GraphicsColorSet
Set oColorSet = oDataSets.CreateColorSet(1)
Call oColorSet.Add(1, 255, 0, 0)
oLineSet.ColorSet = oColorSet
```

Now all that remains is to apply the coordinates, or more specifically the GraphicsIndexSet indices, to the LineSet object, and then update the active view.

```
oLineSet.CoordinateSet = oCoordSet
oLineSet.CoordinateIndexSet = oIndex
ThisApplication.ActiveView.Update
```

This will result in something like the red triangle in the following figure. This triangle will be transformed along with any other parts or assemblies when the Autodesk Inventor rotate tool is invoked. To erase the graphics, call the delete method of the 'CG_Test' ClientGraphics object.



Advanced uses for Client Graphics - InteractionGraphics

Autodesk Inventor recently introduced the InteractionGraphics object. This enables association of client graphics with user interaction. A typical use would be to provide some visual feedback during a custom command.

Interaction client graphics are set up in much the same way as previously, except now the GraphicsDataSets object is obtained from the InteractionGraphics object, which is obtained from the InteractionEvents object. As the client graphics are happening in the context of an InteractionEvents object, they are at liberty to take advantage of mouse movement information, selection information, and so on - any events supported by the InteractionEvents objects.

The developer is still responsible for creating and transforming the client graphics, but now can do this based on user input. InteractionGraphics are not transacted. This greatly improves performance, and leaves the developer free to commit an operation once the user responds to appropriate visual cues.

InteractionGraphics can be of type preview, which equate to regular ClientGraphics, or they can be of type overlay. Overlay InteractionGraphics can be rendered to a single view, and can be updated independently through use of the UpdateOverlayGraphics method.

The following sample code demonstrates client graphics transformed by mouse movement. Create a blank form in VBA, with no controls, and add this code to the form.

```
Private WithEvents oInteraction As InteractionEvents
Private WithEvents oMouseEvents As MouseEvents
Private oPointCoords(1 To 6) As Double
Private oCoordSet As GraphicsCoordinateSet
Private oLineNode As GraphicsNode
Private oLineSet As LineGraphics
Private oClientGraphics As ClientGraphics
Private oIG As InteractionGraphics
Private oDataSets As GraphicsDataSets

Sub UserForm_Initialize()
Set oInteraction = ThisApplication. _
    CommandManager.CreateInteractionEvents
oInteraction.SelectionActive = False
Set oMouseEvents = oInteraction.MouseEvents
oMouseEvents.MouseMoveEnabled = True
Set oIG = oInteraction.InteractionGraphics
Set oDataSets = oIG.GraphicsDataSets
Set oCoordSet = oDataSets.CreateCoordinateSet(1)
oPointCoords(1) = 0
oPointCoords(2) = 0
oPointCoords(3) = 0
oPointCoords(4) = 6
oPointCoords(5) = 0
oPointCoords(6) = 0
Call oCoordSet.PutCoordinates(oPointCoords)
Set oClientGraphics = oIG.OverlayClientGraphics
Set oLineNode = oClientGraphics.AddNode(1)
Set oLineSet = oLineNode.AddLineGraphics
oLineSet.CoordinateSet = oCoordSet
oIG.UpdateOverlayGraphics ThisApplication.ActiveView

oInteraction.Start

End Sub

Private Sub oMouseEvents_OnMouseMove(ByVal Button As MouseButtonEnum, _
   ByVal ShiftKeys As ShiftStateEnum, ByVal ModelPosition As Point, _
   ByVal ViewPosition As Point2d, ByVal view As view)
oPointCoords(4) = ModelPosition.X
oPointCoords(5) = ModelPosition.Y
oPointCoords(6) = ModelPosition.Z
Call oCoordSet.PutCoordinates(oPointCoords)
oIG.UpdateOverlayGraphics ThisApplication.ActiveView
End Sub

Private Sub UserForm_Terminate()
oInteraction.Stop
Set oMouseEvents = Nothing
Set oInteraction = Nothing
End Sub
```

Now add the following code to the Modules section of the project. Running this MouseIG macro will cause a custom graphics line to follow the mouse cursor in Autodesk Inventor. One end of the line will be at 0,0,0, the other end will be governed by the coordinates returned by the mouse movement event.

```
Public Sub MouseIG()
    frmMouseTest.Show vbModeless
End Sub
```

The above sample demonstrates the similarities between interaction graphics and regular client graphics. However, graphics implemented through interaction graphics are not transacted, and so are not subject to the performance overhead imposed by the transaction and undo mechanisms. Interaction graphics are designed for interactive, real-time visual feedback.

Summary

Client graphics are not a substitute for sketches or drawings - they are intended for use as transient visual cues in the modeling environment. A limited set of graphics primitives are provided. Circles, arcs and so on are not supported and should be approximated with short line segment constructs. Efficiencies are provided in that many primitives can use subsets of a common pool of point data - hence the separation of data and graphic objects in the API. Graphics will appear to be transformed by standard Autodesk Inventor pan and zoom operations. Client graphics can be associated with input events, providing for immediate visual feedback of user interaction.

Also consider

When client graphics are based at least in part on an existing model, it is possible to obtain a coordinate data set from the existing surface body. Call the SurfaceBody.CalculateStrokes method to get the list of coordinate points. Determine the tolerance (the number of points you need) by first calling SurfaceBody.GetExistingStrokeTolerances. You may need to do this multiple times in order to assess the average tolerance for the model. Similarly, obtain facets by calling SurfaceBody.CalculateFacets.

DataIO

Introduction to DataIO

Autodesk Inventor users can save their work or import from files in a variety of formats. For example, a part file can be saved as a SAT file, a STEP file, and so on. Autodesk Inventor can open files such as DWG, DXF, IGES, STEP, and so on. The type of format available is dependent on the context.

The purpose of DataIO objects

To provide a similar level of functionality to the developer, many objects in the Autodesk Inventor API provide access to a DataIO object through a read-only DataIO property. The DataIO object provides methods to read from and write to files and data streams. In addition, the DataIO object provides methods to determine what formats and storage types are valid in that context. For example, a sketch can be saved as a DXF file, but cannot be imported from a bitmap.

The following is a list of Autodesk Inventor API objects that can return a DataIO object.

- AssemblyComponentDefinition
- AttributeSet
- AttributeSets
- ComponentDefinition
- DrawingSketch
- DrawingView
- PartComponentDefinition
- PlanarSketch
- PlanarSketchProxy
- SectionDrawingView
- Sheet
- SheetMetalComponentDefinition
- Sketch
- SurfaceBody
- SurfaceBodyProxy
- WeldmentComponentDefinition
- WeldsComponentDefinition

Working with DataIO objects

As DataIO objects are available from such a variety of Autodesk Inventor API objects, so the context of the data to be read or written is variable. It is important to establish, in that context, whether data can be written or read, in what format, and whether to a file or data stream. Never just assume that a given format is valid.

The DataIO object supports two methods which determine available formats: *GetInputFormats* and *GetOutputFormats*. These methods each return a pair of arrays, identically dimensioned. The first array, Formats, contains the list of file formats in string form. For example, "XML" or "ACIS SAT." The second array contains *StorageTypeEnum* values. For example, *kFileStorage* indicates that storing as a file is supported.

Once the supported formats and storage types are known, the appropriate DataIO method can be used, with the format string passed to identify the required format. The output (write) methods are *WriteDataToFile* and *WriteDataToStream*. The input (read) methods are *ReadDataFromFile* and *ReadDataFromStream*. A file storage type is most typically used, but these methods also support streams. (For details on streams, see the Microsoft implementation of the IStream interface for structured storage).

Example Format and StorageType arrays

The following tables depict DataIO object format and storage type array values that might be returned for a component definition in a part document. It shows Input (read) formats and Storage Types, returned by *GetInputFormats*.

Format Array	StorageType Array
ACIS SAT	kFileStorage

ACIS SAB	kFileOrStreamStorage

The following example shows output (write) Formats and Storage Types, returned by *GetOutputFormats*, for the same object.

Format Array	StorageType Array
ACIS SAT	kFileStorage
ACIS SAB	kFileOrStreamStorage
ACIS SAT with TransientKeys	kFileStorage
ACIS SAB with TransientKeys	kFileOrStreamStorage
ACIS SAT with ProceduralToNURBS	kFileStorage
ACIS SAB with ProceduralToNURBS	kFileStorage
ACIS SAT with ProceduralToNURBS with TransientKeys	kFileStorage
ACIS SAB with ProceduralToNURBS with TransientKeys	kFileStorage
ACIS SAT with ProceduralToNURBS with TransientKeys DocUnits	kFileStorage
ACIS SAB with ProceduralToNURBS with TransientKeys DocUnits	kFileStorage

These example arrays indicate that "ACIS SAT" is supported for read and write, but file storage only, while "ACIS SAB with TransientKeys" is supported for write-only, but with file or stream storage.

How to use the API to determine available formats and storage types

This sample assumes an open part document containing a part, and looks at the supported DataIO formats for two objects: the *ComponentDefinition* and that object's *AttributeSets* object. A sequence of message boxes are posted containing the valid formats and storage types.

This code omits error checking for the sake of clarity and brevity. Always check that return values are of the expected type.

First, define a subroutine, and then obtain the active part document and its component definition object.

```
Sub query_dataIO()

Dim oDoc As PartDocument
Set oDoc = ThisApplication.ActiveDocument

Dim oCompDef As ComponentDefinition
Set oCompDef = oDoc.ComponentDefinition
```

Declare the variables for the Format and StorageType arrays, and then get the DataIO object from the component definition.

```
Dim sFormats() As String
Dim sStorageTypes() As StorageTypeEnum
Dim oDataIO As DataIO
Set oDataIO = oCompDef.DataIO
```

Get the supported input (read) formats and display the results in a message box. However, first compose the message string - done by the ioPrint function, defined after this sub.

```
oDataIO.GetInputFormats sFormats, sStorageTypes
MsgBox ("CompDef Input: " & ioPrint(sFormats, sStorageTypes))
```

Get the supported output (write) formats and display the results in a message box, but first compose the message string - done by the ioPrint function, defined after this sub.

```
oDataIO.GetOutputFormats sFormats, sStorageTypes
MsgBox ("CompDef Output: " & ioPrint(sFormats, sStorageTypes))

End Sub
```

That completes the query_dataIO subroutine. Now, define the ioPrint function. This is a utility function that takes the two arrays and pairs each format and storage type, returning a human-readable string. This function demonstrates that the format and storage type arrays always have identical dimensions. The *nth* item in the format array relates to the *nth* item in the storage type array.

```
Function ioPrint(sFormats As Variant, sStorageTypes As Variant) _
   As String
Dim msgString As String
Dim lindex As Long
For lindex = 0 To UBound(sFormats)
  msgString = msgString & sFormats(lindex)
  Dim lType As StorageTypeEnum
  lType = sStorageTypes(lindex)
  Dim sType As String
```

```
  Select Case lType
    Case kFileOrStreamStorage
      msgString = msgString & "(File or Stream) "
    Case kFileStorage
      msgString = msgString & "(File) "
    Case kStorageStorage
      msgString = msgString & "(Storage) "
    Case kStreamStorage
      msgString = msgString & "(Stream) "
    Case kUnknownStorage
      msgString = msgString & "(Unknown) "
  End Select
Next
ioPrint = msgString
End Function
```

Running the query_dataIO sub generates two message dialog boxes, containing something like the following examples.

CompDef Input: ACIS SAT (File) ACIS SAB (File or Stream)

CompDef Output: ACIS SAT (File) ACIS SAB (File or Stream)
ACIS SAT with TransientKeys (File) ACIS SAB with TransientKeys (File or Stream)

Using the format and storage type information

This component definition supports writing to a SAT file. So it is valid to call *WriteDataToFile*, using the string "ACIS SAT" as the format, supplying a file name for storage, as in the following example.

```
oDataIO.WriteDataToFile "ACIS SAT", "c:\MyNewSATfile.SAT"
```

Note: Some supported file formats such as XML can have tags meaningful only in the given context, so it can be very helpful to use *WriteDataToFile* to produce a sample file to see that file's structure.

In addition to using the DataIO object, files of many formats can be saved or opened through the *Open* or *SaveAs* methods of the *Document* object. The extension of the file name indicates to Autodesk Inventor which format is required.

Summary

The Autodesk Inventor API supports reading and writing files in a variety of formats, depending on the context (object type). Many objects in the API provide access to a DataIO object. This object provides methods to determine what formats and storage types (file or stream) are valid in that context. The DataIO object supports methods to read and write to and from such valid formats.

Also consider

Autodesk Inventor supports a type of Addin called a Translator Addin. This is much like a regular Addin, but is an Addin that reads and/or writes data in a specific, possibly proprietary, format. The Addin may do all the work or it may use an existing DataIO object. Such Addins are classed differently because they hook into Autodesk Inventor's File > Load and File > Save commands. For more information, refer to the TranslatorAddin object reference.

Zero Impact Migration

Zero Impact Migration

The idea behind Zero Impact Migration (ZIM) is to eliminate the performance and memory cost in opening Inventor files saved with earlier versions of the software. The Document Interests mechanism has been added to Inventor to aid ZIM.

Note: Document Interests are the recommended means of marking schema versions for application custom data in a document, and in fact is now the preferred method for an Add-in to implement a document sub-type. However, it is not recommended that applications rely on DocumentInterests when querying a document type. An application should not draw any meaning from DocumentInterests that it did not add. In other words, application A should not infer anything from application B's DocumentInterests.

The following table lists APIs available to implement Zero Impact Migration.

ZIM related APIs
Document.DocumentInterests property
Document.NeedsMigrating property
ApprenticeServerDocument.DocumentInterests property
DocumentInterests collection object
DocumentInterest object
ApplicationAddin.DataVersion property

ApplicationEvent.OnMigrateDocument event

Requirements

Here are the set of tasks that must be performed by an application to participate in ZIM:

1. During registration, create a DWORD key called DataVersion under the same key (viz. `'HKEY_CLASSES_ROOT\CLSID\{CLSID}\Settings'`) where the other keys such as LoadOnStartUp are currently being created for the Add-in application. The value of the DataVersion key is the data or schema version for the Add-in application's custom data.
2. In all code paths where the Add-in application creates any custom data for the first time and hence intends to mark the document as one containing its data, check whether the Add-in's Document Interest exists and that its DataVersion property value is correct.

   If a Document Interest does not already exist, register a new DocumentInterest on the document by using the DocumentInterests.Add method. Make sure that the DataVersion value is correct and matches the one in the add-in application's registry key set in step 1 and also available through the ApplicationAddin.DataVersion property.

   The 'Name' argument supplied during creation of the Document Interest can be used to define an application specific document sub-type e.g. "Application Top Level Assembly document".
3. Move any existing migration code to the OnMigrateDocument event handler.
4. Create a ChangeProcessor with ChangeType as kSchemaChangeCmdType to wrap the migration code in the OnMigrateDocument event handler.
5. In the OnMigrateDocument event handler, after the migration code is executed, update the value of the DataVersion property of the add-in application's Document Interest to match the one in the add-in application's registry key set in step 1 and also available through ApplicationAddin.DataVersion property.

Migration Event

The migration event is fired when the 'File->Migrate' menu item is clicked or when an edit operation in requested on a non-migrated document.

Whether an application's data needs migration is decided by Inventor by comparing the DataVersion stored in the DocumentInterest registered by the application and the DataVersion set in the registry by the application during its registration. If the DataVersion in the registry is greater than that in the DocumentInterest in the document, a migration event will be fired that the application can react to and migrate its data.

User-Defined Functions in Parameter Expressions

Introduction

In Autodesk Inventor's Parameters command, you can specify an expression for a parameter. Expressions can be a specific value or an equation that references other parameters and uses functions. Earlier releases of Inventor limited the use of functions in expressions to a small set of built-in functions such as sine, cosine, minimum and maximum. Now, you can write your own functions using VBA in parametric expressions. These user-defined functions can be used in the expressions just like any of the built-in functions.

While you can make your functions powerful using all the capabilities of VBA, you must be aware of the requirements and rules specific to these functions, which are described in the rest of this section.

Writing User-Defined Functions

You must write the functions in the code module called "Functions." This code module is automatically created by Inventor for all document projects. It will initially be empty. Any parameters in the document will be able to use functions contained within this module. The parameters in a document cannot, however, use functions created in a documented project associated with another document. Because of the parameter's dependency on the function used in an expression, the function must exist in the document project in the document containing the parameter. Also, the function must be a public function. You can use all capabilities of VBA and the VBA development environment while writing these functions.

The user-defined function must take at least one argument, but it can have any number of additional arguments. All arguments are treated as input and their data type as Double. The return type of the function must be Double.

The Parameter command converts the return value of the user-defined function to a unit-less number.

The function below demonstrates a correctly defined function.

```
Public Function Sample(Arg1 As Double, Arg2 As Double) As Double
    'Use the input arguments to compute a value.
    If Arg1 < 5 Then
        Sample = Arg2
    ElseIf Arg1 < 10 Then
        Sample = Arg2 * 2
    Else
        Sample = Arg2 * 3
    End If
End Function
```

This function takes two arguments as input, uses these values to compute a new value, and returns it as the result.

Handling Units in the Function

One thing to pay particular attention to when writing VBA functions that will be used in parameter equations is the units. All of the Inventor API deals exclusively with internal database units. For example, if you use the API to obtain the length of an edge, the result is returned in centimeters. Centimeters are the internal database units for length. Internally all lengths are computed and saved as centimeters. Using the Document Settings command, the user can choose which units they want to use for that document. For example, let's say that inches are chosen as the length unit. When the user enters a length in a dialog, Autodesk Inventor will assume it is inches and when displaying results it will be in inches. In both cases a conversion is being done to and from centimeters during the reading of the input and the display of the output. The advantage of this behavior is that the units the user chooses for a document can be changed at any time without any impact on the data in the document. It also allows the mixture of parts using different units within assemblies. For a more complete discussion of units, see Units of Measure.

The VBA functions for parameter equations need to take this behavior into consideration because all values passed into the function will be in database units. This may make writing functions a little more confusing until you understand the way units are used in Autodesk Inventor, but soon you will discover the advantage of them working correctly regardless of the document unit settings. Let's look at the previous sample to better understand this behavior and how to correctly write a function.

This function has two input arguments. If we look at the use of this function in the Parameters dialog, you can see that two parameters are passed in as input values for these arguments. These two parameters happen to define length units and in this case have the values 1 in and 2 in. When Inventor calls the VBA function these values will be converted to length database units (centimeters). So the values the function receives will be 2.54 and 5.08. If one of the input arguments happened to be the parameter d3, the value received by the function would be 0.03490658, which is 2 deg converted to radians. Radians are the internal database unit for angles.

Understanding that these values are input as centimeters is important in this particular function because it is comparing these values in some logic statements. The If statement checks to see if Arg1 is less than 5. It would be a common mistake to assume you are checking to see if Arg1 is less than 5 inches. Instead it is checking to see if Arg1 is less than 5 centimeters. If the comparison needed to be for 5 inches the If statement could use 5 * 2.54 instead. The important thing to remember is that within the VBA function, you should always work in the world of Autodesk Inventor's database units: centimeters and radians.

Debugging

Debugging user-defined functions can easily be performed by placing break points within the function. If you set a break point, performing an operation within Autodesk Inventor that causes the function to be called will place you in debugging mode within the VBA environment.

In addition to setting break points, if for some reason the function should fail, you will be allowed to enter debug mode at that point and correct the function and continue processing.

Limitations

One limitation that has been deliberately imposed on user-defined functions in parameter expressions is that they cannot use any of the Autodesk Inventor API within the function. For example, you cannot have a function that will directly edit another parameter or change the suppressed state of a feature. Any calls to the Autodesk Inventor API will fail within the context of the function.

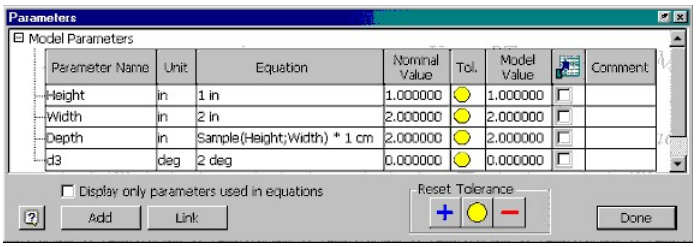
The reason for this limitation is that the function is called during the compute process of Autodesk Inventor. Calls to the API while Autodesk Inventor is in this state can cause problems and the results can be unreliable. For example, let's say you have a part made up from 10 features. When this part is entirely recomputed each feature is computed in sequence. Based on dependencies, Autodesk Inventor determines when to compute the value of a parameter. Let's say in this case the parameter that uses the user-defined function is computed after the fifth feature is computed. The model is only partially computed and is not in an editable state. Even query operations of the model will be unreliable in this state since they would only provide a snapshot at the point of what the model looks like, not the final result. And since you don't know when the parameter will be computed, you don't know what the point of that snapshot would be.

Although you cannot use the Autodesk Inventor API within the function, there are no other limitations. For example, the function could perform queries and obtain values from an Access database, or look up information from Excel.

Using User-Defined Functions

The VBA function is used the same way as the built-in functions are used in Autodesk Inventor. One requirement that is easily overlooked is that the separator between arguments is a semi-colon, not a comma.

The figure below illustrates using the user-defined function in the Parameters dialog.



You can also use the user-defined functions with the Edit Dimensions command. The following figure shows the Edit Dimension dialog box using a function:

While using the function in Parameters and Edit Dimensions dialog boxes, you need not enter "VBA:" before the function. Autodesk Inventor automatically prefixes the function with "VBA:".

Translator Options

The tables below list the valid name-value pairs that can be used to define the various options when translating files using the translator add-ins. The serve as the API equivalent to the Options dialog when translating a file interactively.

Questions about what most of the options do can be answered by reading the Inventor online help for the thranslator and by experimenting interactively with the various options in the Options dialog of the "Save Copy As" or "Open" dialogs.

Import options for Alias, CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, STL, SAT, STEP, SolidWorks, Fusion and OBJ.

Option Name	Value Type	Default Value	Supported Translators
SaveComponentDuringLoad	Boolean	False	Alias, CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, STL, STEP, SolidWorks, OBJ
SaveLocationIndex	Integer	0 (Save in workspace)	Alias, CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, STL, SAT, STEP, SolidWorks , OBJ
ComponentDestFolder	String		Alias, CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, STL, SAT, STEP, SolidWorks , OBJ
SaveAssemSeperateFolder	Boolean	False	Alias, CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, SAT, STEP, SolidWorks
AssemDestFolder	String		Alias, CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, SAT, STEP, SolidWorks, Fusion
EmbedInDocument	Boolean	True	Alias, CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, STL, STEP, SolidWorks , OBJ
SaveToDisk	Boolean	False	Alias, CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, STL, STEP, SolidWorks , OBJ
ImportSolid	Boolean	True	Alias, CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, STEP, SolidWorks, Fusion
ImportSurface	Boolean	True	Alias, CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, STEP, SolidWorks, Fusion
ImportWire	Boolean	True	Alias, CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, STEP, SolidWorks, Fusion
ImportWorkPlane	Boolean	True	JT, NX, Pro/ENGINEER
ImportWorkAxe	Boolean	True	JT, NX, Pro/ENGINEER
ImportWorkPoint	Boolean	True	JT, NX, Pro/ENGINEER
ImportPoint	Boolean	True	Alias, CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Rhino, STEP
ImportMeshes	Boolean	True	CATIA V4, CATIA V5, Alias, NX, STEP, Rhino, Fusion
ImportMeshes	Boolean	False	JT
ImportGraphicalPMI	Boolean	True	JT
CreateIFO	Boolean	False	Alias, CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, STEP, SolidWorks, Fusion
ImportAASP	Boolean	False	CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, SAT, STEP, SolidWorks, Fusion
ImportAASPIndex	Integer	0 (Part with multiple solids)	CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, SAT, STEP, SolidWorks, Fusion
CreateSurfIndex	Integer	1 (Single composite)	CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, SAT, STEP, SolidWorks, Fusion
GroupNameIndex	Integer	0 (Default)	CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, SAT, STEP, SolidWorks, Fusion
GroupName	Integer	0	CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, SAT, STEP, SolidWorks, Fusion
ExplodeMSB2Assm	Boolean	True	SAT
CHKSearchFolder	Boolean	False	NX
SearchFolder	String Array		NX
AssociativeImport	Boolean	True	Alias, Catia, NX, Pro/ENGINEER, STEP, SolidWorks, Fusion
DefaultNames	Boolean	True	SAT
CEGroupLevel	Integer	0 (Level)	IGES, Rhino
CEPrefixCk	Boolean	False	Alias, IGES, Rhino, STEP
ImportUnit	Integer	0 (Source unit) STL, OBJ: 0 (Template units)	CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, SAT, STEP, SolidWorks, STL, OBJ
CheckDuringLoad	Boolean	False	CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, SAT, STEP, SolidWorks
AutoStitchAndPromote	Boolean	True	CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, SAT, STEP, SolidWorks
AdvanceHealing	Boolean	False	CATIA V4, CATIA V5, IGES, JT, NX, Parasolid, Pro/ENGINEER, Rhino, SAT, STEP, SolidWorks
NoShowExpModelList (Input)	Boolean	True	CATIA V4 (For *.exp and *.div3 import)
ExpModelIndex (Input)	Integer	N/A	CATIA V4 (For *.exp and *.div3 import)
ExpModelCount (Output)	Integer	N/A	CATIA V4 (For *.exp and *.div3 import)
ExpModelNameList (Output)	String Array	N/A	CATIA V4 (For *.exp and *.div3 import)
ImportColor	Boolean	True	STL
ImportColorIndex	Integer	0	STL
TessellationDetailIndex	Integer	0	JT
SplitGroup	Boolean	True	OBJ

Import options for DWG import when importing into a sketch.

For other DWG import cases, an .ini file is used to define the options. You can manually use the dwg translator to set the various options you want and then write out an .ini file with the settings that you can then specify later when translating via the API.

Option Name	Value Type	Default Value	Notes
ImportModelSpace	Boolean	True	Specifies whether to import model space or not, set this to False to import the layout paper space where you specify the layout name using the SelectedLayout setting.
SelectedLayout	String		Specifies the layout name to import when the ImportModelSpace setting is set to False.
FileUnits	String		Specifies the units for import. The units can be “Meters”, ”Feet”, “Microns”, “Inches”, “Centimeters”, or “Millimeters”. If not specified it will use the units from the document.
SelectedLayers	String		Specifies the layer names to import. The names are comma delimited.
InvertLayersSelection	Boolean	True	Specifies whether to invert the layers selection during import. Leaving the SelectedLayers empty and setting this value to True will import all the layers.
ConstrainEndPoints	Boolean	False	Specifies if end points constraints should be applied to the imported geometry.
ApplyGeometricConstraints	Boolean	False	Specifies if geometric constraints should be applied to the imported geometry to fully constrain the sketch. This is ignored if the ConstrainEndPoints setting is set to False.
ImportParametricConstraints	Boolean	False	Specifies if AutoCAD 2D parametric constraints should be imported.
ProxyObjectsToUserDefinedSymbols	Boolean	False	Specifies if proxy objects should be converted to user-defined symbols. This is only valid in Inventor drawings.

Definitions of Import Values

Option Name	Valid Values
SaveLocationIndex	SAVE_IN_WORKSPACE = 0 SELECT_SAVE_LOCATIONS = 1 SAVE_AT_IMPORT_LOCATION = 2
CreateIFO	1. When CreateIFO is true, CreateSurfIndex only supports SINGLE_COMPOSITE or MULTIPLE_COMPOSITE; if CreateSurfIndex’s value is illegal, it will be set as SINGLE_COMPOSITE. 2. When CreateIFO is true, AutoStitchAndPromote can be set as true or false. 3. When CreateIFO is true, AdvanceHealing will be set as false.
ImportAASPIndex	MULTIPLE_SOLID_PART = 0 SINGLE_COMPOSITE_FEATURE = 1
CreateSurfIndex	INDIVIDUAL_SURFACE = 0 (Translates all surfaces to individual ones, and consumes much more memory and time.) SINGLE_COMPOSITE = 1 MULTI_COMPOSITE = 2 SINGLE_CONSTRUCTION = 3 MULTI_CONSTRUCTION = 4
GroupNameIndex	DEFAULT = 0 ENTERED_NAME = 1
CEGroupLevel	eLevelType = 0 eGroupType = 1
ImportUnit	SOURCE_UNITS = 0 TEMPLATE_UNITS = 1 INCH = 2 FOOT = 3 CENTIMETER = 4 MILLIMETER = 5 METER = 6 MICRON = 7
NoShowExpModelList	When set to true, a dialog that lists all models in the *.exp or *.dlv3 file is shown and ask the user select a file to open.
ExpModelIndex	Used for user’s input to open one of the models in the *.exp or *.dlv3 file.
ExpModelCount	User can get the model count in the *.exp or *.dlv3 file.
ExpModelNameList	User can get a list of all model"s name in the *.exp or *.dlv3 file.
ImportColorIndex	RGB = 0 BGR = 1
TessellationDetailIndex	eHighestDetail = 0 eLowestDetail = 1

Export options for CATIA V5, IGES, JT, Parasolid, Pro/ENGINEER, SAT, STEP, STL and OBJ.

Option Name	Value Type	Default Value	Supported Translators
Work Plane Export	Boolean	TRue	JT, Pro/ENGINEER
Work Axes Export	Boolean	True	JT
Work Point Export	Boolean	True	JT, Pro/ENGINEER
IncludeSketches	Boolean	True	CATIA V5, IGES, JT, Parasolid, Pro/ENGINEER, SAT, STEP
Version	Integer	CATIA: 28 JT: 102 Pro/ENGINEER Granite: 11 Parasolid: 31 SAT: 7	CATIA V5, JT, Parasolid, Pro/ENGINEER, SAT
ConfigFileEnabled	Boolean	False	JT
ConfigFilePath	String		JT
OutputFileType	Integer	0 (BrepAndFacets)	JT
XTBrep	Boolean	True	JT
ExportPMI	Boolean	True	JT
FileStructure	Integer	0 (Monolithic)	JT
ApplicationProtocolType	Integer	4 (eAP214IS)	STEP

Author	String		STEP
Organization	String		STEP
Authorization	String		STEP
Description	String		STEP
GeometryType	Integer	1 (Solids)	IGES
SurfaceType	Integer	0 (143-Bounded)	IGES
SolidFaceType	Integer	0 (NURBS)	IGES
ExportBodyNames	Boolean	False	SAT
OutputFileType	Integer	0 (Binary)	STL
ExportUnits	Integer	STL: 4 (Centimeters) OBJ: 1 (Source unit)	STL, OBJ
ExportFileStructure	Integer	0 (One File)	STL , OBJ
Resolution	Integer	1 (Medium)	STL , OBJ
SurfaceDeviation	Integer	60	STL , OBJ
NormalDeviation	Integer	14	STL , OBJ
MaxEdgeLength	Integer	100	STL , OBJ
AspectRatio	Integer	21.5	STL , OBJ
AllowMoveMeshNode	Boolean	False	STL
export_fit_tolerance	Double	0.001	IGES/STEP
ExportColor	Boolean	True	STL

Definitions of Export Values

Option Name	Valid Values
Version	CATIA V5 (10 - 28) Pro/Engineer Granite (1 - 11) JT (80, 81, 82, 90, 91, 92, 93, 94, 95, 100, 101, 102) e.g. 102 means 10.2 version file. PARASOLID (9 - 31) SAT (7)
OutputFileType	eBrepAndFacets = 0 eBrepOnly = 1 eFacetsOnly = 2
XTBrep	XTBrep = True JTBrep = False
FileStructure	JtkMONOLITHIC = 0 JtkPER_PART = 1 JtkFULL_SHATTER =2
ApplicationProtocolType	eAP203 = 2 eAP214IS = 4 eAP242 = 5
GeometryType	SURFACES = 0 SOLIDS = 1 WIREFRAMES = 2
SurfaceType	143_Bounded = 0 144_Trimmed = 1
SolidFaceType	NURBS = 0 ANALYTICS = 1
OutputFileType	BINARY = 0 ASCII = 1
ExportUnits	INCH = 2 FOOT = 3 CENTIMETER = 4 MILLIMETER = 5 METER = 6 MICRON = 7
ExportFileStructure	ONE FILE = 0 ONE FILE PER PART INSTANCE = 1
Resolution	HIGH = 0 MEDIUM = 1 LOW = 2 CUSTOM = 3 BREP = 4
SurfaceDeviation	Range 0 to 100, the value has precision of 0.0001. None-zero value is used if Resolution is CUSTOM (3), otherwise value is ignored.
NormalDeviation	Range 0 to 41. Non-zero value is used if Resolution is CUSTOM (3), otherwise value is ignored. Any input value out of this range will be changed to 14 by force.
MaxEdgeLength	Range 0 to 100. Non-zero value is used if Resolution is CUSTOM (3), otherwise value is ignored.
AspectRatio	Range 0 to 21.5. Non-zero value is used if Resolution is CUSTOM (3), otherwise value is ignored.
export_fit_tolerance	Set the tolerance for the IGES/STEP file. The accepted range for the tolerance value is from 0.00001 (cm) to 0.001 (cm). You can change this value, using cm units only. A smaller tolerance creates more accurate geometry approximations and larger file size.
ExportColor	Write color information to STL binary file.

Export options for DWF.

Option Name	Value Type	Default Value	Supported Documents	Supported Translators	Notes
Launch_Viewer	Long	0	Part, Assembly, Drawing,Presentation	DWF	
Publish_Mode	DWFPublishModeEnum	kBasicDWFPublish		DWF	

			Part, Assembly, Drawing,Presentation		
Publish_Component_Props	Boolean	False	Part, Assembly, Drawing,Presentation	DWF	Use this to publish ALL component properties
Publish_Mass_Props	Boolean	False	Part, Assembly, Drawing	DWF	Use this to publish ALL mass properties
Publish_All_Component_Props	Boolean	False	Part, Assembly, Drawing,Presentation	DWF	Same as Publish_Component_Props
Publish_All_Physical_Props	Boolean	False	Part, Assembly, Drawing	DWF	Same as Publish_Mass_Props
Enable_Measure	Boolean	True	Part, Assembly, Drawing,Presentation	DWF	DRM protection
Enable_Printing	Boolean	True	Part, Assembly, Drawing,Presentation	DWF	DRM protection
Enable_Markups	Boolean	True	Part, Assembly, Drawing,Presentation	DWF	DRM protection
Enable_Markup_Edits	Boolean	True	Part, Assembly, Drawing,Presentation	DWF	DRM protection
Password	String	""	Part, Assembly, Drawing,Presentation	DWF	
Password_Protect	Boolean	False	Part, Assembly, Drawing,Presentation	DWF	
Password_Type	string or enum		Part, Assembly, Drawing,Presentation	DWF	
Enable_Large_Assembly_Mode	Boolean	True	Assembly	DWF	Enables large assembly optimizations (transactions and out of process publishing)
Output_Path	String		Part, Assembly, Drawing,Presentation	DWF	
Publish_3D_Models	Boolean	False	Drawing	DWF	Specify the publish mode.
Publish_All_Sheets	Boolean	False	Drawing	DWF	
Include_Sheet_Tables	Boolean		Drawing	DWF	
Sheet_Count	Integer		Drawing	DWF	
Sheets	NameValueMap		Drawing	DWF	
Override_Sheet_Color	Boolean	False	Drawing	DWF	Set whether to override the sheet background color.
Sheet_Color	Integer		Drawing	DWF	Specify the sheet background color using an Integer. The Integer is calculated with R,G,B values(0 to 255) using below equation: Sheet_Color = R+G*256+B*65536.
Include_Sheet_Tables	Boolean	True	Drawing	DWF	Set whether to include sheet tables or not.
BOM_Structured	Boolean	False	Assembly;Presentation	DWF	Set whether to publish the Structured BOM.
BOM_Parts_Only	Boolean	False	Assembly;Presentation	DWF	Set whether to publish the Parts Only BOM.
Design_Views	NameValueMap		Assembly;Presentation	DWF	Specify the design views which will be published.
Positional_Representations	NameValueMap		Assembly;Presentation	DWF	Specify the positional representations which will be published.
Sheet_Metal_Style_Information	Boolean	False	Sheetmetal	DWF	Specify whether to publish the sheet metal style infomation.
Sheet_Metal_Flat_Pattern	Boolean	False	Sheetmetal	DWF	Specify whether to publish the flat pattern.
Sheet_Metal_Part	Boolean	True	Sheetmetal	DWF	Specify whether to publish the sheet metal part. At least one of the Sheet_Metal_Flat_Pattern and Sheet_Metal_Part should be specified as True for a sheet metal publish.
Facet_Quality	AccuracyEnum	kHigh		DWF	Specify the facet quality. Only kLow, kMedium and kHigh are valid values for this.
Animations	Boolean	False	Presentation	DWF	Specify whether to publish the animations.
Instructions	Boolea	False	Presentation	DWF	Specify whether to publish the instructions. This will be ignored if the Animations is set to False.
Presentations	NameValueMap		Presentation	DWF	
iAssembly_All_Members	Boolean	False	iAssembly	DWF	Specify whether to publish all the iAssembly members.
iAssembly_3D_Models	Boolean	False	iAssembly	DWF	Specify whether to publish all the 3D models for all the iAssembly members.
iAssemblies	NameValueMap		iAssembly	DWF	Specify which of the iAssembly members will be published.
iPart_All_Members	Boolean	False	iPart	DWF	Specify whether to publish all the iPart members.
iPart_3D_Models	Boolean	False	iPart	DWF	Specify whether to publish all the 3D models for all the iPart members.
iParts	NameValueMap		iPart	DWF	Specify which of the iPart members will be published.
Include_Empty_Properties	Boolean	False	Part, Assembly, Drawing,Presentation	DWF	Specify whether publish the empty properties

Definitions of Export Values

Option Name	Name	Value	Supported Documents	Notes
Sheets	Sheet1	NameValueMap	Drawing	Create a NameValueMap("Sheet1",NameValueMap) to specify options for a sheet, the name should start from Sheet1 and the number increases one by one.
	Sheet2	NameValueMap	Drawing	See above.
	Sheet#	NameValueMap	Drawing	See above.
Sheet1	Name	String	Drawing	The name of a sheet in drawing, this can be any sheet name but not just for Sheet:1.
	3DModel	Boolean	Drawing	Specify whether to publish the 3D model for the sheet.
Sheet2	Name	String	Drawing	The name of a sheet in drawing, this can be any sheet name but not just for Sheet:2.
	3DModel	Boolean	Drawing	Specify whether to publish the 3D model for the sheet.
Sheet#	Name	String	Drawing	The name of a sheet in drawing, this can be any sheet name but not just for Sheet:#.
	3DModel	Boolean	Drawing	Specify whether to publish the 3D model for the sheet.
Design_Views	Design_View1	NameValueMap	Assembly	Create a NameValueMap("Design_View1",NameValueMap) to specify a design view to publish, the name should start from Design_View1 and the number increases one by one.
	Design_View2	NameValueMap	Assembly	See above.
	Design_View#	NameValueMap	Assembly	See above.
Design_View1	Name	String	Assembly	The name of a design view.
Design_View2	Name	String	Assembly	The name of a design view.
Design_View#	Name	String	Assembly	The name of a design view.
Positional_Representations	Positional_Representation1	NameValueMap	Assembly	Create a NameValueMap("Positional_Representation1",NameValueMap) to specify a positional representation to publish, the name should start from Positional_Representation1 and the number increases one by one.
	Positional_Representation2	NameValueMap	Assembly	See above.
	Positional_Representation1#	NameValueMap	Assembly	See above.
Positional_Representation1	Name	String	Assembly	The name of a positional representation.
Positional_Representation2	Name	String	Assembly	The name of a positional representation.
Positional_Representation1#	Name	String	Assembly	The name of a positional representation.
Presentations	Presentation1	NameValueMap	Presentation	Create a NameValueMap("Presentation1",NameValueMap) to specify a presentation view to publish, the name should start from Presentation1 and the number increases one by one.
	Presentation1	NameValueMap	Presentation	See above.
	Presentation#	NameValueMap	Presentation	See above.
Presentation1	Name	String	Presentation	The name of a presentation explosion view.
Presentation2	Name	String	Presentation	The name of a presentation explosion view.
Presentation#	Name	String	Presentation	The name of a presentation explosion view.
iAssemblies	Member1	NameValueMap	iAssembly	Create a NameValueMap("Member1",NameValueMap) to specify options for an iAssembly member, the name should start from Member1 and the number increases one by one.
	Member2	NameValueMap	iAssembly	See above.
	Member#	NameValueMap	iAssembly	See above.
Member1	Name	String	iAssembly	The name of an iAssembly member, this can be any iAssebly member name.
	3DModel	Boolean	iAssembly	Specify whether to publish the 3D model for the iAssembly member.
Member2	Name	String	iAssembly	The name of an iAssembly member, this can be any iAssebly member name.
	3DModel	Boolean	iAssembly	Specify whether to publish the 3D model for the iAssembly member.
Member#	Name	String	iAssembly	The name of an iAssembly member, this can be any iAssebly member name.
	3DModel	Boolean	iAssembly	Specify whether to publish the 3D model for the iAssembly member.
iParts	Member1	NameValueMap	iPart	Create a NameValueMap("Member1",NameValueMap) to specify options for an iPart member, the name should start from Member1 and the number increases one by one.
	Member2	NameValueMap	iPart	See above.
	Member#	NameValueMap	iPart	See above.
Member1	Name	String	iPart	The name of an iPart member, this can be any iPart member name.
	3DModel	Boolean	iPart	Specify whether to publish the 3D model for the iPart member.
Member2	Name	String	iPart	The name of an iPart member, this can be any iPart member name.
	3DModel	Boolean	iPart	Specify whether to publish the 3D model for the iPart member.
Member#	Name	String	iPart	The name of an iPart member, this can be any iPart member name.
	3DModel	Boolean	iPart	Specify whether to publish the 3D model for the iPart member.

Export options for 3D PDF.

Option Name	Value Type	Default Value	Supported Documents	Supported Translators	Notes
AttachedFiles	String array		Part, Assembly	3D PDF	An array of string values indicating the full file paths.
ExportAllProperties	Boolean	True	Part, Assembly	3D PDF	Use this to publish ALL component properties. If set this to False, the ExportProperties can be used to specify which properties to export.

ExportDesignViewRepresentations	String array		Part, Assembly	3D PDF	An array of string values indicating the names of the design view representations.
ExportProperties	String array		Part, Assembly	3D PDF	A string array that indicates the properties to be exported. Use below format to specify a property to export: "PropertySet.InternalName:Property.Name". A sample is "{F29F85E0-4FF9-1068-AB91-08002B27B3D9}:Title" which indicates the Title property in "Summary Information" property set.
ExportTemplate	String		Part, Assembly	3D PDF	Specify the template PDF full file name.
FileOutputLocation	String		Part, Assembly	3D PDF	A string value indicates the PDF full filename to export the document to.
GenerateAndAttachSTEPFile	Boolean	False	Part, Assembly	3D PDF	Specify whether to generate the STEP file and attach it in PDF.
LimitToEntitiesInDVRs	Boolean	True	Part, Assembly	3D PDF	Specify whether the export scope is limited to the selected design view representations.
STEPFileOptions	NameValueMap		Part, Assembly	3D PDF	Specify the STEP save options when generate the STEP file.
ViewPDFWhenFinished	Boolean	True	Part, Assembly	3D PDF	Specify whether to view the PDF after the export is finished.
VisualizationQuality	AccuracyEnum	kMedium	Part, Assembly	3D PDF	Specify the visualization quality.

Definitions of Export Values

Option Name	Name	Value	Supported Documents	Notes
STEPFileOptions	ApplicationProtocolType	Integer value that indicates the application protocol type. Valid values are: eAP203 = 2, eAP214IS = 4 and eAP242 = 5.	Part, Assembly	
	Author	String value that specifies the author.	Part, Assembly	
	Authorization	String value that specifies authorization.	Part, Assembly	
	Description	String value that specifies description.	Part, Assembly	
	ExportFitTolerance	Double value that specifies export fit tolerance	Part, Assembly	
	Organization	String value that specifies organization	Part, Assembly	
	IncludeSketches	Boolean value that specifies whether include sketches or not.	Part, Assembly	

STL Export Format

When exporting an STL through the user-interface or using the API, you have an option of whether to include color information. Color information is not part of the STL standard but companies have figured out ways to add color information into the file in ways that allows standard STL readers to continue to be able to read the file and ignore the colors. If you have an STL reader and want to support colors from Inventor STL files you need to understand how the color is encoded within the binary STL file. Color is not supported in STL ASCII files. Inventor uses a format designed by Materialise and is described on Wikipedia.

© Copyright 2023 Autodesk, Inc.

Comment on this page.

XML Tags for FormattedText

This table and the tag list below indicates which text formatting tags are supported by the listed API calls. For example, the table shows that the HoleTag.FormattedText property supports style overrides, line breaks, fractions and super/subscripts. Details of these XML tags are can accessed by clicking on the relevant table column title, or just scroll down to the appropriate section.

API	Style Override	Line Break	Fraction, Sub/Superscript	Derived Properties	Document Properties	Physical Properties	Parameter	Prompted Text	Dimension Placeholder	DrawingView Name/Scale	Deli
DimensionText.FormattedText	✓	✓	✓				✓		✓		
DrawingNote.FormattedText	✓	✓	✓	✓	✓	✓	✓				
DrawingViewLabel.FormattedText	✓	✓	✓	✓	✓	✓				✓	
GeneralNote.FormattedText	✓	✓	✓	✓	✓	✓	✓				
GeneralNotes.AddByRectangle FormattedText arg	✓	✓	✓	✓	✓	✓	✓				
GeneralNotes.AddFitted FormattedText arg	✓	✓	✓	✓	✓	✓	✓				
HoleTableCell.FormattedText	✓	✓	✓				✓				
HoleTag.FormattedText	✓	✓	✓								
LeaderNote.FormattedText	✓	✓	✓	✓	✓	✓	✓				
LeaderNotes.Add FormattedText arg	✓	✓	✓	✓	✓	✓	✓				
RevisionTableCell.FormattedText	✓	✓	✓								
TextBox.FormattedText	✓	✓	✓	✓	✓	✓	✓	✓			
TextBoxes.AddByRectangle FormattedText arg	✓	✓	✓	✓	✓	✓	✓	✓			
TextBoxes.AddFitted FormattedText arg	✓	✓	✓	✓	✓	✓	✓	✓			
DrawingStandardStyle.SetViewLabelDefaults	✓	✓	✓	✓	✓	✓				✓	
DrawingStandardStyle.GetViewLabelDefaults	✓	✓	✓	✓	✓	✓				✓	

Style Override <StyleOverride>

The StyleOverride tag defines all text style information. The various style options are defined within the tag using attributes. These attributes are described below.

Font – Sets which font to use. The input to this attribute is the name of the font.

FontSize – Sets the size of the font. The input to this attribute is the size in centimeters.

Bold – Allows you to turn bolding on or off. The input to this attribute is the String "True" or "False".

Italic – Allows you to turn italics on or off. The input to this attribute is the String "True" or "False".

Underline – Allows you to turn underlining on or off. The input to this attribute is the String "True" or "False".

Example:

"<StyleOverride Font='Arial' Bold='True'>Notice</StyleOverride>: All holes larger than 0.500 <StyleOverride Font='AIGDT'>n</StyleOverride> are to be checked."

This results in the following text: **Notice**: All holes larger than 0.500 ∅are to be checked.

The first StyleOverride tag defines the font to be Arial and Bold. This is applied to the text “Notice” since that is what is between the opening and closing tags for this style override. The second StyleOverride just changes the font, but uses the font AIGDT which contains some common mechanical drafting symbols. In this example it uses the character “n” which corresponds to the diameter symbol in this font.

Line Break

This is the equivalent of a carriage return. Well-formed XML requires opening and closing tags, i.e.
</Br>. In the case of the line break there will never be text between the tags (this is call an “empty tag”) so you can simplify the definition and combine the opening and closing tags into a single tag using <Br/>.

Example:

"Line 1:
Line 2:
Line3:" results in the following text:

Line1:

Line2:

Line3:

Derived Properties <DerivedProperty>

The DerivedProperty tag specifies that information derived from the document will be used in the text string. Derived properties are different from standard document properties in that they are created "on the fly" from information associated with the document. Currently, if a derived property is used it must be the only text input for the formatted string. The StyleOverride tag can be used to control the style but no other text can be mixed with the derived property text. In order to define which derived property is to be used the following attribute is used:

DerivedID – The ID value that corresponds to a particular derived property. These ID’s are defined in the DerivedPropertyEnum enum list. Derived properties that are associated with a model will be extracted from the model associated with the first view placed on the sheet.

- kDerivedModelFilename – Filename of the model.
- kDerivedModelFilenameAndPath – Full path of the model.
- kDerivedModelVersion – Internal version number of the model.
- kDerivedDrawingFilename – Filename of the drawing.
- kDerivedDrawingFilenameAndPath – Full path of the drawing.

- kDerivedDrawingVersion – Internal version number of the drawing.
- kDerivedNumberOfSheets – The number of sheets in the drawing.
- kDerivedSheetNumber – The number of this sheet.
- kDerivedSheetSize – The size of this sheet.
- kDerivedSheetRevision – The revision number of this sheet.
- kDerivedSheetName – The name of this sheet.

Document Properties <Property>

The Property tag specifies that the value of a document property is to be used in the text string. Currently, if a document property is used it must be the only text input for the formatted string. The StyleOverride tag can be used to control the style but no other text can be mixed with the property text. In order to define which property is to be used the following attributes are used:

Document – The Document attribute is optional and used to specify which document the property value is to be extracted from. If this attribute is not specified, it uses the properties from the model document referenced by the first view placed on the sheet. Valid values for this attribute are "model" or "drawing". If "drawing" is used, the properties from the drawing document are used.

FormatID – The FormatID attribute specifies which property set the property is within. This is specified as a GUID in string form. The "Document Properties" topic in the API Overview discusses how to obtain the format ID's for specific property sets.

PropertyID – The PropertyID is a numeric value, which uniquely identifies a property within the property set.

The "Document Properties" topic in the API Overview discusses how to obtain the property ID's for specific properties. The Id of a property is the same as the value of the associated enum.

Example:

The example below will display the string associated with the description property of the design tracking properties associated with the document containing the model referenced by the first view on the sheet. The backslash at the end of the tag definition denotes that the opening and closing tags have been combined. This is useful in this case since a property tag does not require any additional information besides what's provided by the attributes.

<Property Document="model" FormatID="{32853F0F-3444-11d1-9E93-0060B03C1CA6}" PropertyID="29" />

Prompted Text <Prompt>

The prompt tag defines text that will prompt the user for the associated string. This type of text can only be placed within a sketch that is used for the definition of a title block, border, or sketched symbol. Currently, if prompted text is used no other text or tags can be used within the formatted text string. The StyleOverride tag can be used to control the style but no other text can be mixed with the prompted text. The override will be applied to the text entered by the user.

Examples:

The text between the opening and closing tags is used as the prompt string. The example below will create a string that will cause a dialog to be displayed with the prompt "Enter designers name:".

"<Prompt>Enter designers name:</Prompt>"

The text below will result in the same prompted string but the resulting text will have the style override applied.

"<StyleOverride Font='Courier New' Italic='True'><Prompt>Enter designers name:</Prompt></StyleOverride>"

Parameter <Parameter>

The parameter tag defines that the value of a parameter is to be used in the text string.

Example:

Here is a sample (this assumes that "C:\temp\test.ipt" is a document referenced by the drawing and contains a model parameter named "d0"). The ComponentIdentifier tag is not required if working within part sketches.

"<Parameter ComponentIdentifier='C:\temp\test.ipt' Name='d0' Precision='2' ></Parameter>"

Physical Properties From Model <PhysicalProperty>

The PhysicalProperty tag specifies that physical properties information derived from the model will be used in the text string. In order to define which physical property is to be used the following attribute is used:

PhysicalPropertyID - The ID value that corresponds to a particular physical property. These IDs are defined in the PhysicalPropertyEnum enum list. Physical properties that are associated with a model will be extracted from the model.

- kPhysicalModelMass - Mass of the model.
- kPhysicalModelArea - Area of the model.
- kPhysicalModelVolume - Volume of the model.
- kPhysicalModelDensity - Density of the model.

Example:

<PhysicalProperty PhysicalPropertyID='72449' Precision='3'></PhysicalProperty>

where '72449' represents the integer enum value representing the Mass property. If Precision is not specified, it defaults to 0.

Dimension Placeholder <DimensionValue>

This tag defines where the dimension value will be placed. If not specified, the tag will be placed at the beginning of the text.

DrawingView Name/Scale <DrawingViewName> and <DrawingViewScale>

These tags define where the drawing view name or scale will be placed, respectively. If not specified, the tag will be placed at the default location.

Delimiter <Delimiter>

This tag defines where the delimiter will be placed.

Hole Properties <HoleProperty>

The HoleProperty tag specifies that hole properties from the model will be used in the text string. In order to define which hole property is to be used, the following attribute is used:

HolePropertyID - The ID value that corresponds to a particular hole property.

These IDs are defined in the HolePropertyEnum list.

Example:

<HoleProperty HolePropertyID='77581' Precision='2'></HoleProperty>

where '77581' represents the long enum value of the hole diameter property (kHoleDiameterHoleProperty). If Precision is not specified, it defaults to 2.

Stacked Fraction, Superscript and Subscript <Stack>

The stack tag defines the stacking of sub-strings in the text string.

- "/" character within the substring specifies horizontal stacking.
- "#" character within the substring specifies diagonal stacking.
- "^" character prefix within the substring specifies a subscript.
- "^" character suffix within the substring specifies a superscript.
- An optional “FractionalTextScale” value can be specified.

Examples:

“1<Stack FractionalTextScale=’0.7’>1/2</Stack>” will result in $1\frac{1}{2}$

“1<Stack FractionalTextScale=’0.7’>1#2</Stack>” will result in 1½

“1<Stack>1#2</Stack>” will result in 1½

“H<Stack>^2</Stack>SO<Stack>^4</Stack>” will result in H_2SO_4

“x<Stack>2^</Stack> + y<Stack>2^</Stack> = z<Stack>2^</Stack>” will result in $x^2 + y^2 = z^2$

Model State <ModelState>

The ModelState tag specifies model state name in the text string.

Note - remember when using XML that certain characters must be represented by codes. The following table gives examples.

Code	Result	Description
&lt;	<	Less than
&gt;	>	Greater than
&amp;	&	Ampersand
&apos;	'	Apostrophe
&quot;	"	Quotation mark