MIGRATING SOLIDWORKS APPLICATIONS TO AUTODESK INVENTOR.

HINTS, TIPS AND RECOMMENDATIONS

Contents

Introduction

Who is this guide aimed at?

This guide is aimed at software developers who are already integrating their applications with SolidWorks, and who are considering (or actively) migrating their applications to work with Autodesk Inventor. The guide assumes you have a good working knowledge of the SolidWorks API, software development practices, COM and relevant programming languages.

How this guide helps you.

This guide provides you with a comparative overview of the core SolidWorks and Inventor APIs -- their similarities and differences, and how the key objects and constructs in SolidWorks can be emulated via the Inventor API.

This guide loosely follows the ten core topics of the Introductory Inventor API virtual training class, which can be downloaded here.

Although this guide touches upon various programming topics and debugging tools that are available to help understand the Inventor API, the intention isn't to describe in minute detail every aspect of the API. There are recommended learning resources available as a part of the Inventor product documentation and online. The following site is a good the next step for learning about the Inventor API in more detail: www.autodesk.com/developinventor.

COM APIs

Both SolidWorks and Inventor API's are based on Microsoft's Component Object Model (COM) technology. The products and respective APIs also perform very similar functions, which makes migrating from SolidWorks to Inventor a surprisingly straightforward task. You can start programming in Inventor using exactly the same COM-compliant languages and tools that you will have been using to work with the SolidWorks API.

In principle you can use any COM compliant programming language to access the Inventor API, including Microsoft .NET languages (VB and C#), and Visual C++. Autodesk Inventor also currently includes Microsoft's VBA IDE, which is the same technology framework that is used for writing macros and other VBA-enabled applications in SolidWorks.

A note about VBA - although the IDE is still available in Inventor 2011, Autodesk has announced that VBA will be removed from Inventor at some point in the future. So you are encouraged to learn how to use the Inventor 'add-in' infrastructure and use that as your primary method for future Inventor application development. If you have already worked with some Inventor VBA code, here's an excellent white paper on how that could be migrated to an Add-In application: http://modthemachine.typepad.com/files/vbatoaddins.pdf.

For simplicity, this guide uses VBA notation for code examples. Any type of application -- whether a macro, add-in, or EXE -- will use the same underlying object model, and VBA still provides the most concise way to illustrate the architectural aspects of the Inventor and SolidWorks APIs.

Elaborate error checking steps in the various bits of sample code are omitted in the interest of brevity. It is the responsibility of the programmer to ensure good error handling practice is adhered to, to help your production code be as robust as possible.

Application Integration Types

The table below summarizes the different types of applications supported by SolidWorks, and outlines the equivalent Inventor related application types.

Table 1: Application Types

Application Purpose	SolidWorks Application Type	In Inventor, Use…
To learn to program; inspect code and objects; create simple forms	VBA macros	VBA macros
Drive model generation from Microsoft Excel, Access, Visio etc.	VBA-enabled applications	VBA-enabled applications
External application that needs to access the API	Standalone EXEs	Standalone EXEs, Add-in EXEs, Apprentice
Custom embedded applications	Add-in DLLs	Add-in DLLs or EXEs

How Do You Decide?

Here's some additional information to keep in mind about the suitability of different types of applications in Inventor:

- Standalone EXEs can be developed where you have a program that uses Inventor but has its own interface and doesn't require the user to interactively work with Inventor. For example, a batch plot utility would typically be developed as a standalone EXE.
- An Add-In application is able to modify the user interface and integrates tightly within the Inventor environment. In Inventor, there is the choice of creating the add-in as a DLL, which will run in the same process as Autodesk Inventor, or an EXE, which will run in a separate process.

Almost all Add-Ins will be written as DLLs for increased performance benefits.

- Autodesk also provides a subset of the Inventor API via a dll library called 'Apprentice server'. If you need read-only access to Autodesk Inventor data such as the assembly structure, B-Rep, geometry, and file properties, you can develop an Apprentice based application. This is an ActiveX server that can be used within other applications and essentially provides read-only access to Inventor Documents.

Finding Your Way Around the Inventor API

Now that programming languages and target application-types have been discussed, let's take a closer look at some useful tools and some important areas for consideration when developing in the Inventor environment.

You Cannot Record Command Sequences But Master the Macros Anyway!

One big difference between Inventor and SolidWorks is that Inventor doesn't support recording command sequences in a macro. However, macros are still the easiest mechanism to get started with the Inventor API. Follow this link for a concise overview detailing steps to edit and run Inventor VBA macros.

Use a Different Ruler

All Inventor documents use the internal units of: Centimeters, Radians, Seconds, and Kilograms. Please keep this in mind as you migrate your code, as all SolidWorks APIs use meters for length by default, and metric units in general.

Simple VBA Macros

To study the differences between SolidWorks and Inventor macros, let's look at some very simple code from the Inventor VBA documentation.

This simple subroutine will display a message box with the name of the active document. Although trivially simple, this example eases you into Inventor's VBA environment. A few differences between SolidWorks and Inventor will be identified along the way. Refer to this guide for a detailed look at how to get started with programming in Inventor's VBA environment, but following is a quick overview:

1. Access VBA using the **Macro | Visual Basic Editor** command in the Tools menu, or by pressing Alt-F11;

2. A code module is automatically created named "Module1". Double-click on the module in the Project Explorer window, as shown in Figure 1. This will cause the code window for that module to be displayed.

Figure 1: Inventor's VBA Editor
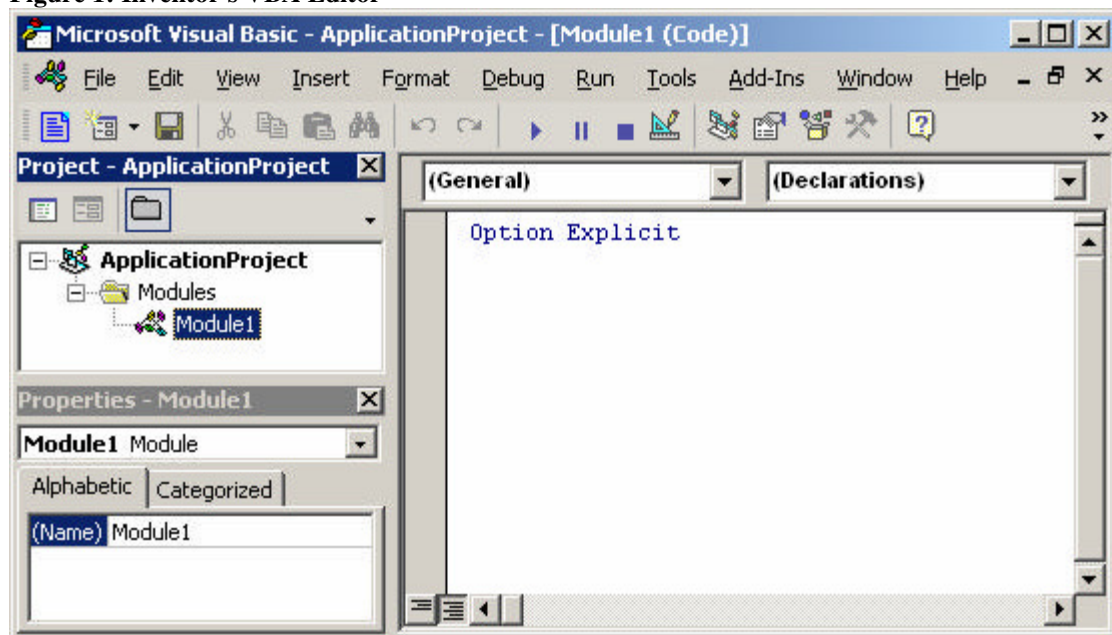
3. Copy and paste the Inventor code snippet below in the editor window, after the `Option Explicit` line shown in the figure.

Displaying Active Document Name in SolidWorks

Here's how the sub would look in SolidWorks:

```
Public Sub DocDisplayName ()
```

```
   Dim swApp As SldWorks.SldWorks

   Dim swDoc As SldWorks.ModelDoc2

   Dim sDocDisplayName As String

   Set swDoc = swApp.ActiveDoc

   sDocDisplayName = swDoc.GetTitle

   MsgBox (sDocDisplayName)

End Sub
```

This is How It Will Look in Inventor

```
Public Sub DocDisplayName ()

   Dim oDoc As Inventor.Document

   Dim sDocDisplayName As String

   Set oDoc = ThisApplication.ActiveDocument

   sDocDisplayName = oDoc.DisplayName

   MsgBox (sDocDisplayName)

End Sub
```

The Application and Document Are Right At Hand

Note the "ThisApplication" global variable in the Inventor example that provides direct access to the Inventor application object. Similarly, the "ThisDocument" variable provides direct access to the Inventor document. However, it's not available for all macros, only for projects contained within an Inventor Document.

Inventor's VBA supports three types of projects: document, application, and user, which are different in terms of the location in which the project is stored. Document projects are stored within Autodesk Inventor documents. Application and user projects are stored in external files.

The document project exposes the "ThisDocument" object representing the document that contains the macro. Writing code within the ThisDocument module gives you direct access to the Inventor document.

Exploring SolidWorks and Inventor Object Models

Charting the Objects

The APIs for both SolidWorks and Inventor are essentially a set of "hooks", called as interfaces, into the underlying capability of the software. The terms interfaces and objects are often used interchangeably in API documentation, and the Object Model provides a representation of the various objects, their properties, and functions that are exposed to the outside world.

The various SolidWorks objects are documented primarily through API training material and the API product documentation. Similar resources are also available for Inventor and additionally, there is an Inventor 2011 Object Model diagram available here in PDF form. This can be printed as a wall poster and serves as an excellent at-glance reference to the Inventor API. Figure 2 shows an image of this poster.

Figure 2: Inventor 2011 API Object Model
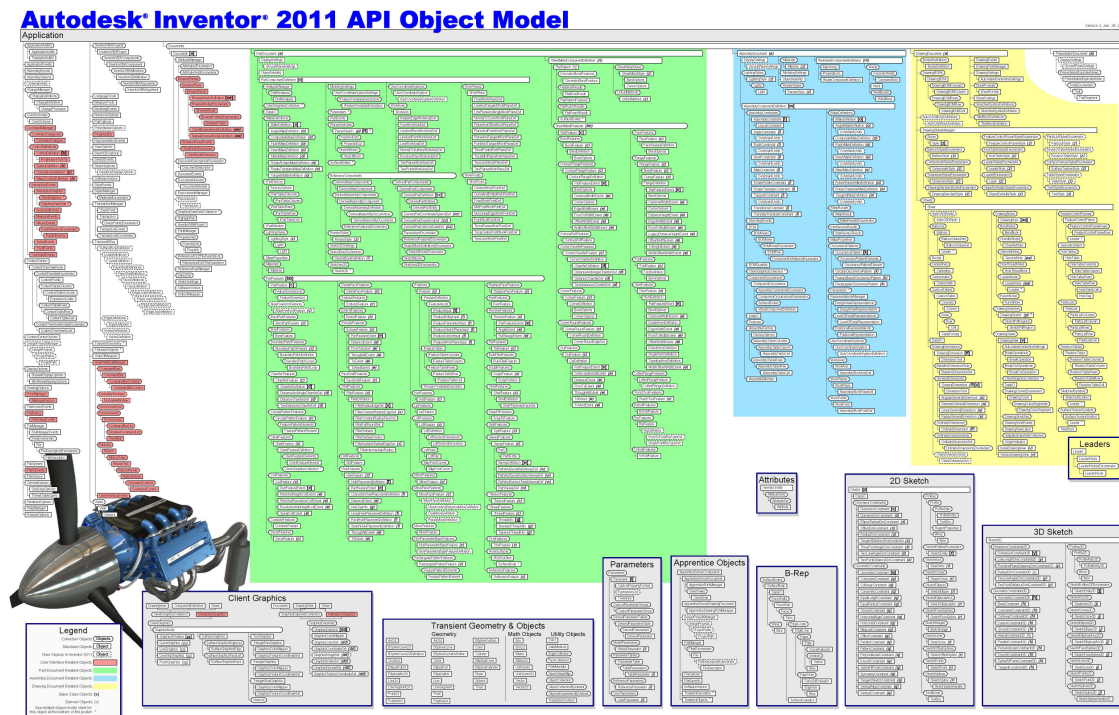
Finding the Objects That You Need

There are two very helpful tools that you can use in the VBA Editor as you get acquainted with the Inventor object model. Use the Object Browser (Figure 3) to review the contents of the Inventor type library.

Figure 3: The Object Browser
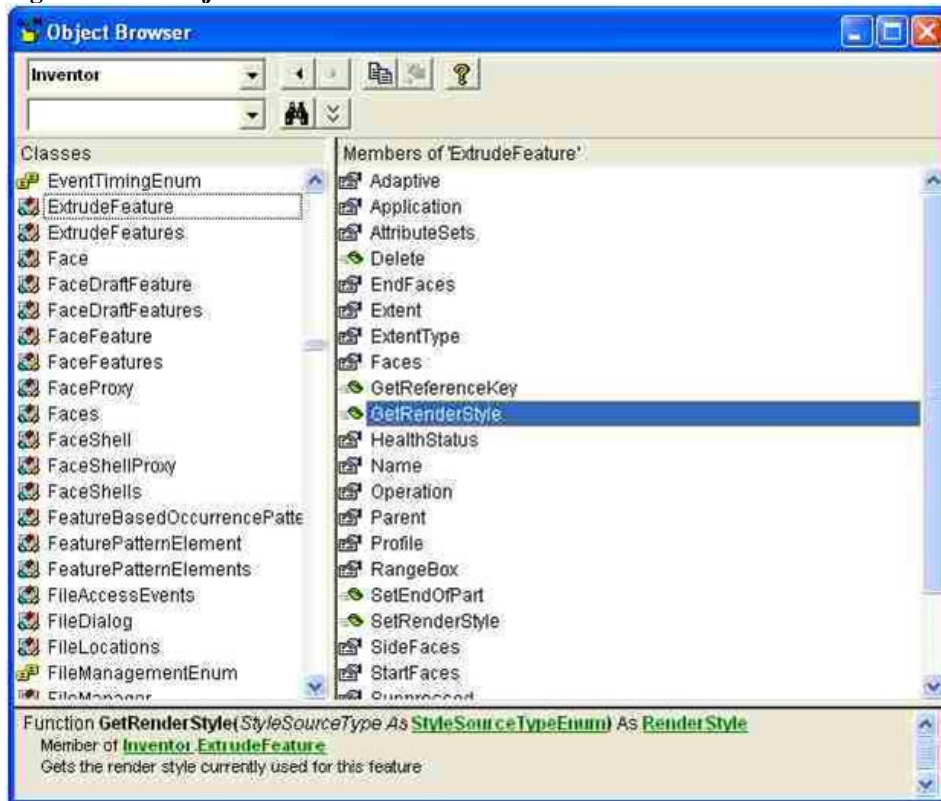
Access it by clicking on the ![icon] toolbar button or by selecting the **Object Browser** command from the **View** menu.

You can also obtain a "live" view of the object model as you are debugging your code by using the VBA Debugger (Figure 4). Use it to see the values of object properties and contents of collections, for example.

Figure 4: The VBA Debugger
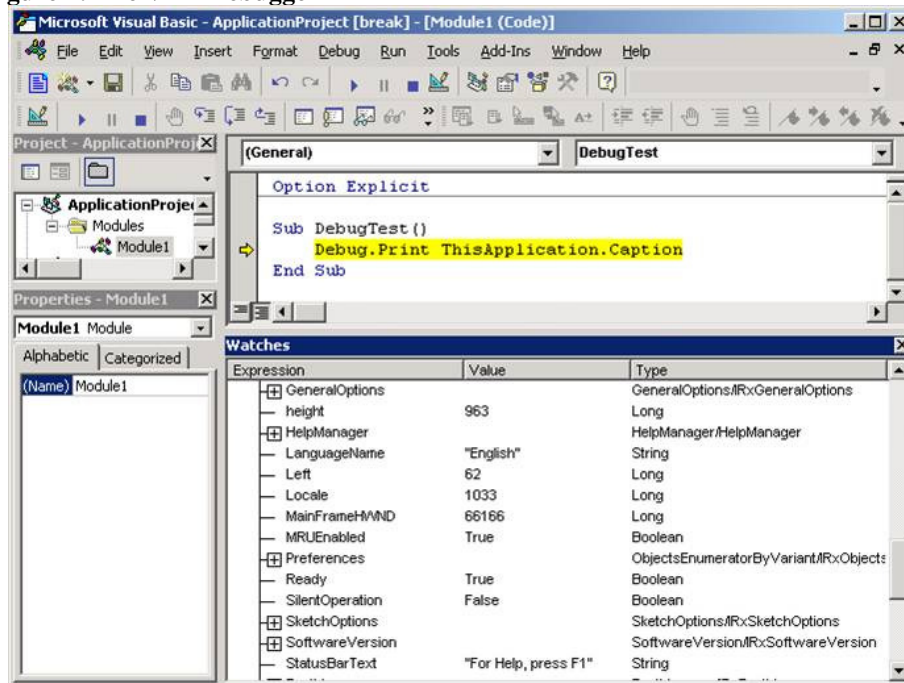
Getting the Right Bucket

In SolidWorks, the core API functionality is contained within the SolidWorks.Interop.sldworks namespace. Add a reference to the type library using the "References…" command in VBA or VB. In Visual C++ you would use the #import statement.

Besides the main type library, you would reference one or more additional type libraries in SolidWorks for constants, add-ins, and utilities. In Inventor, referencing the namespace Autodesk.Inventor.Interop provides access to all of the objects that make up the Autodesk Inventor API.

What Do You Need to Find?

The following table provides an overview of some of the key SolidWorks objects or object groups categorized by their functional area, and their Inventor equivalents.

Table 2: Mapping High-Level Objects

SolidWorks Object/ Object Group	Inventor Object/ Object Group
Application Interfaces	
SldWorks	Inventor
ModelDoc2	Document
PartDoc	PartDocument
AssemblyDoc	AssemblyDocument
DrawingDoc	DrawingDocument
Model Interfaces	
Attribute	Attribute
Body2	SurfaceBody
Face2	Face
Loop2	EdgeLoop
CoEdge	EdgeUse
Edge	Edge
Vertex	Vertex
Dimension	FeatureDimension, SketchDimensions
Modeler	TransientGeometry
Assembly Interfaces	
Component2	ComponentOccurrence
Interference	InterferenceResult
Mate2	AssemblyConstraint
Drawing Interfaces	
Layer	Layer
Sheet	Sheet

HoleTable	HoleTable
RevisionTableFeature	RevisionTable
TitleBlock	TitleBlock
View	DrawingView
Feature Interfaces	
Feature	PartFeature
Fillets	FilletFeatures
Holes	HoleFeatures
Mold Tools	CoreCavityFeature
Patterns	PartFeatures
Reference Geometry	Work Features
Sheet Metal	SheetMetalFeatures
Surface	PartFeatures
Weldments	Welds
Configuration Interfaces	
Configuration	iPartFactory, iAssemblyFactory
DesignTable	iPartTable*, iAssemblyTable*
Sketch Interfaces	
Sketch	PlanarSketch, Sketch3D
Annotation Interfaces	
CenterLine	Centerline
CenterMark	Centermark
Note	DrawingNote
BomTableAnnotation	PartsList, AssemblyComponentDefinition::BOM

Enumeration Interfaces	
EnumBodies2	SurfaceBodies
EnumComponents2	ComponentOccurrences
EnumDocuments2	Documents
EnumModelViews2	Document::Views, DrawingViews
Utility Interfaces	
CustomPropertyManager	PropertySets
EquationMgr	Parameters
MassProperty	MassProperties
MathUtility	TransientGeometry
SelectionMgr	SelectSet
User-interface Interfaces	
CommandManager	CommandManager, CommandBar, RibbonTab
FeatMgrView	BrowserPane
Frame	Views/ UserInterfaceManager methods?
ModelViewManager	Document, Views
PropertyManager	BrowserPane and/or Dockable Dialogs
Custom Interfaces	
SwAddin	ApplicationAddInServer

It's Arranged A Little Differently

Glancing at the object model diagram, there are quite a few singular and plural forms of the same object name -- Document and Documents, Parameter and Parameters, etc. There are also a lot of "Definition" objects, or object names of

the form "xxDefinition". For example, PartComponentDefinition, FilletDefinition, iMateDefintion, and so on. These are explained in a bit more detail below.

Collection Objects

In SolidWorks, collections are mainly used when working with the B-Rep; i.e. when building lists of specific faces, edges, and so on.

In the Inventor API however, most of the key objects have top-level collection objects. For example: Documents is the base collection that contains all open documents in the current session; the PartFeatures collection contains all the PartFeatures defined in a part, and so on.

A collection is a special type of object that provides iteration over existing objects. Collections, at a minimum, support the Count and Item properties which provide access to the number of objects within the collection and return specific objects from the collection respectively. They also support methods that allow you to create new objects.

Here's an example of how the collection object for extruded features makes it easy to get their count and names in Inventor:

```
Dim i As Long

For i = 1 To oExtrudeFeatures.Count

Debug.Print(oExtrudeFeatures.Item(i).Name)

Next
```

In SolidWorks, you would traverse the feature tree using the FirstFeature and GetNextFeature methods as we'll see in a later section, identify extruded features using the GetTypeName2 string, and increment a counter within the loop to accomplish the same.

One important thing to note with collections in the Inventor API is that the index always begins with '1' and not '0' for all collections.

Definition Objects

Definition objects are used in Inventor if you need to modify an object, and not just change its settings or visibility. For example, you will need to go through the PartComponentDefinition object if you need to change the BRep or geometric feature constraints for a part.

So for example, to add a sketch in SolidWorks, you would directly access the sketch object on the part document as follows:

```
Dim sketchManager As SldWorks.sketchManager

Set sketchManager = partDocument.sketchManager

sketchManager.InsertSketch True
```

In Inventor, you would need to go through the definition object for the part as follows:

```
Dim oCompDef As PartComponentDefinition

oCompDef = _InvApplication.ActiveDocument.ComponentDefinition

Dim oSketch As PlanarSketch

oSketch = oCompDef.Sketches.Add(oCompDef.WorkPlanes.Item(3))
```

Note that in some cases definition objects are abstract forms, from which you create instances. For example, TitleBlockDefinition is used to create TitleBlocks. Others like PartComponentDefinition and AssemblyComponentDefinition are concrete objects, associated with Part and Assembly geometry respectively in this case.

Getting Things Done

Now that we have a handy reference to the Inventor object model and a preliminary correlation with some of the key SolidWorks objects, let's take a closer look at some commonly used objects and tasks.

Accessing the Application Object

As we have seen in an earlier section, you can access the top-level Inventor application object from Inventor's VBA using the ThisApplication property. Additionally, you can access it from an add-in, which we'll cover a little later, and also from outside Inventor using GetObject or CreateObject methods.

Here's a very detailed explanation of various means available to you to connect with Inventor and to get access to the Application object.

Creating and Accessing Documents

Here's a side-by-side example of how you would open an existing document or a new one based on the default templates using Inventor API as compared to SolidWorks.

SolidWorks Code	Inventor Code
'Opens an existing document	
<pre>Public Sub OpenDoc()

  Dim swApp As SldWorks.SldWorks
  Set swApp = Application.SldWorks
  Dim fileerror   As Long
  Dim filewarning As Long

  swApp.OpenDoc6
  "C:\Temp\Part1.sldprt",
  swDocPART,
  swOpenDocOptions_Silent, "",
  fileerror, filewarning

End Sub</pre> | <pre>Public Sub OpenDoc()

  Dim oDoc As Document
  oDoc =
  _InvApplication.Documents.Open("
  C:\Temp\Part1.ipt")

End Sub</pre> |
| 'Creates a new document using the default template ||

20

```
Public Sub CreateDoc()

   Dim swApp As SldWorks.SldWorks
   Set swApp = Application.SldWorks

   'Get the default template file
   name with path for Part document
   Dim templateFileName As String
   templateFileName =
   swApp.GetDocumentTemplate(swDocu
   mentTypes_e.swDocPART, "", 0, 0,
   0)

   'Create a new SolidWorks part
   document with the default part
   template
   Dim partDocument As
   SldWorks.ModelDoc2
   Set partDocument =
   swApp.NewDocument(templateFileNa
   me, 0, 0, 0)

End Sub
```
```
Public Sub CreateDoc()



   'Get the default template file
   name with path for Part document
   Dim templateFileName As String
   templateFileName =
   _InvApplication.FileManager.GetT
   emplateFile(DocumentTypeEnum.kPa
   rtDocumentObject)

   'Create a new Inventor part
   document with the default part
   template
   Dim oDoc As PartDocument
   oDoc =
   _InvApplication.Documents.Add(Do
   cumentTypeEnum.kPartDocumentObje
   ct, templateFileName, True)

End Sub
```

Here are the main objects that we used in this example:

- SolidWorks: SldWorks, ModelDoc2
- Inventor: Inventor, Documents, FileManager.

Traversing Features

We have seen a code snippet above of how the PartComponentDefinition object would need to be accessed first in order to insert a sketch in Inventor. Creating a sketch is typically the first step of creating a part feature.

The PlanarSketch and Sketch3D API's in Inventor are very similar to the equivalent SolidWorks Sketch API, so let's focus on traversing existing features instead. This is typically a very frequent operation in SolidWorks, so it's important to understand how to traverse the features in Inventor.

Figure 5 shows the features of the tutor1f.sldprt part file in SolidWorks and nozzle.ipt part in Inventor. Both of these files are installed as standard samples with your installations.

Figure 5: Feature Trees in SolidWorks & Inventor
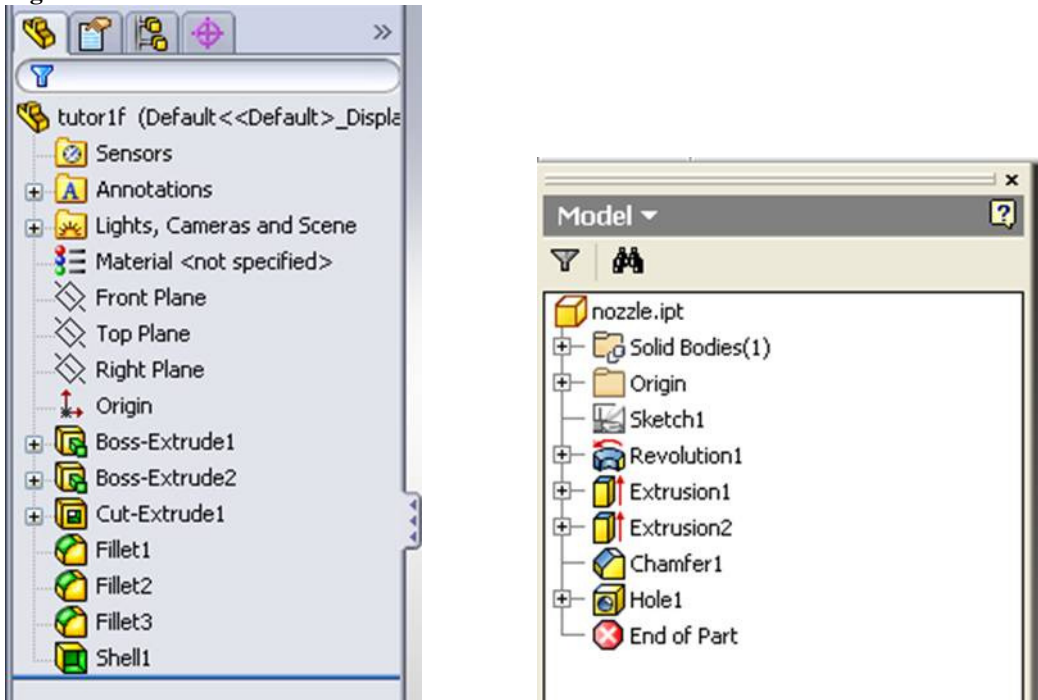
The collection objects in Inventor greatly simplify traversing features - we simply use the Features or PartFeatures collection. The code, as you can see below, is fairly straightforward:

```
Private Sub FeatureTraversal()

  Dim Doc As PartDocument

  Set Doc = ThisApplication.ActiveDocument

  Dim oFeature As PartFeature

  For Each oFeature In Doc.ComponentDefinition.Features

    Debug.Print oFeature.Name

Next

End Sub
```

This macro produces the following output, and can be compared with the model tree shown in the figure:

```
Revolution1

Extrusion1

Extrusion2

Chamfer1

Hole1
```

Accessing Inventor Part features is typically accomplished using this technique, however if you were looking to do something equivalent of traversing the SolidWorks FeatureManager Design Tree in Inventor, then you would use methods on the Model BrowserPane, which is similar to the FeatureManager user interface in SolidWorks.

The main objects that we would use for the model tree traversal are as follows:

- SolidWorks: SldWorks, ModelDoc2, Feature
- Inventor: Inventor, Document, BrowserPanes, BrowserNode, BrowserNodeDefinition.

Here is a side by side look at the above objects in action:

Table 3: Feature Tree Traversal

SolidWorks	Inventor

`Sub main()`
 	
`'First, obtain the Document object.`
 	
`   Dim swApp As SldWorks.SldWorks`
 	
`   Dim swModel As SldWorks.ModelDoc2`
 	
`   Set swApp = Application.SldWorks`
 	
`   If Not swApp Is Nothing Then`
 	
`       Set swModel = swApp.ActiveDoc`
 	
`    End If`
 	
`'Now obtain the first feature in the`
	`Private Sub QueryModelTree()`
	
	`'First, obtain the Document`
	`object.`
	`     Dim Doc As Document`
	
	`    If`
	`(ThisApplication.Documents.Count`
	`<> 0) Then`
	`         Set Doc =`
	`ThisApplication.ActiveDocument`
	`     End If`

```
FeatureManager design tree

    Dim swFeature As SldWorks.Feature

    Set swFeature = swModel.FirstFeature


'Print the feature name if the feature is
visible

'and move on to the next feature, if any

    While Not swFeature Is Nothing

        If Not
swFeature.GetUIState(swIsHiddenInFeatureMgr)
Then

            Debug.Print swFeature.Name

            Set swFeature =
swFeature.GetNextFeature

        End If

    Wend


End Sub
```

```
'Now obtain the top node for the
browser pane of the model tab.
    Dim oTopNode As BrowserNode
    Set oTopNode =
Doc.BrowserPanes("Model").TopNode



'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

'This 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
```

Here is what the output looks like in the VBA Immediate window:

Table 4: Feature Traversal Output

Comments	nozzle.ipt
Sensors	Solid Bodies(1)
Design Binder	Solid1
Annotations	Revolution1
Surface Bodies	Sketch1
Solid Bodies	Extrusion1
Lights, Cameras and Scene	Sketch1
Equations	Extrusion2
Material <not specified>	Sketch1
Front Plane	Chamfer1
Top Plane	Hole1
Right Plane	Sketch2
Origin	Origin
Sketch2	YZ Plane

| ```
Boss-Extrude1
Sketch4
Boss-Extrude2
Sketch6
Cut-Extrude1
Fillet1
Fillet2
Fillet3
Shell1
``` | ```
XZ Plane
XY Plane
Center Point
Sketch1
Revolution1
Sketch1
Extrusion1
Sketch1
Extrusion2
Sketch1
Chamfer1
Hole1
Sketch2
End of Part
``` |

As you can see, this has generated a listing of all the nodes in the model tree.

Accessing User Selections

User selections are typically made via the graphics area, model view or the client area (or whatever you like to call the model display area). The following section illustrates how items and objects in a model that can be selected via the screen are accessed through the API.

The example below compares how a single face, picked by the user on screen, is accessed programmatically, and its surface area calculated. Models used in the previous example are re-used here.

The respective objects used are:-

- SolidWorks: SldWorks, ModelDoc2, SelectionManager, Face2
- Inventor: Inventor, Document, SelectSet, and Face.

Note that both the Inventor SelectSet object and the SolidWorks SelectionManager have their first index as 1, not zero. Also, knowing what the internal default API units are, can you tell what units the area values will be in for each program?

Table 5: Onscreen Selection & Query Code

SolidWorks Code	Inventor Code
`Sub main()`	`Public Sub ShowSurfaceArea()`

SolidWorks Code	Inventor Code

```
    Dim swApp As SldWorks.SldWorks
    Dim swModel As SldWorks.ModelDoc2
    Dim swSelMgr As
SldWorks.SelectionMgr

' Set a reference to the
SelectionManager object of the active
document.
    Set swApp = Application.SldWorks
    Set swModel = swApp.ActiveDoc
    Set swSelMgr =
swModel.SelectionManager

' Check to make sure a single item
was selected.
    If
(swSelMgr.GetSelectedObjectCount2(0))
= 1 Then
        ' Check to make sure a face
was selected.
        Dim selType As Long
        selType =
swSelMgr.GetSelectedObjectType3(1, 0)
        If (selType =
SwConst.swSelFACES) Then
            ' Set a reference to the
selected face.
            Dim swFace As Face2
            Set swFace =
swSelMgr.GetSelectedObject6(1, 0)

            ' Display the area of the
selected face.
            MsgBox "Surface area: " &
swFace.GetArea & " m^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
```
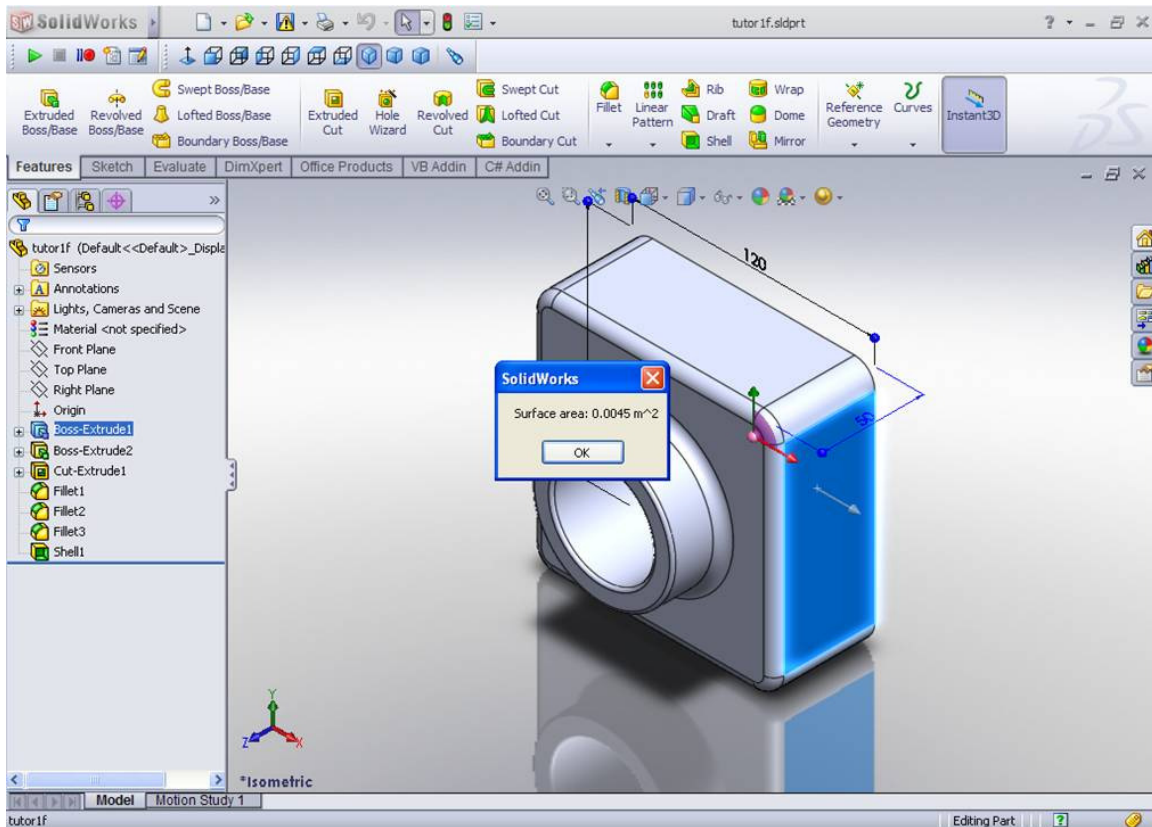
```
' Set a reference to the select set of
the active document.
    Dim oSelectSet As SelectSet
    Set oSelectSet =
ThisApplication.ActiveDocument.SelectSet

' 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
```

Assemblies -- New Terminology to Learn

Assemblies add a additional level of flexibility to modeling systems, and with it an additional level of complexity. The following concepts need to be understood when dealing with assemblies:- parts, part references, multiple instances of parts in an assembly, subassemblies; mate constraints, transforms, and interference. Taken in isolation these individual concepts are reasonably straight forward to understand. Together they provide a very powerful mechanism for building large scale models.

In Inventor, the API terminology for assemblies differs from that used in SolidWorks, and once you become familiar with these differences and understand the underlying concepts it becomes relatively easy to cross-correlate the objects and find new features and object within Inventor to supplement your toolbox to help you obtain even more power and control over assemblies. The

Inventor API online help documents provide an excellent introduction, so a brief overview is provided here.

Firstly the API object terminology. Table 6 details the key SolidWorks objects along with the equivalent Inventor API objects:

Table 6: Assembly Objects

SolidWorks Object	Inventor Object	Comments
Component	ComponentOccurence	As with a SolidWorks component, a ComponentOccurrence can also be a part or an assembly. There is also a ComponentOccurrences collection object.
Mate	Mate Constraint	
MathTransform	Matrix	In general, use the TransientGeometry object for various matrix transform and math utilities.

Proxy Objects

Inventor also uses a concept called 'Proxies', or 'Proxy Objects'. These are a set of objects that make it easy for you to work with parts, features, and geometric entities while they are being used within an assembly.

For example, a part may be located and oriented differently while being used in an assembly, and the coordinates of say, a vertex on that part will be different in the original part document than in the assembly document. A transformation matrix needs to be applied to go from one system to another.

Proxy Objects take care of this transform in Inventor. Simply stated, the proxy will obtain the data related to any sub parts or subassemblies in the context of the main assembly.

The following table details some of the main tasks typically performed on assemblies, along with the associated SolidWorks and Inventor APIs.

Table 7: Assembly APIs

Task	SolidWorks APIs	Inventor APIs
Add components into an assembly	AssemblyDoc::AddComponent5	ComponentOccurrences::Add
Locate and orient the components	• MathUtility::CreateTransform • Component2::Transform2	• TransientGeometry::CreateMatrix • ComponentOccurrence::Transformation
Define Joints between components	AssemblyDoc::AddMate3	AssemblyConstraints::AddMateConstraint
Check for interference	AssemblyDoc::ToolsCheckInterference2	AssemblyComponentDefinition::AnalyzeInterference
Traverse assembly components	• IConfiguration::GetRootComponent3 • IComponent2::GetChildren	• AssemblyComponentDefintion::ComponentOccurrence • ComponentOccurrence::SubOccurrences

To use proxies, the CreateGeometryProxy method of the ComponentOccurrence object is the main access point. You can request a proxy object for any entity that exists under the tree of that ComponentOccurrence.

Working with Drawings

Assuming you have an existing model that you would like to create a new drawing for, here is a look at functions that you would use in Inventor corresponding to those in SolidWorks.

Creating Drawing Sheets

The following methods are used to create a new drawing and add a blank sheet:

- Create a new drawing document: SldWorks::NewDocument --> Documents::Add
- Add a new sheet: DrawingDoc::NewSheet3 --> Sheets::Add
- Additionally, you will need to add a border and title block in Inventor using the following methods: Sheet::AddDefaultBorder, and Sheet::AddTitleBlock.

Placing Model Views On A Sheet

To insert standard orthographic views, equivalent of DrawingDoc::Create3rdAngleViews2 in SolidWorks, create a base view first and then add 2 projected views using the following methods:

- DrawingViews::AddBaseView
- DrawingViews::AddProjectedView

Use methods on the TransientGeometry objects to define the locations of the views.

Retrieving Dimensions From the Base Model

To insert dimensions from the underlying model in a drawing sheet in Inventor, use the Retrieve method of the GeneralDimension object as shown in the example below. This will produce results equivalent to the SolidWorks DrawingDoc::InsertModelAnnotations3 method.

```
Public Sub RetrievDimensionsFromModel()

Dim oDrawing As DrawingDocument

Set oDrawing = ThisApplication.ActiveDocument
```

31

```
Dim oView As DrawingView

Set oView = oDrawing.ActiveSheet.DrawingViews(1)

Dim oDimColl As GeneralDimensionsEnumerator

Set                             oDimColl                             =
oDrawing.ActiveSheet.DrawingDimensions.GeneralDimensions.Retrieve(oView
)

End Sub
```

If this does not achieve the desired result for any reason, create the dimensions using appropriate "GeometryIntent" objects. The concept of Geometry Intent is explained in the next section.

New Drawing Document Concept: Geometry Intent

In Inventor, methods to create drawing dimensions 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.

There is more information on using GeometryIntent objects, and drawing automation, in the Inventor API documentation and also the following article:

http://augiru.augi.com/content/library/au07/data/paper/DE111-4.pdf.

Customizing the User Interface

Start With a Wizard

Customizing the Inventor user interface is typically done via an add-in application. An add-in provides you with the widest access to user interface elements and the tightest integration. This is true in both Inventor and SolidWorks.

The easiest and best method to get started with add-in development in Inventor is to use one of the various Visual Studio wizards provided, which are similar to the wizards distributed with SolidWorks. Use the following step-by-step guide to have your skeletal add-in application up and running in a few easy steps: http://modthemachine.typepad.com/files/VBAtoAddIn.pdf. The steps attributed to Visual Basic 2008 Express in this paper will apply equally well to the latest Visual Basic 2010 Express edition as well.

When following the steps outlined in the above whitepaper, take a special note of the following:

- **ThisApplication No More:** An Add-In doesn't have the ThisApplication property and needs to get access to the Inventor Application object another way. The m_inventorApplication variable will provide this access.
- **Compare the Add-In Class and Methods:** Note the add-in object's class and required methods that correspond to your add-in in SolidWorks:
    - SwAddIn --> ApplicationAddInServer
    - SwAddIn::ConnectToSW --> ApplicationAddInServer::Activate
    - SwAddIn::DisconnectFromSW --> ApplicationAddInServer::Deactivate

Mapping the User Interface

The default user interface for Inventor 2011 is the "Ribbon" interface, which is based on the Microsoft Windows Presentation Foundation (WPF) technology. Inventor still supports the classic interface with menus and panel bars, which you will find comparable to the SolidWorks 2010 interface. The following discussion focuses on the Ribbon interface as this is the current default in Inventor, and represents the immediate future.

The figure below shows the Inventor 2011 user interface elements along with the corresponding SolidWorks element names (in parentheses). Refer to the Overviews articles in the User Interface Customization section of the Inventor API documentation for additional images and descriptions of the UI elements.

Figure 6: Comparing the User Interfaces

Comparing Customizable UI Elements

This section summarizes how the UI elements that can be customized in SolidWorks would be accessed and modified in Inventor.

Menus and Toolbars

Although the default Ribbon interface replaces the older menu and panel bars in Inventor, the following options are available to replicate your SolidWorks menus and toolbars in Inventor:

- Add them to a Ribbon tab;
- Use the classic Inventor interface and add to the menus and command bars.

If the API for the "classic" Inventor interface is used, and the user switches to the Ribbon interface the application program will continue to work and will automatically have its controls added to the Ribbon interface, but to have better placement of controls the Ribbon API should be used.

FeatureManager

As mentioned earlier, use the Browser Pane in Inventor to have an equivalent of the SolidWorks FeatureManager design tree.

PropertyManager

There is no direct equivalent of the SolidWorks PropertyManager page in Inventor. But there are a few other options:

- Construct a standard Visual Studio dialog to migrate the PropertyManager page;

- Create additional panes within the browser. Because the additional panes are just ActiveX control containers, they can contain any type of information and can even consist of other ActiveX controls;
- Create dockable windows that can host dialogs and controls.

Model Views

To add controls to the graphics screen, simply obtain View::HWND and use the relevant Microsoft Windows APIs. The View::Camera object can also be used to define or modify the model views.

Pop-up Menus

To achieve the equivalent of adding a command group to a shortcut menu in SolidWorks, use the context menu interface provided by the Inventor API.

Status Bar

Similar to the Frame object on the SolidWorks application object, the StatusBarText property of the Inventor application object can be used to set the text in the status bar. To provide 'real time' instructions to the end user during a specific interaction, use the StatusBarText property of the InteractionEvents object.

Mapping the User Interface API Objects

The table below summarizes the approximate equivalents of the key objects that you would use in SolidWorks to access the UI elements described above:

- **Command Group:** CommandManager::CreateCommandGroup --> RibbonTabs::Add
- **Command Item:** CommandGroup::AddCommandItem --> CommandControls::AddButton

- **Feature Manager:** ModelViewManager::CreateFeatureMgrControl3 -->
  BrowserPanes::Add
- **Pop-up Menus:** CommandManager::AddContextMenu -->
  CommandBarControls::AddButton
- **Status Bar Text:** StatusBarPane::Text --> Inventor::StatusBarText

Advanced Topics

The B-Rep API

The Boundary Representation method, abbreviated as B-Rep or BREP, forms the geometrical foundation for both SolidWorks and Inventor and indeed for the majority of modern CAD applications. It is a method to represent solid geometry in terms of its constituent "boundary" elements such as faces, edges etc., and their interrelationships.

B-Rep, and computer-aided geometric design (CAGD) in general is a vast topic, so this discussion is limited to accomplishing common geometry tasks familiar to SolidWorks users using the Inventor B-Rep API. For a more thorough discussion of B-Rep, please refer to the Inventor 2011 Online API Reference and the Virtual Inventor API Training Webcast B-Rep topic.

Comparing the B-Rep Models

When it comes to the B-Rep API's, one of the most important concepts to understand is the classification of B-Rep elements into "topology" and "geometry". Topology refers to the structure of a part whereas geometry refers to the spacial definition of the entities that make up the part.

Here is the topology object model in SolidWorks:

- SolidWorks: Body2 --> Face2 --> Loop2 --> CoEdge --> Edge --> Vertex

The corresponding topology objects in Inventor could be outlined as follows, not considering the collection objects:

- SurfaceBody --> FaceShell --> Face --> EdgeLoop --> Edge --> Vertex.

The Inventor FaceShell object provides an additional level of representation which could be useful for determining whether the body it belongs to is closed (as opposed to open sheet metal, for example), and for calculating volumes etc.

Table 8 shows the correspondence between the topology and geometry objects in SolidWorks and Inventor:

Table 8: Topology and Geometry Objects

SolidWorks Topology → Geometry	Inventor Topology → Geometry
Face2 → Surface	Face → Surface (BSplineSurface, Cone, Cylinder, etc., for example)
Edge → Curve	Edge → Curve (Arc2D, Circle2D, etc.)
Vertex → Point	Vertex → Point

Accessing Topology Objects

There are several ways to access the topological information of a model, including from features, from user selections, by proximity calculations, from attributes, and more. One method that is commonly used in both SolidWorks and Inventor is to traverse the B-Rep hierarchy starting with the body object.

Here are some of the accessor functions to bodies from SolidWorks and Inventor part and assembly documents:

- PartDoc::GetBodies2 --> PartComponentDefinition::SurfaceBodies
- Component2::GetBodies3 --> ComponentOccurrence::SurfaceBodies

To get to the faces in a body, the equivalent of SolidWorks' Body2::GetFirstFace and Face2::GetNextFace sequence in Inventor is as follows, starting with how the body object is accessed:

```
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

...
```

The B-Rep tree can also be traversed using short-cut accessors without traversing every child object, e.g.:-

- Face2::GetEdges --> Face::Edges
- Edge::GetStartVertex --> Edge::StartVertex

Evaluating Geometry

Once the required topology object(s) are obtained, the underlying geometry element can be queried, which provides the associated geometric information such as lengths, areas, curvatures, and so on.

Here are some of the accessor objects to the geometry for SolidWorks and Inventor:

- Face2::GetSurface --> Face::Geometry
- Edge::GetCurve --> Edge::Geometry
- Vertex::GetPoint --> Vertex::Point

The geometry can then be evaluated for information that is required for any subsequent calculations or analysis. As an example, a face can be examined to determine whether or not a face it is planar by using the Inventor equivalent of the SolidWorks Surface::IsPlane method. The 2D rectangular representation of it in the parameter space can also then be obtained if necessary. The following example evaluates a face, and then obtains the range box values using the equivalent of the SolidWorks' Surface::Parameterization, as follows:

41

```
Dim oFace As Face

If oFace.SurfaceType = kPlaneSurface Then

Dim oEval As SurfaceEvaluator

Set oEval = oFace.Evaluator

Dim oRange As Box2d

Set oRange = oEval.ParamRangeRect

...
```

Refer to the Inventor API help documentation for a complete list of evaluator objects and methods.

Persistent References

BRep objects tend to be transient in nature and change as the model features are modified. Any time the model is recomputed, the BRep object references become invalid. For example, a reference to a Face object becomes invalid if features that drive or impact on that face are added or modified.

In SolidWorks safe entities could be used to address such changes. Similarly, in Inventor, you can create a reference key to a topological entity before any features edits and then use this reference key to obtain that entity again after each feature has been added.

Reference keys provide a persistent reference to a particular BRep object between model recomputes.

For further details, search for the Reference Keys example in the API product documentation.

Custom Attributes

This section discusses custom attributes in SolidWorks, and how they can be translated into Inventor attributes. Attributes are used to store application-specific information along with the CAD model, and can be added to almost any object.

The following table outlines the correspondence between SolidWorks and Inventor attribute objects:

- **Define Attributes**
- SldWorks::AttributeDef --> [InventorEntity]::AttributeSet
- SldWorks::DefineAttribute --> AttributeSets::Add
- **Populate Attributes**
- SldWorks::Parameter --> AttributeSet::Attribute
- AttributeDef::AddParameter --> AttributeSet::Add
- **Attach Attributes**
- AttributeDef::CreateInstance5 --> Not needed in Inventor since the attribute is already attached to an object during creation.
- **Find Attributes**
- Just as feature traversal is the preferred method to search for attributes in SolidWorks and not B-Rep traversal, Inventor provides a special object for efficient attribute searches, called Document::AttributeManager.

Events and Notifications

Events are either user actions or software actions that are of interested to an application. The application may need to handle specific events, and respond to them in some way. The Notification capability in both SolidWorks and Inventor provide you with mechanisms to do that.

Start with Add-In Template Code

It is straight forward to test events in Inventor VBA, but not recommended that macros take advantage of them. Events are best managed from a full blown Add-In. A wizard-generated Inventor add-in project or the AddInEvents labs solution from the Inventor webcast archives are good starting points when trying to understand this topic -- you will find this in Module 10. Familiarity with handling notifications in SolidWorks is also essential.

Look for AddHandler Calls

Once you have the skeletal Inventor add-in application, look in the file StandardAddInServer.vb for calls to "AddHandler" in the Activate() function. As outlined in the section on Customizing the User Interface, the Activate() function is similar to the ConnectToSW() function in SolidWorks. Compare the AddHandler calls in SolidWorks as well, which can be reached by stepping through ConnectToSW(). These are the notifications that this sample application is going to handle.

Let's look at a specific AddHandler call in SolidWorks code that will notify us when a file is opened:

```
AddHandler iSwApp.FileOpenPostNotify, _

   AdddressOf Me.SldWorks_FileOpenPostNotify
```

Here's how the equivalent AddHandler call will appear in the Inventor-generated code:

```
AddHandler m_ApplicationEvents.OnOpenDocument, AddressOf
Me.m_ApplicationEvents_OnOpenDocument
```

Add Your Code to the Notification Functions

Now look for the subroutine with the name highlighted above, and that is where the definition of the OnOpenDocument handler is located. The SolidWorks code

to handle file open will need to appear in this function, modified appropriately to use Inventor objects. Also note that while the SolidWorks function in the above example will notify you *after* a file is opened, Inventor handles both before and after notifications in a single function and informs you which one is currently triggered via the *BeforeOrAfter* argument for the function.

Here's an example of how the function could be used to display the name of the document after it was opened:

```
Private Sub m_ApplicationEvents_OnOpenDocument(ByVal DocumentObject As
Inventor._Document, _ ByVal FullDocumentName As String, _ ByVal
BeforeOrAfter As Inventor.EventTimingEnum, _ ByVal Context As
Inventor.NameValueMap, _ ByRef HandlingCode As
Inventor.HandlingCodeEnum)

  Select Case (BeforeOrAfter)

    Case kAfter

      System.Windows.Forms.MessageBox.Show("OnOpenDocument: " +
      DocumentObject.DisplayName)

    Case kBefore

Debug.Print " Before"

    Case Else

Debug.Print " Aborted"

  End Select

End Sub
```

The Autodesk Inventor API supports notification of many events ranging across the entire API and notifications are available for commonly used objects.

This was just an outline describing how to migrate event handling code from SolidWorks to Inventor. For additional learning material on handling notifications in Inventor, please refer to the API Overviews in product documentation and the Module 10 presentation mentioned above.
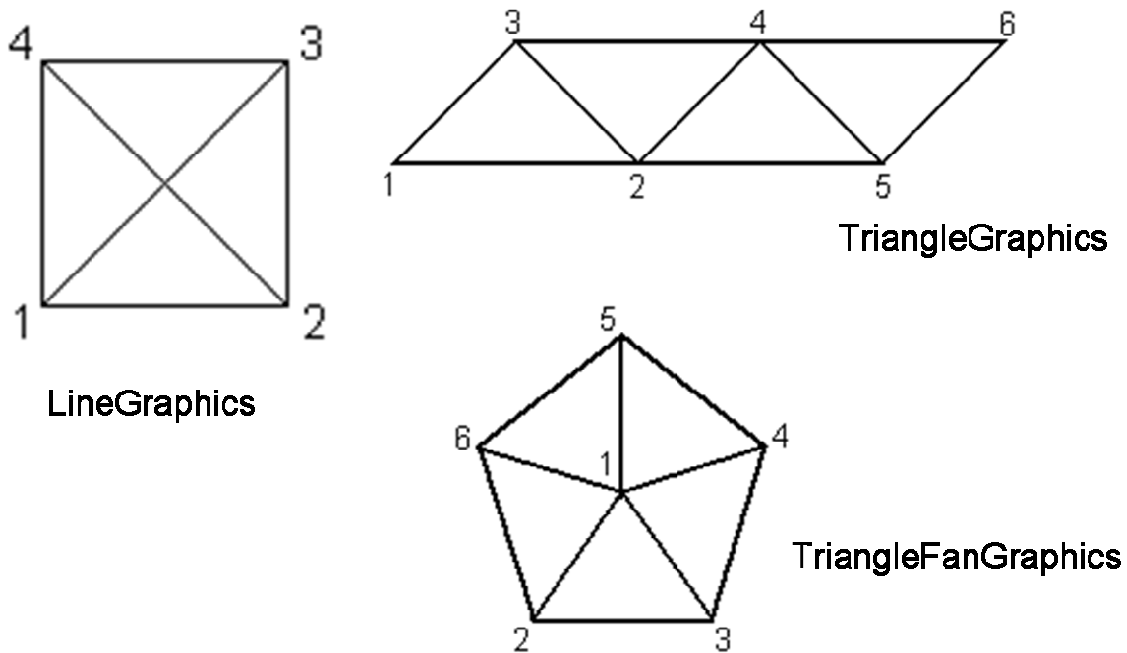
Graphics

Computer graphics is a vast topic and a basic understanding of interactive computer graphics concepts can help considerably when working with CAD graphics. Some of the key concepts that are typically taught in introductory computer graphics curricula include graphics pipeline, graphics hardware, transformations and viewing, mouse and keyboard input, lighting and surface shading, rendering optimization and so on. A detailed discussion of these topics is outside the scope of this paper.

Use Native Primitives

In SolidWorks add-ins can use the API to draw directly into the model view window using notifications available on the ModelView object, typically using OpenGL. The Add-In responds to repaint and buffer swap events so that custom graphics primitives can be added into the SolidWorks model window.

Inventor graphics are handled very differently. The Inventor API provides its own graphic primitives -- points, lines, triangles, text; collectively called "ClientGraphics" -- that are maintained and transformed by Inventor. Some of these primitives are shown in Figure 7.

Figure 7: ClientGraphics Primitives

TriangleGraphics

LineGraphics

TriangleFanGraphics

Like all graphics programming, creating Client Graphics follows a two step process, first creating graphics data, and secondly displaying this data using primitives. This separation of data from graphics allows a single set of data to be referenced by many graphics primitives.

Graphics for Real-time Interaction

A recently introduced object in Inventor called InteractionGraphics operates in a manner similar to regular ClientGraphics, except that it is available only when the user is interacting with the model. InteractionGraphics are much faster and so are well suited to real-time feedback during a command.
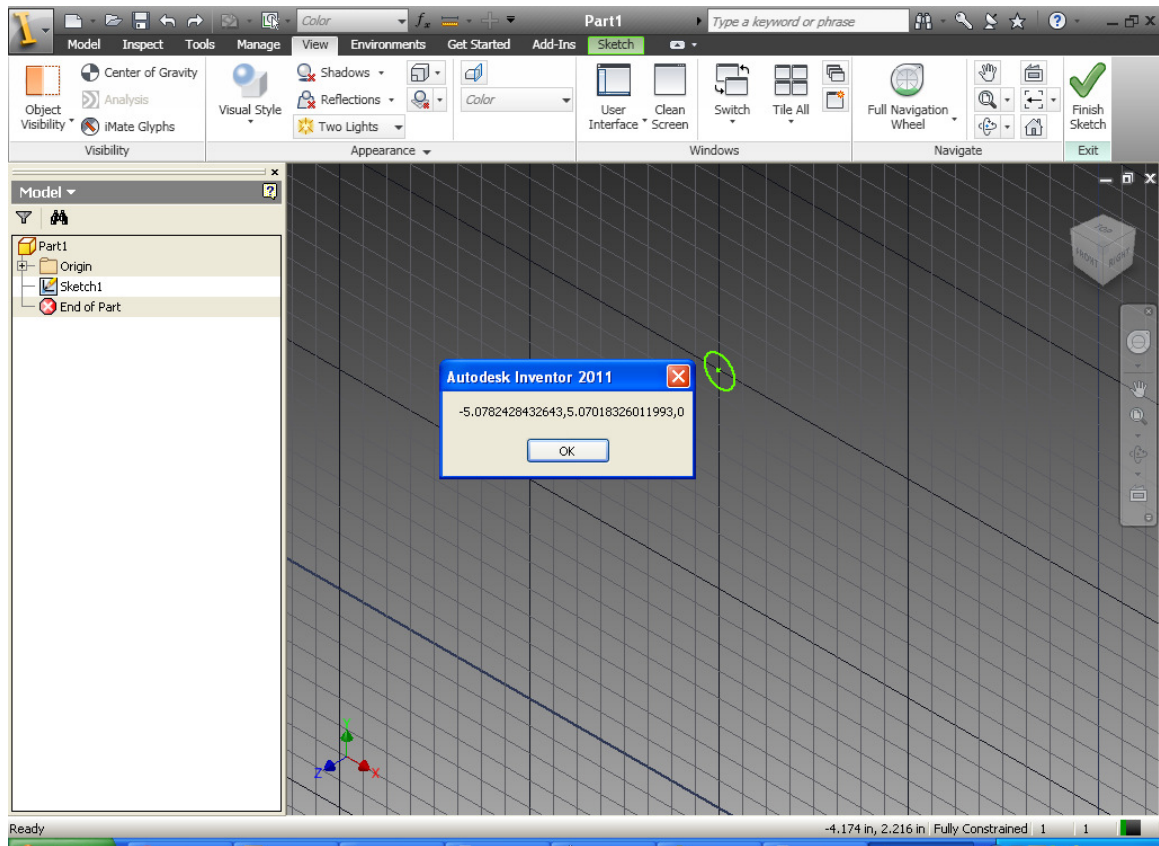
Mouse and Keyboard Inputs

In the SolidWorks API, if you want to perform custom graphical actions or calculations in response to mouse inputs – button clicks, move, drag, etc. – mouse event handlers would typically be added and the relevant callbacks implemented using the procedure outlined above. If, however, custom graphics objects are created using OpenGL and the Add-In needs to capture selection,

keyboard input and so on, then functions outside the SolidWorks API would be required. The typical method in that case would be to override the graphics window procedure with a call to a Windows SDK function, a technique known as subclassing the WndProc.

In Inventor, if ClientGraphics are used to create custom graphical entities, there is no need to use Windows SDK API's for mouse or keyboard inputs. The InteractionEvents object in the Inventor API provides access to mouse and keyboard events, and behaves like an Autodesk Inventor command. When started, the currently running command is terminated and it becomes the active command. All input from the user is then directed to the InteractionEvents object. Depending on which events you choose to subscribe to you can listen and respond to the user's input. Figure 8 shows the output from the MouseEvents example in the API training lab 10b which handles the OnMouseDown event to let the user select a point on a sketch plane and create a circle at that point.

Figure 8: MouseEvents Example

The other events supported by the MouseEvents object are also straightforward and are very similar to the mouse events available for VB/ VBA forms. Using these events you can receive notification that the mouse moved or a button was clicked and the coordinates, both model and view, where this occurred. The KeyboardEvents object is also obtained from the InteractionEvents object, and Keyboard events can be listened to in conjunction with the mouse events. Table 9 shows the corresponding API event handlers for some common mouse notifications.

Table 9: Mouse Event Handlers in SolidWorks and Inventor

Event	SolidWorks	Inventor
Left mouse button double clicked	MouseLBtnDblClkNotify	OnDoubleClick
Left-mouse button	MouseLBtnDownNotify	OnMouseDown

Event	SolidWorks	Inventor
pressed down		
Mouse pointer moved	MouseMoveNotify	OnMouseMove

Redo Undo Objects

One of the basic capabilities available to users in both SolidWorks and Inventor is to be able to undo an action by using a simple menu command or the CTRL + Z key combination, and likewise, to redo that action using the CTRL + Y keys. You can typically undo and redo up to ten previous steps this way per the default settings in either application, which can be changed by the user.

When a custom application creates a complex, high-level object in a single command or offers a new feature, users may expect to be able to undo and redo that operation in the same way as normal commands. The mechanism available to application Add-In developers for this involves the ability to group several API calls into a single unit that can then be undone or redone by the user using familiar menus and keyboard shortcuts.

The typical programming constructs and steps involved in implementing undo redo capability in SolidWorks and Inventor are as shown in Table 10:

Table 10: Redo Undo Objects

Description	SolidWorks	Inventor
1. Mark the start of the custom unit	ModelDocExtension::StartRecordingUndoObject	TransactionManager::StartTransaction
2. Mark the end of the custom unit	ModelDocExtension::FinishRecordingUndoObject	Transaction::End
3. Undoes the action(s)	ModelDoc2::EditUndo2	TransactionManager::UndoTransaction
4. Redoes the action(s)	ModelDoc2::EditRedo2	TransactionManager::RedoTransaction

In Inventor, this functionality is referred to as "Transaction". As described above, transactions essentially encapsulate the 'create', 'edit' and 'delete' operations within Inventor to behave as a single unit.

There is a lot more to transactions including when to use them and when not to; handling notifications of transactions; nested transactions and so on. Additionally, an advanced capability called Change Processor will let you automatically implement undo/redo behavior without requiring the use of transactions. This and more complete information about Transactions and Change Processor is available in the API Overviews section of the Inventor API help documentation.

Summary

This guide has been developed to help SolidWorks Add In developers assess the effort required to migrate their applications to work with Autodesk Inventor. The guide is also designed to provide a starting point for any migration activity.

If you are embarking on a project to port your SolidWorks application over to Autodesk Inventor, and require more information please feel free to contact:-

Gary Wassell
Developer Technical Services
Autodesk Global Subscription & Support

Autodesk Ltd.
1 Meadow Gate Avenue,
Farnborough Business Park,
Hants, GU14 6FG

www.autodesk.com/adn