PROGRAMMER'S GUIDE

Customizing SOLID EDGE™

MU28000-ENG

Version 15

Contents

Before You Begin

Who Should Read This Book

This book was written for Solid Edge users who want to customize or automate their modeling workflow. You should be familiar with using Solid Edge interactively and with the basic concepts of programming using ActiveX Automation. Although this book supplies sample code using the Microsoft Visual Basic® syntax, you can use any programming language that supports ActiveX Automation.

Running Sample Code

To run the sample code included in this book, create a new project in Visual Basic. Cut and paste the sample code into the Form Load section of the default form. On the Project menu, click References to add the Solid Edge Type Libraries to the project's references.

Note Microsoft recommends setting object variables to "Nothing" after you are finished using the object. This ensures that the object will be released from memory. For example:

```
Dim objX As Object
Set objX = CreateObject("Excel.Sheet")
MsgBox objX.Version
Set objX = Nothing
```

Accessing the Solid Edge Programming Help

The Solid Edge Programming Help is referenced throughout this document as a place to find more detailed information on objects, methods, and properties. On the Help menu, click Programming with Solid Edge to access the Solid Edge Programming Reference on-line Help file. You can also access context-specific information on any Solid Edge type library element by highlighting the object in the Visual Basic object browser and pressing F1.

Document Conventions

Quotation marks	Indicates a reference to an actual parameter or a variable name used in the sample code.
`Courier`	Indicates sample code.

If You Need Assistance

The Global Technical Access Center provides technical support for Solid Edge customers.

Telephone

In the USA and Canada, call 1-800-955-0000 or 1-714-952-5444. Outside North America, contact your local EDS office. For more information or the telephone number of an office near you, call 800-807-2200.

World Wide Web

For more information about Solid Edge, visit:

http://www.solidedge.com/

You can also access GTAC on the Web:

http://support.ugs.com/

Mail

Our mailing address is:
EDS, Inc.
675 Discovery Drive
Suite 100
Huntsville, Alabama 35806
U.S.A.

Technical Support

For more details about technical support, follow these steps:

1. In Solid Edge, on the Help menu, click Solid Edge Help.
2. On the Help dialog box, click the Index tab.
3. On the Index tab, type "support."
4. Click the Display button to display information about Solid Edge support.

C H A P T E R 1

Introduction

This chapter contains overview information on how to use various programming utilities. Prerequisites and suggested references are also included.

Purpose of the Programming Utilities

The Solid Edge programming utilities allow you to quickly customize or automate Solid Edge with client-server automation. With these tools, you can modify and enhance standard commands to tailor Solid Edge to your specific needs. You can also create tools that allow you to reduce or automate repetitive tasks.

Customizing Solid Edge is easy and efficient. Using your favorite standard Windows programming tools and languages, such as Visual Basic or Visual C++, you can create command extensions that precisely match the needs of your workplace. Through the automation of repetitive tasks, you save time, promote consistency, and ensure design integrity.

What You Should Know Before You Start

To customize Solid Edge, you need access to a Windows programming tool to be used as an automation client. An automation client is an application that can access objects from other applications that support client-server automation. There are several applications that can be used to create an automation client, such as Visual Basic and Visual C++. Any product that supports client-server automation that follows the Microsoft COM automation model can be used to automate Solid Edge.

This book is tailored primarily for the Visual Basic user. Suggested self-paced training for Visual Basic is included in Appendix A, *Learning Visual Basic*. Visual Basic allows you to quickly create an application that uses features from other applications.

Because many of the programs you create will automate the interactive workflow, you should have a working knowledge of that workflow. That is, you need to understand the task you are automating. A set of tutorials is delivered with Solid Edge to teach you many of the fundamentals for using the product. On the Help menu, click Tutorials to access the on-line tutorials.

Some Visual Basic program examples are delivered with Solid Edge. See Appendix B, *Sample Programs*, for a description of the samples and for information on how to access them.

Using this Document

Solid Edge consists of five environments: Part, Sheet Metal, Weldment, Assembly, and Draft. Part, SheetMetal, and Weldment share a programming interface; Assembly and Draft each have a unique programming interface.

This document begins with generic information about how to program Solid Edge. Chapter 2 defines COM automation and provides a basis for the remainder of the document. If you do not have any previous experience with COM automation, this is an important chapter to study. Chapter 3 describes some helpful Visual Basic and Solid Edge tools.

The remaining chapters describe specific components of the product. Some components are only available within a single environment; others are available in all environments.

C H A P T E R 2

Introduction to COM Automation

This chapter contains general information on how to access Solid Edge objects and their properties and methods.

Introduction to COM Automation

COM is the term used to describe the Microsoft component object technology. The ability to customize Solid Edge is based on an important part of this object technology—COM automation. This is the same technology frequently referred to as OLE automation.

Through automation, Solid Edge exposes functions called objects to programming languages.

Objects, Properties, and Methods

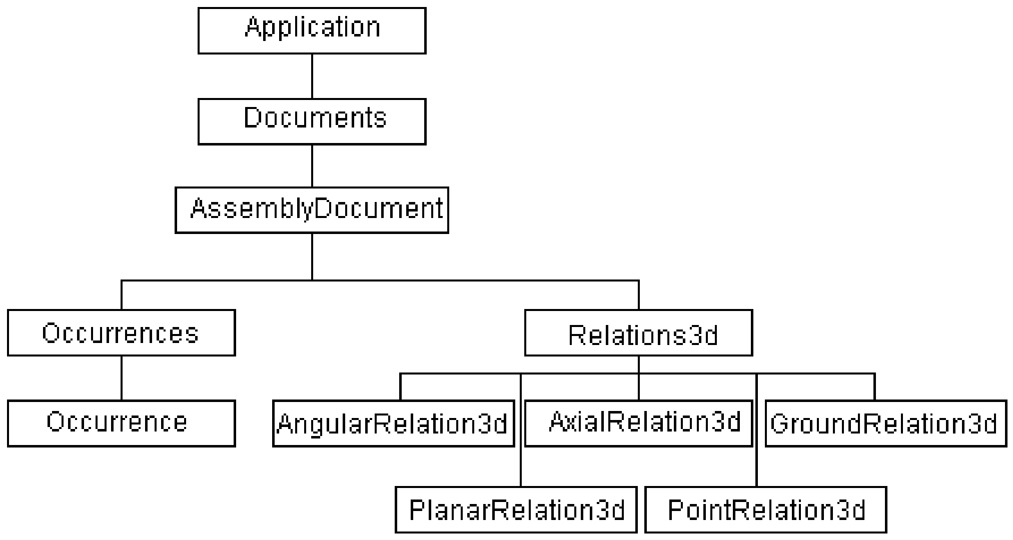
Properties are the characteristics that define an object (for example, the length and angle of a line). In Solid Edge, changes to an object's properties usually change the object's graphics. Methods are the operations that can be performed on an object (such as copy). For example, a circle as a graphic object has certain characteristics. Its location, color, and radius are examples of the circle's properties. Methods for the circle include copy, move, delete, and so on.

The concepts of properties and methods are common to Visual Basic programming, as Visual Basic uses object-oriented programming techniques. An example of an object in Visual Basic is the command button. When you place a command button on a dialog box, properties are provided to control the appearance and behavior of the button. Some of these properties are Caption, Font, Height, Width, Name, and Tag. The command button also supports methods such as Drag and Move.

When you program with Solid Edge, every part of the product—every document, every model, every part, and every feature of that part—is treated as an object and has its own set of properties and methods. Using these objects and their properties and methods, you can interact with Solid Edge to create, edit, and query the objects. You can program these objects using Visual Basic because Solid Edge has exposed, or made them available, to automation.

Understanding the Object Hierarchy

Objects are related to one another in a form that is called an object hierarchy. Customizing Solid Edge requires an understanding of the various hierarchies that form the automation interface. In this document, an object hierarchy is depicted as a cascading diagram, with the parent object always being the application. Solid Edge consists of five environments: Part, Sheet Metal, Weldment, Assembly, and Draft. Part, Sheet Metal, and Weldment share an object hierarchy; Assembly and Draft each have their own unique hierarchy. A simplified object hierarchy of the Assembly module is as follows:



Many of the items in the hierarchy are listed twice; one is plural and one is singular (for example, Occurrences/Occurrence). The plural names are collection objects. A collection is a special type of object whose purpose is to provide a way to create and access all of the objects of a specific type. For example, the Occurrences collection has methods to place Occurrence objects (instances of parts and subassemblies). It also contains all the Occurrence objects that have been placed, and provides the interface to access them.

The following program, which places two parts into an assembly and positions them, shows how you traverse parts of the hierarchy to automate Solid Edge.

```
 'Declare the program variables.
 Dim objApp as Object
 Dim objOccurrences as Object
 Dim objBox as Object
 Dim objLid as Object

 'Connect to a running instance of Solid Edge.
 Set objApp = GetObject(,"SolidEdge.Application")

 'Open an existing document.
 Call objApp.Documents.Open("c:\My Documents\Drawing
Files\file.asm")

 'Access the Occurrences collection.
 Set objOccurrences = objapp.ActiveDocument.Occurrences
 'Place two grounded parts.
 Set objBox = objOccurrences.AddByFilename("C:\My Documents\Drawing
Files\Box.par")
 Set objLid = objOccurrences.AddByFilename("C:\My Documents\Drawing
Files\Lid.par")

 'Move the lid to the correct location.
 Call objLid.Move(0, 0, .055)
```

This program connects to a running instance of Solid Edge, places two parts, and repositions one of them.

The first lines of the program set up the variables, which are declared as type "Object." Object variables are used exclusively with COM automation to store objects that are exposed by an application. Object variables can also be declared as specific Solid Edge object types, which is known as *early binding*. For example, in the preceding code fragment, the objApp variable could have been declared as the Solid Edge Application object type. Early binding and late binding are advanced Visual Basic subjects, but for those readers who are familiar with them, the Solid Edge type libraries support early binding.

After declaring the variables, the program connects to the Solid Edge application. The GetObject function is a standard Visual Basic function that connects to an application and returns a reference to the application as an object. In this case, the first argument is empty, and the second argument specifies the type of application to be returned. The application object is stored in the object variable objApp. The Set statement assigns an object reference to an object variable. The GetObject function is described in more detail in Chapter 4.

After connecting to the Solid Edge application object, the program opens an existing document and accesses the Occurrences collection object from it. All methods to create objects are accessible through collection objects. In this case, we want to add a part to an assembly, so we use an Add method provided by the Occurrences collection.

To access objects in the application, use a period to step down through the hierarchy. To access the Occurrences collection object, you establish a connection to the Application object, step down through the hierarchy to the AssemblyDocument object, and step to the Occurrences collection object.

Some objects provide properties and methods that enable you to access an "active" object. For example, Solid Edge can have multiple documents loaded, yet only one document can be active at a time. The Application object has a method, ActiveDocument, that enables you to access this document. The example program accesses the Occurrences collection from the active document; the reference to this collection is stored in the objOccurrences variable.

It is not required that the Occurrences collection be stored in a variable, but doing so creates a more optimized and readable program. There is nothing syntactically incorrect in expressing a statement as follows:

```
Set Box =
objApp.ActiveDocument.Occurrences.AddByFilename("Box.par")
```

However, this statement requires Visual Basic to resolve the references of the methods and properties specified, which increases the processing time of your program. If you will need to reference an object more than once, your program will run faster if you assign the object to a variable. It can also make your code easier to read.

After establishing a connection to the Occurrences collection object, the example program places two occurrences—in this case, parts—using the AddByFilename method. This method uses as input the file name of the part or subassembly to be added. The Set statement assigns the object variables objBox and objLid to the new Occurrence objects. These objects can be used in subsequent statements. If you do not need to use the objects later, you can create them without storing the resulting object.

In the final lines of the program, objLid is set when the lid part is placed into the file, and the Move method is called to move the lid to the correct location.

This example shows the importance of understanding the object hierarchy. The purpose of this program is to place two parts, but to do that, you must trace down through the hierarchy to the collection that can place parts. This is similar to all other tasks you may perform with the Solid Edge automation interface. You must determine where the methods that you need are, and then access those methods by tracing through the hierarchy, beginning at the application object.

Understanding the hierarchy is only one part of the process. The other part is to understand the methods and properties that each object supports. You can use the on-line Help and the Object Browser to learn about the hierarchies, objects, properties, and methods that Solid Edge exposes. Information on how to use these tools is provided in Chapter 3.

Dereferencing Object Variables

Whenever you use object variables in Visual Basic, you must ensure that any references to objects are released before your object variables go out of scope. If you declare an object variable within a subroutine or function, you must set that object variable equal to Nothing within that function or subroutine. If you declare a public or private object variable in a form or module, you must set that variable equal to Nothing in a corresponding unload or terminate event.

If you fail to dereference object variables, your application will use memory inefficiently. It is also likely to cause run-time errors, possibly causing Solid Edge to abort as well.

It is good programming practice to write the lines of code that dereference your object variables at the same time that you write the declaration statements for the object variables.

For brevity, this important programming step is omitted within the code samples in this document.

Handling Measurement Values

Internally, Solid Edge converts all measurements to a uniform set of units, sometimes referred to as database units. All methods and properties expect database units as input. Therefore, when automating Solid Edge, you must first convert user input to these internal units. Conversely, because calculations and geometric placements use the internal units, you must convert from internal units to default units when displaying output. The UnitsOfMeasure object handles these conversions.

The following internal units are used:

Unit Type	Internal Units
Distance	Meter
Angle	Radian
Mass	Kilogram
Time	Second
Temperature	Kelvin
Charge	Ampere
Luminous Intensity	Candela
Amount of Substance	Mole
Solid Angle	Steradian

C H A P T E R 3

Using Programming Tools

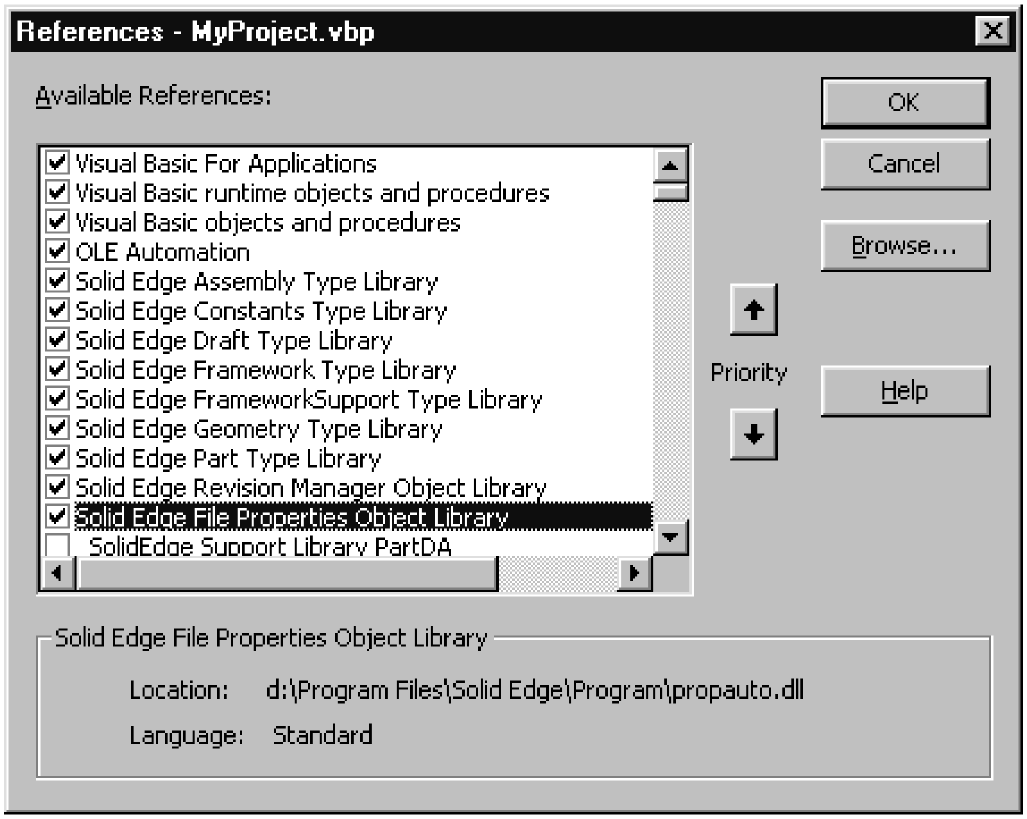
This chapter describes how Solid Edge and programming tools interact to gather input.

Using the Visual Basic Object Browser

Visual Basic's Object Browser allows you to examine objects and review their supported properties and methods. Because the Object Browser classifies a child object as a property of its parent object, you can also determine the object hierarchy from within the Object Browser. For example, a Sheet object is the parent of the Lines2d collection, so the Object Browser shows the Lines2d collection as one of the properties of the Sheet object.

The information about an application's objects is contained in files known as object libraries (OLB files), type libraries (TLB files), or dynamically linked libraries (DLL files). Solid Edge delivers several TLB files and DLL files.

To make Solid Edge type libraries available to the Visual Basic Object Browser, click References on the Project menu. On the References dialog box, select the Solid Edge libraries you want to access.


References - MyProject.vbp

Available References:

- ✓ Visual Basic For Applications
- ✓ Visual Basic runtime objects and procedures
- ✓ Visual Basic objects and procedures
- ✓ OLE Automation
- ✓ Solid Edge Assembly Type Library
- ✓ Solid Edge Constants Type Library
- ✓ Solid Edge Draft Type Library
- ✓ Solid Edge Framework Type Library
- ✓ Solid Edge FrameworkSupport Type Library
- ✓ Solid Edge Geometry Type Library
- ✓ Solid Edge Part Type Library
- ✓ Solid Edge Revision Manager Object Library
- ✓ Solid Edge File Properties Object Library
- ☐ SolidEdge Support Library PartDA

Solid Edge File Properties Object Library

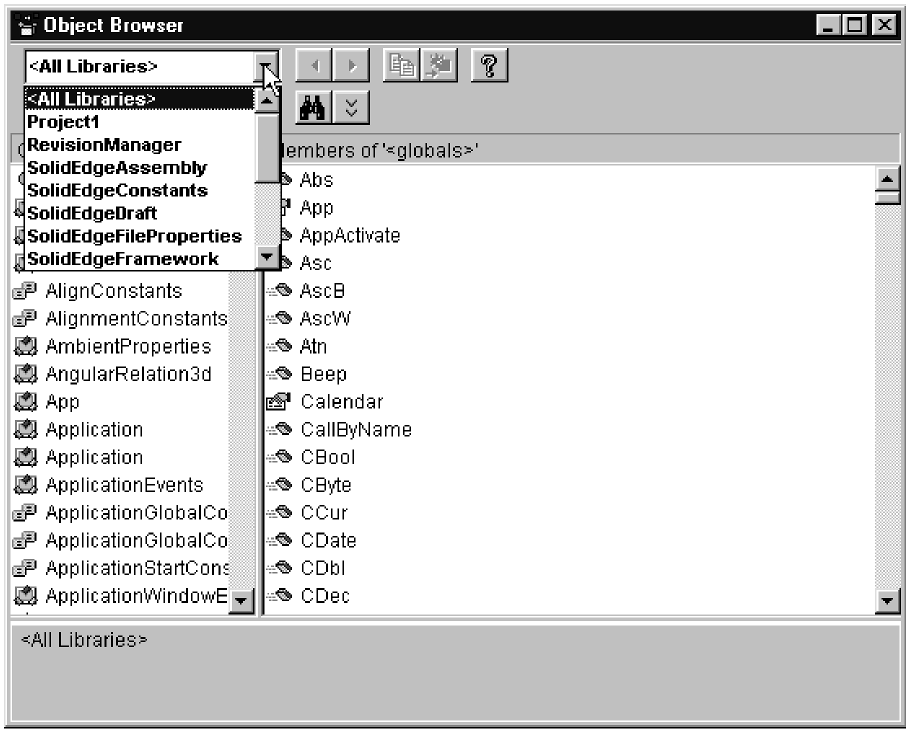
Location: d:\Program Files\Solid Edge\Program\propauto.dll

Language: Standard

Once you have added the Solid Edge libraries to your project, you can click Object Browser on the View menu to access the Object Browser dialog box.

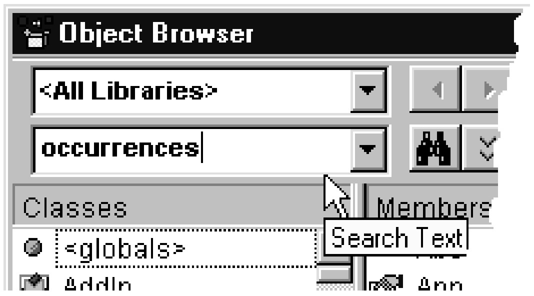
Project/Library

The Project/Library list, which is located in the top left of the dialog box, displays all available libraries. The following libraries are delivered with Solid Edge: Assembly, Constants, Draft, File Properties (delivered as a DLL), FrameWork Support, FrameWork, Geometry, Part (which includes SheetMetal and Weldment), and Revision Manager.



Search Text

The Search Text box, located below the Project/Library list, allows you to search for a string within the specified library. You can use standard Visual Basic wildcard characters to extend a search. In addition, you can limit the search to find only whole words by setting the Find Whole Word Only option on the shortcut menu.



Classes

The Classes list contains all of the objects in the selected library. Class is an object-oriented programming term for a type of object. For example, a line is one class of object, while a circle is another.

You can use the class names when declaring object variables. For example, the following syntax shows how to declare a variable that will be used to hold a line object:

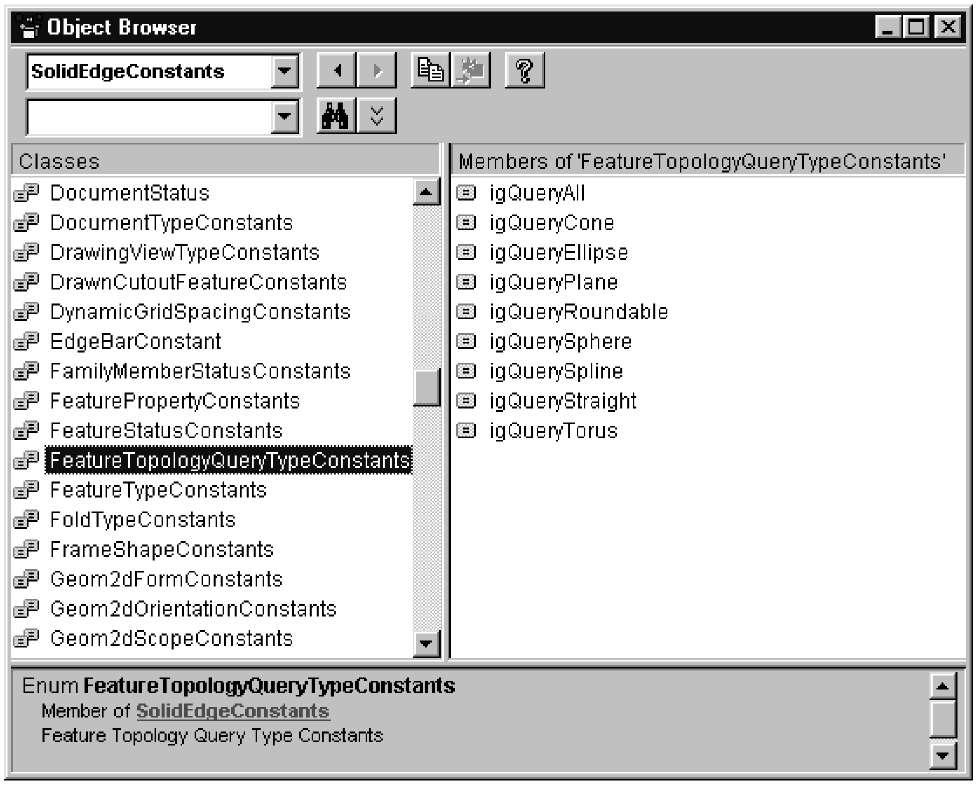
```
Dim Line1 as Object
```

With the correct type library attached, you can also declare the line to be a specific class. In the following syntax, Line1 is declared as class Line2d:

```
Dim Line1 as Line2d
```

Declaring an object as a specific object type allows Visual Basic to do type checking, creates faster running code, and makes the code more readable. The generic Object type is useful when you do not know what type of object will be assigned to the variable until runtime or if the variable will hold many types of objects.

Constants are also listed in the Classes section of the Object Browser.



Classes:
- DocumentStatus
- DocumentTypeConstants
- DrawingViewTypeConstants
- DrawnCutoutFeatureConstants
- DynamicGridSpacingConstants
- EdgeBarConstant
- FamilyMemberStatusConstants
- FeaturePropertyConstants
- FeatureStatusConstants
- FeatureTopologyQueryTypeConstants
- FeatureTypeConstants
- FoldTypeConstants
- FrameShapeConstants
- Geom2dFormConstants
- Geom2dOrientationConstants
- Geom2dScopeConstants

Members of 'FeatureTopologyQueryTypeConstants':
- igQueryAll
- igQueryCone
- igQueryEllipse
- igQueryPlane
- igQueryRoundable
- igQuerySphere
- igQuerySpline
- igQueryStraight
- igQueryTorus

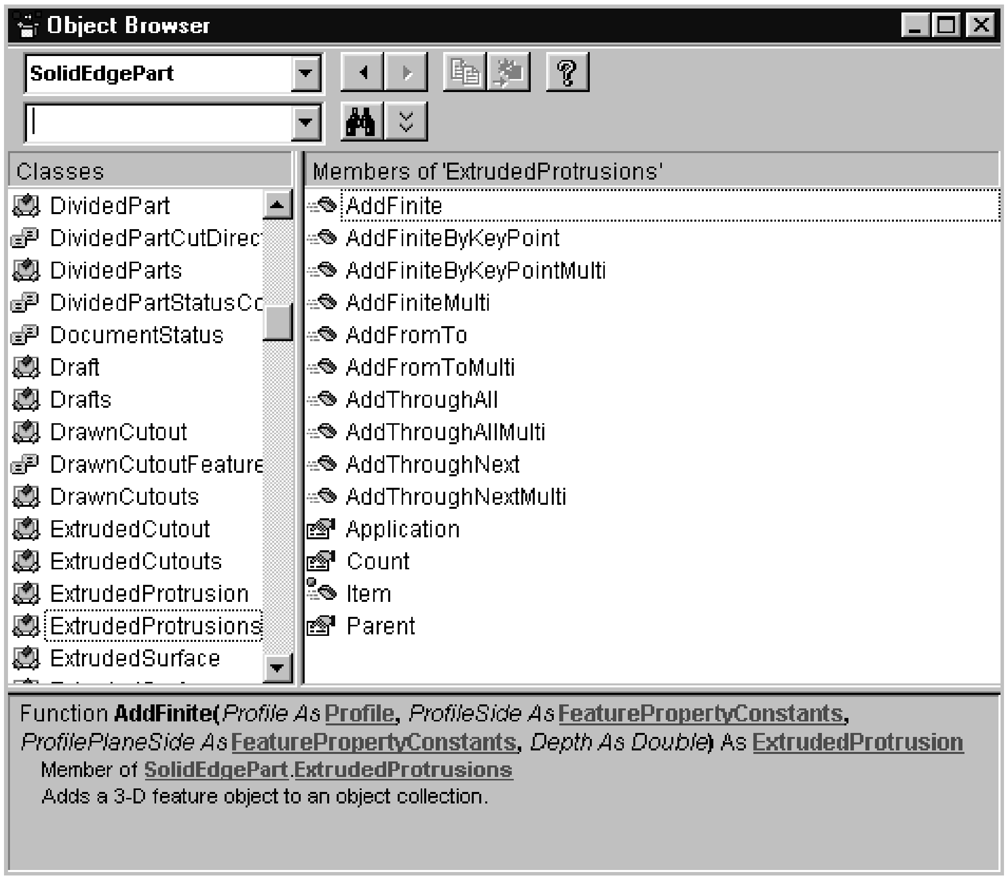
Enum **FeatureTopologyQueryTypeConstants**
Member of SolidEdgeConstants
Feature Topology Query Type Constants

The constant set shown in the illustration specifies the types of edges or faces to find when querying a model or feature. When a method or property has an argument that can be one of several values, a constant is usually provided. Using these constants rather than the corresponding number makes the code easier to read, and this is the recommended practice. Having these constants available through the Object Browser also provides a convenient interface for determining valid options for a property or method.

Members

The Members list shows all of the available methods, properties, and events for the selected object class. When you select an item from the Members list, the definition for the member is displayed in the Details pane at the bottom of the Object Browser. This definition includes jumps to the library and class to which the element belongs, and to constants and classes that are included in the element's definition. You can copy or drag text from the Details pane to the Code window.

For example, the following illustration shows the definition for the AddFinite method of the ExtrudedProtrusions class. The definition also provides jumps to the Profile, FeaturePropertyConstants, ExtrudedProtrusion, SolidEdgePart, and ExtrudedProtrusions class definitions.



Function **AddFinite(***Profile As* **Profile**, *ProfileSide As* **FeaturePropertyConstants**, *ProfilePlaneSide As* **FeaturePropertyConstants**, *Depth As Double***)** As **ExtrudedProtrusion**
Member of **SolidEdgePart**.**ExtrudedProtrusions**
Adds a 3-D feature object to an object collection.

The Object Browser visually differentiates among the types of members displayed in the Members list as follows:

Method—A member that performs some action on the object, such as saving it to disk.

Property—A member that establishes characteristics of the associated object.

Event—A notification that the object sends for an associated condition, such as when a file is opened or closed.

You can also differentiate among members by the appearance of the definition. The following are example definitions for methods and properties available for the Circle2d class.

- A property that declares the entire function to be of a specific type, in this case Double:

  ```
  Radius() As Double
  ```

- A method that takes a single value, called factor, as input:

  ```
  Scale(factor As Double)
  ```

- A method that takes three values (angle, x, and y) as input:

  ```
  Rotate(angle As Double, x As Double, y As Double)
  ```

Gathering Input from the User at Runtime

Visual Basic allows the programmer to interact with the end user through dialog boxes. The dialog boxes present a user interface that can be designed specifically for the task at hand. Using Visual Basic controls, such as text boxes and command buttons, the programmer can design a user interface that will communicate information to the program.

Gathering Input from a Data File

Visual Basic provides the ability to directly access many types of data files. The three types described here are text files, Microsoft Access database files, and Microsoft Excel spreadsheet files.

Text Files

One of the most common data files is a text file. A text file has the advantage of being readable through the basic text editor. Using Visual Basic file commands such as Open, Input, and Close, you can open a data file, read its contents, and then close the file. See the Visual Basic on-line Help for more information on manipulating text files.

Microsoft Access Database Files

Visual Basic has a built-in Jet Database Engine that allows the program to access and manipulate databases such as the one used by Microsoft Access. The Visual Basic on-line Help describes the Jet database engine as "...a database management system that retrieves data from and stores data in user and system databases. The Microsoft Jet database engine can be thought of as a data manager component with which other data access systems, such as Microsoft Access and Visual Basic, are built."

The Routing sample uses this built-in data manager to retrieve information from a Microsoft Access database. For more information about using the Jet database engine, refer to the Visual Basic on-line Help.

Microsoft Excel Spreadsheet Files

Microsoft Excel workbooks and worksheets can be accessed directly or through a Microsoft Jet database. This allows you to access the Excel spreadsheet without actually having Excel installed on the system. For details, refer to the Visual Basic on-line Help. An example spreadsheet that is delivered with Solid Edge, bearing.xls, shows how to work with Excel data. For a description of this program and for information on how to run it, see the *Modifying Graphics from Excel Data* section in Appendix B, Sample Programs.

Gathering Input from Solid Edge

Solid Edge supports COM Automation. This gives Visual Basic programs the ability to control Solid Edge through its automation interface. The samples provided with Solid Edge demonstrate ways in which the automation interface can be used.

Using the automation interface, a Visual Basic program can start and stop Solid Edge, draw, retrieve, create, modify, and delete graphical objects, manipulate document routing slips, and customize the user interface. You can also develop custom commands. Most Solid Edge functions can be controlled through COM Automation. For more information on objects exposed through COM Automation, see the Programming with Solid Edge on-line Help.

In addition to the automation interfaces, three custom controls are provided with Solid Edge: the command control, the mouse control, the part viewer control, and the draft viewer control. Using the mouse and command controls, Visual Basic applications can intercept Solid Edge events that are generated by moving the mouse, pressing any of the mouse buttons, or pressing any of the keys on the keyboard.

Gathering Input from Other Applications

A Visual Basic program can control and retrieve data from any application that supports COM Automation. Solid Edge delivers several samples that demonstrate using automation from other applications. For example, the Parametric Family of Parts sample (Custom\Bearing directory) demonstrates how COM Automation can be used to retrieve data from Microsoft Excel and then transfer it to Solid Edge. For more information, see Appendix B, *Sample Programs.*

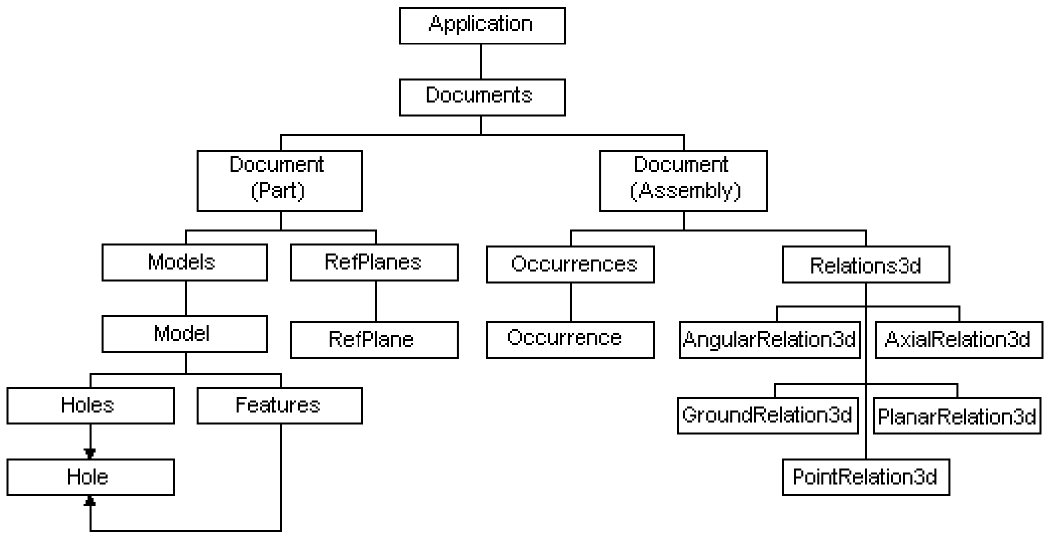
C H A P T E R 4

Invoking Solid Edge from Visual Basic

This chapter describes commands you can use to access Solid Edge from Visual Basic.

Invoking Solid Edge from Visual Basic—Overview

To program any application that supports COM automation from Visual Basic, you need to be able to communicate with that application. To make this possible, Solid Edge exposes the Application automation object, which is the root object that provides access from Visual Basic to all of the other objects in Solid Edge. There is one Solid Edge Application object, and there are five types of documents that the Solid Edge Application object can reference: part, sheet metal, weldment, assembly, and draft. The following illustration shows a simplified version of the object hierarchy of Solid Edge with two types of documents.



Syntax Examples

Two Visual Basic functions are available to invoke Solid Edge: CreateObject and GetObject. CreateObject creates a new instance of the Application object. GetObject allows you to create a new instance of an object or to connect to an existing instance.

For example, the following syntax uses CreateObject to launch Solid Edge:

```
Set objApp = CreateObject("SolidEdge.Application")
```

Similarly, GetObject can also launch Solid Edge:

```
Set objApp = GetObject("", "SolidEdge.Application")
```

Both CreateObject and GetObject create invisible instances of Solid Edge. New instances of Solid Edge created through the Application object can be made visible by setting the visible property of the Application object to True. The following syntax makes an application visible once it has been started:

```
objApp.Visible = True
```

The following syntax uses GetObject to connect to an existing instance of Solid Edge:

```
Set objApp = GetObject(, "SolidEdge.Application")
```

GetObject looks for an existing instance of Solid Edge. If an instance is found, the objApp variable points to it. The command fails if there is not an existing instance.

When starting an application using GetObject or CreateObject, the application does not automatically create a Document object. To create one, use the Add method of the Documents collection. If no arguments are passed to the Add method, you are prompted to select a template. Arguments can be used to specify the type of document and the template to be used. For example, you can add a Part document using the associated normal template as follows:

```
 Set objApp = GetObject(, "SolidEdge.Application")
 Set objDocument = objApp.Documents.Add("SolidEdge.PartDocument")
```

To remove an instance of Solid Edge from memory, use the Quit method of the Application object. The following statement removes an instance of Solid Edge:

```
objApp.Quit
```

Sample Program—Invoking Solid Edge from Visual Basic

The following sample code connects to a running instance of Solid Edge. If Solid Edge is not running, the Visual Basic program starts Solid Edge. Once Solid Edge is running, the Visual Basic program accesses the Application object, determines if any documents are open, and creates one if none exist. From the reference to the AssemblyDocument object, the program accesses the Occurrences collection.

```
'Declare the program variables.
Dim objApp As Object
Dim objDocs As Object
Dim objDoc As Object
Dim objOccurrences As Object

'Turn on error handling.
On Error Resume Next

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")
If Err Then
  'Clear the error.
  Err.Clear
  'Start Solid Edge.
  Set objApp = CreateObject("SolidEdge.Application")
End If

'Turn off error handling.
On Error GoTo 0

'Make the application window visible.
objApp.Visible = True

'Access the Documents collection.
Set objDocs = objApp.Documents

'Find out if any documents are open.
If objDocs.Count = 0 Then
  'Add an Assembly document.
  Set objDoc = objDocs.Add("SolidEdge.AssemblyDocument")
Else
  'Access the currently open document.
  Set objDoc = objApp.ActiveDocument
End If

'Check to make sure the active environment is Assembly.
If objApp.ActiveEnvironment <> "Assembly" Then
  MsgBox "This program must be run in the Assembly environment."
    End
End If

'Access the Occurrences collection.
Set objOccurrences = objDoc.Occurrences
```

In this example, variables are defined for the Application object (objApp), the Documents collection (objDocs), the Document object (objDoc), and the Occurrences collection (objOccurrences). First, GetObject connects to Solid Edge. GetObject is successful only if Solid Edge is already running. If this fails, CreateObject creates a new instance of Solid Edge.

The next step is to access a Document object. If there is not already an open document, one is created using the Add method on the Document object. If there is already a document open in the application, it is referenced using the Application's ActiveDocument property. Finally, the Occurrences collection is referenced from the Document object.

COM Interoperability Between Solid Edge and .NET

Generating Interop Assemblies

To achieve COM interoperability with Solid Edge, you must create a set of interop assemblies from the Solid Edge type libraries in one of two ways:

- Import a reference to the type libraries using the VS.NET IDE's Add Reference command.
- Use the tlbimp.exe tool provided with VS.NET.

When you use tlbimp.exe to generate interop assemblies, you should use the /safearray qualifier, because Solid Edge APIs work with non-zero-based SAFEARRAYs. The /safearray qualifier is automatically used when you import type libraries using the VS.NET IDE. For more information on tlbimp.exe, see the Microsoft .NET documentation.

Deploying Interop Assemblies

Until Solid Edge generates strongly-named assemblies and delivers them in the GAC, .NET clients should reference their own generated interop assemblies and deploy them as private assemblies for the application.

Using the Solid Edge Interop Assemblies

Once the interop assemblies are generated, .NET clients can code calls into Solid Edge using the classes and interfaces contained in the assemblies. The .NET framework also provides useful classes, interfaces and APIs.

Visual Basic 6.0 Compatibility

There is a set of Visual Basic 6.0 compatibility assemblies that can be referenced. For example, if a VB 6.0 application that uses GetObject (SolidEdge.Application) needs to be ported to VB.NET, the Microsoft.VisualBasic.Compatibility.VB6 assembly can be referenced.

System Interop Services

For all .NET developers, .NET provides the System.Runtime.InteropServices namespace. One of the most important objects in that namespace is the Marshal object. Using Marshal.GetActiveObject ("SolidEdge.Application"), a caller can obtain an instance of the Solid Edge application from the running object table.

Since the Solid Edge type libraries do not currently contain a co-class for the application, .NET users can resort to using the System.Type and System.Activator classes.

For more information on the Marshal object, see the Microsoft .NET documentation.

Using Solid Edge Events

Many objects in Solid Edge provide event sets. Many of the event sets are obtained from objects returned as properties from the APIs. For example, in order to use the application event set, the Application.ApplicationEvents property is used.

When you examine the Solid Edge Framework interop assembly, you will see that the type returned from the ApplicationEvents property is a generic object. This is different from what a user sees when referencing the type library from VB 6.0 or importing the type library in C++. In those cases, the type returned is a co-class that supports the event set. This difference does not mean that the object returned does not support the event set. However, connecting to the event set is slightly different using the interop assembly.

In order to connect to the event set, the interface for the event set must be obtained from the returned object. You can do this by declaring the correct event interface and casting the returned object to that interface. So, for example, in order to connect to the application events event set, you should use the ApplicationEvents_Event interface.

Sample Program

For C# programmers, obtaining Solid Edge and connecting to the application events would look something like this:

```
 Object TheEdge;
 TheEdge = Marshal.GetActiveObject("SolidEdge.Application");
 TheEdgeApplicationInterface =
(interop.EdgeFrameworkLib.Application)TheEdge;
 object oAppEvents = TheEdgeApplicationInterface.ApplicationEvents;
 interop.EdgeFrameworkLib.ISEApplicationEvents_Event
appEvents_Event;
                        appEvents_Event =
(interop.EdgeFrameworkLib.ISEApplicationEvents_Event)oAppEvents;
 appEvents_Event.AfterDocumentSave += new
interop.EdgeFrameworkLib.ISEApplicationEvents_AfterDocumentSaveEven
tHandler(AfterDocumentSave);
 .
 .
 .
 public void AfterDocumentSave( Object TheDocument )
 {
 }
```

For VB.NET programmers accustomed to using the "WithEvents" qualifier when declaring an ApplicationEvents object type, the interop assembly's ApplicationEvents_Event interface is the type that should be declared.

```
 Imports interop.SolidEdgeFrameworkLib
 Imports System.Runtime.InteropServices
 Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim WithEvents EdgeAppEvents As
interop.SolidEdgeFrameworkLib.ISEApplicationEvents_Event
 #Region " Windows Form Designer generated code "
    Public Sub New()
        MyBase.New()
        'This call is required by the Windows Form Designer.
        InitializeComponent()
        'Add any initialization after the InitializeComponent()
call
    End Sub
    'Form overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As
Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub
    'Required by the Windows Form Designer
    Private components As System.ComponentModel.Icontainer
    'NOTE: The following procedure is required by the Windows Form
Designer
   'It can be modified using the Windows Form Designer.
    'Do not modify it using the code editor.
    Friend WithEvents OpenFileDialog1 As
System.Windows.Forms.OpenFileDialog
    Friend WithEvents FileOpenButton As System.Windows.Forms.Button
    <System.Diagnostics.DebuggerStepThrough()> Private Sub
 InitializeComponent()
        Me.OpenFileDialog1 = New
System.Windows.Forms.OpenFileDialog()
        Me.FileOpenButton = New System.Windows.Forms.Button()
        Me.SuspendLayout()
        '
        'FileOpenButton
        '
        Me.FileOpenButton.Location = New System.Drawing.Point(16,
24)
        Me.FileOpenButton.Name = "FileOpenButton"
        Me.FileOpenButton.TabIndex = 0
        Me.FileOpenButton.Text = "Open File"
        '
        'Form1
        '
        Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
        Me.ClientSize = New System.Drawing.Size(292, 266)
        Me.Controls.AddRange(New System.Windows.Forms.Control()
 {Me.FileOpenButton})
        Me.Name = "Form1"
        Me.Text = "Form1"
        Me.ResumeLayout(False)
    End Sub
 #End Region
    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e
As
 System.EventArgs) Handles MyBase.Load
        Dim TheEdge As interop.SolidEdgeFrameworkLib.Application
        TheEdge = Marshal.GetActiveObject("SolidEdge.Application")
        EdgeAppEvents = TheEdge.ApplicationEvents
    End Sub
    Private Sub FileOpenButton_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles FileOpenButton.Click
    End Sub
    Private Sub EdgeAppEvents_AfterDocumentOpen(ByVal theDocument
As Object) Handles EdgeAppEvents.AfterDocumentOpen
        MsgBox("Received AfterDocumentOpen event")
    End Sub
 End Class
```

C H A P T E R 5

Invoking Solid Edge from Visual C++

This chapter describes how to access Solid Edge from Visual C++.

Invoking Solid Edge from Visual C++—Overview

To create programs with Visual C++, you must be able to communicate with that application. The Solid Edge Application automation object is the root object that provides access to all of the other Solid Edge objects. The implied hierarchy for automation is the same whether you are programming with Visual C++ or Visual Basic.

When programming Solid Edge with Visual C++, you will import entire type libraries into the client code. Visual C++ automatically creates .TLI and .TLH files from the imported type libraries. The .TLH files define the smart pointers for every interface (both v-table and dispatch interfaces) defined in the associated type library using the _COM_SMARTPTR_TYPEDEF macro. If there is an interface of type "ISample" in the typelib, the smart pointer associated with that is named "ISamplePtr" (by appending "Ptr" to the interface name).

The smart-pointer implementation encapsulates the real COM interface pointers and eliminates the need to call the AddRef, Release, and QueryInterface functions that all interfaces support. Additionally, the CoCreateInstance call is hidden when creating a new COM object. Because these smart pointers also know the UUID for the interface they are wrapping, simply equating two smart pointers will call QueryInterface on the interface on the right-hand side for the UUID of the interface supported by the smart pointer on the left-hand side (much like Visual Basic).

For more information on COM smart pointers, see the topics "Compiler COM Support Classes" and "Compiler COM Support: Overview" in the MSDN Library. Useful information is also included in the comdef.h file delivered with Visual Studio.

Structuring Visual C++ Programs

The following information applies to setting up a Visual C++ program to run with Solid Edge:

- Be sure that the following directories are in your include path:
  - The directory containing the Solid Edge type libraries
  - The directory containing the Visual Studio includes
- Keep the #import statements in the standard pre-compiled header so that all the .CPP files automatically have access to the smart pointers generated from the type libraries.
- Smart pointers handle error returns by converting error HRESULTs into "_com_error" exceptions. The "_com_error" class encapsulates the HRESULT error code. Because these objects generate exceptions, you will need an exception handler in your code. However, if you use the "raw" versions of the interface functions that are returned, you can avoid exceptions, and deal with regular HRESULTs instead.
- The compiler support implementation converts properties into Get/Put pairs. But the property is also usable directly, as in Visual Basic. For example, the "Visible" property on the Application object is usable in the following ways:

```
ApplicationPtr pApp;   ... [get the app pointer] ...   pApp-
>Visible = VARIANT_TRUE; // this is VB-like syntax pApp-
>PutVisible(VARIANT_TRUE); // this is the equivalent C++
like syntax
```

However, methods are called as usual, such as "pApp->Activate()".

Solid Edge creates self-contained type libraries. That is, they are self-contained with respect to the constants that are used by the objects within that type library. This allows you to browse for constants used in a specific type library within that same type library, without having to browse a different type library. However, when you use the compiler support #import of type libraries, you must explicitly qualify the constant as coming from a specific type library (because more than one may have the same constant). In most such cases, you must qualify the constant to the type library where the method/property/object resides, because that is how the compiler expects it to be declared. If that does not work, qualify it to SolidEdgeConstants, which contains *all* constants.

Unfortunately, parameters of type SAFEARRAY do not have compiler support classes, unlike VARIANT, whose corresponding compiler support class is "_variant_t"; or BSTR, whose corresponding class is "_bstr_t." Therefore, you must manage SafeArrays using the various SafeArray APIs that Visual C++ provides to manage the creation/manipulation/deletion of SafeArrays.

Interfaces that start with an underbar (for example, "_<some_interface>") are v-table interfaces that support the corresponding dispatch versions. Although these do show up in the type library and "#import" generates smart pointers for them, clients *must not* use these in their code.

The v-table interfaces that do not have an "underbar" prefix can be used. For example,

- Do not use "_IApplicationAutoPtr", but use "ApplicationPtr"
- Do not use "_IPartDocumentAutoPtr", but use "PartDocumentPtr"
- Do not use "_IDMDBodyPtr", but use "BodyPtr"
- Do use "ISEDocumentEventsPtr"
- Do use "ISEMousePtr"

Be careful when you mix smart pointers and non-smart pointers (that is, straight COM interfaces). You must be aware of the AddRefs and Releases going on in the background, and you may have to manually insert code to be COM compliant.

C H A P T E R 6

Using the SEInstallData Library

This chapter describes how to use the SEInstallData library delivered with Solid Edge.

Using the SEInstallData Library

The file SEInstallData.dll contains the SEInstallData object. This library is delivered during Solid Edge setup, but can be used independently of Solid Edge. Use it to obtain information about the Solid Edge installation, including major and minor version numbers, major and minor Parasolid version numbers, installation path, and language ID.

Sample Program—Using the SEInstallData Library

This program declares objINS as an SEInstallData object and creates this object. If the creation is successful, the program accesses the properties that describe the Solid Edge version installed on the client system, and accesses the property describing where Solid Edge is installed on the client system.

```
Dim objINS As SEInstallData
Dim strVersion As String
Dim nMajorVersion As Long
Dim nMinorVersion As Long
Dim nServicePackVersion As Long
Dim nBuildNumber As Long
Dim strInstallPath As String

Set objINS = CreateObject ("SolidEdge.InstallData")
If objINS is Nothing Then
MsgBox ("Failed to find SolidEdge.InstallData")
Else
strVersion = objINS.GetVersion
nMajorVersion = objINS.GetMajorVersion
nMinorVersion = objINS.GetMinorVersion
nServicePackVersion = objINS.GetServicePackVersion
nBuildNumber = objINS.GetBuildNumber
strInstallPath = objINS.GetInstalledPath
End If
```

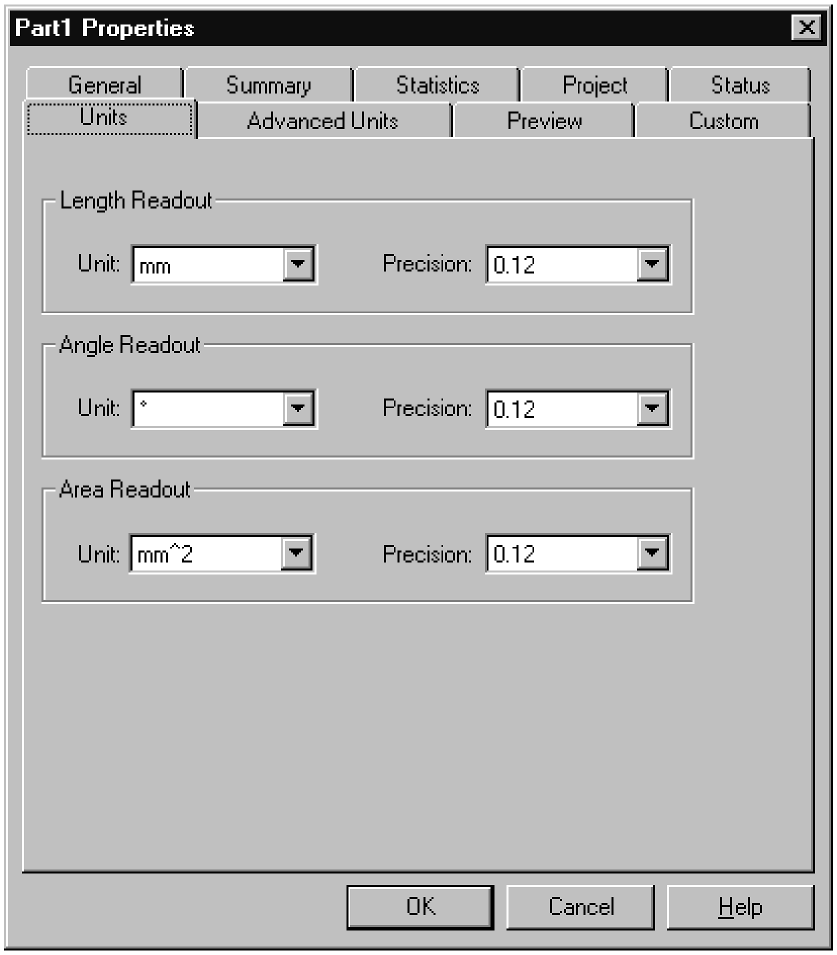
C H A P T E R 7

Working with Units of Measure

This chapter describes how to work with units of measurement when customizing Solid Edge.

Working with Units of Measure—Overview

In the interactive environment, Solid Edge allows you to specify the length, angle, and area units to use when placing, modifying, and measuring geometry. For example, you can specify millimeters as the default length unit of measurement; you can also specify the degree of precision of the readout. You specify these properties on the Units and Advanced Units tabs of the Properties dialog box. (On the File menu, click Properties to display the dialog box.)



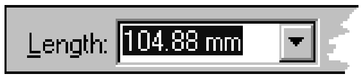
Length Readout
Unit: mm
Precision: 0.12

Angle Readout
Unit: °
Precision: 0.12

Area Readout
Unit: mm^2
Precision: 0.12

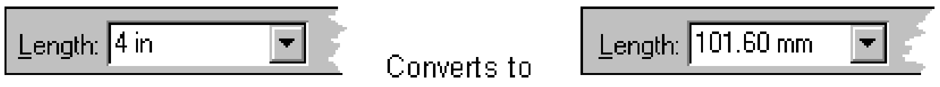
This is strictly a manipulation of the display of the precision; internally all measurements are stored at their full precision.

With a Length Readout precision of 0.12, the length of any linear measurement is displayed as follows:



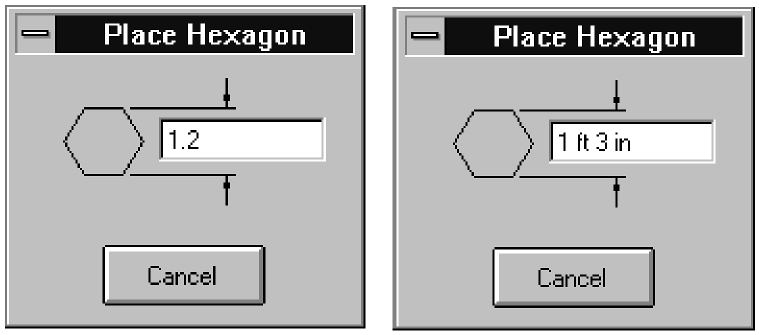
Length: 104.88 mm

Because millimeters are the default units in this example, whenever distance units are entered, they have to be in millimeters. If a user enters a distance value in inches, for example, the units are automatically converted to millimeters.



The units system in Solid Edge allows users to specify the default units and to control how values are displayed for each of the units. Users can change the default units and their display at any time and as often as necessary.

You can customize Solid Edge so that your commands behave in a similar way. For example, suppose you are creating a program to place hexagons. The program displays a dialog box that allows you to enter the size of the hexagon and then creates the hexagon at a location specified by a mouse click. When users enter the size of the hexagon, they should be able to enter the value in the user-specified default unit. Also, users should be able to override the default unit and specify any linear unit. The program will need to handle any valid unit input.



In this example, the dialog box on the left assumes that the key-in is in the user-defined default unit. The dialog box on the right shows how the user can override the default and specify any unit. The automation for units allows you to easily handle either inputs.

Internal Units

The following internal units are used:

Unit Type	**Internal Units**
Distance	Meter
Angle	Radian
Mass	Kilogram
Time	Second
Temperature	Kelvin
Charge	Ampere
Luminous Intensity	Candela

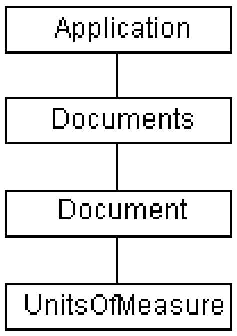
Amount of Substance	Mole
Solid Angle	Steradian

All other units are derived from these. All calculations and geometry placements use these internal units. When values are displayed to the user, the value is converted from the internal unit to the user-specified unit.

When automating Solid Edge, first convert user input to internal units. Calculations and geometric placements use the internal units. When displaying units, you must convert from internal units to default units. The UnitsOfMeasure object handles these conversions.

Object Hierarchy

The hierarchical diagram for UnitsofMeasure is as follows:



UnitsOfMeasure Methods

The UnitsofMeasure object provides two methods: ParseUnit and FormatUnit. In addition, a set of constants is provided to use as arguments in the methods. The following syntax shows how to access the UnitsofMeasure object from the application:

```
Dim objApp as Object
Dim objUOM as object

Set objApp = GetObject(,"SolidEdge.Application")
Set objUOM = objApp.ActiveDocument.UnitsOfMeasure
```

The ParseUnit method uses any valid unit string to return the corresponding database units. The FormatUnit method uses a value in database units to return a string in the user-specified unit type, such as igUnitDistance, igUnitAngle, and so forth. The units (meters, inches, and so forth) and precision are controlled by the active units for the document.

Sample Program—Formatting and Displaying Units

The following example uses both the ParseUnit and FormatUnit methods to duplicate the behavior of unit fields in Solid Edge. The code could be used in association with the LostFocus event of a text box control. In this example, the text box is named TxtSize.

This code checks whether the input is a valid unit key-in and replaces it with a correctly formatted string according to the user-specified setting.

Error handling is used to determine if a valid unit has been entered. The Text property from the text field is used as input to the ParseUnit method, and the unit is a distance unit. If the ParseUnit method generates an error, focus is returned to the text field, and an error is displayed, giving the user a chance to correct the input. This cycle continues until the user enters a correct unit value. If the key-in is valid, then the database value is converted into a unit string and displayed in the text field.

```
'Declare the program variables.
Dim ObjApp As Object
Dim objUOM As Object

Private Sub Form_Load()
'Connect to a running instance of Solid Edge.
Set ObjApp = GetObject(, "SolidEdge.Application")

'Access the UnitsOfMeasure object.
Set objUOM = ObjApp.ActiveDocument.UnitsOfMeasure
End Sub

Private Sub Text1_LostFocus()
Dim HexSize As Double

'Turn on error handling.
On Error Resume Next
HexSize = objUOM.ParseUnit(igUnitDistance, Text1.Text)
If Err Then
  'Set focus back to text field.
  Text1.SetFocus
  'Display error.
  MsgBox "Invalid unit key-in."
End If
'Turn off error handling.
On Error GoTo 0

'Assign correct text to the control's text property.
Text1.Text = objUOM.FormatUnit(igUnitDistance, HexSize)
End Sub
```

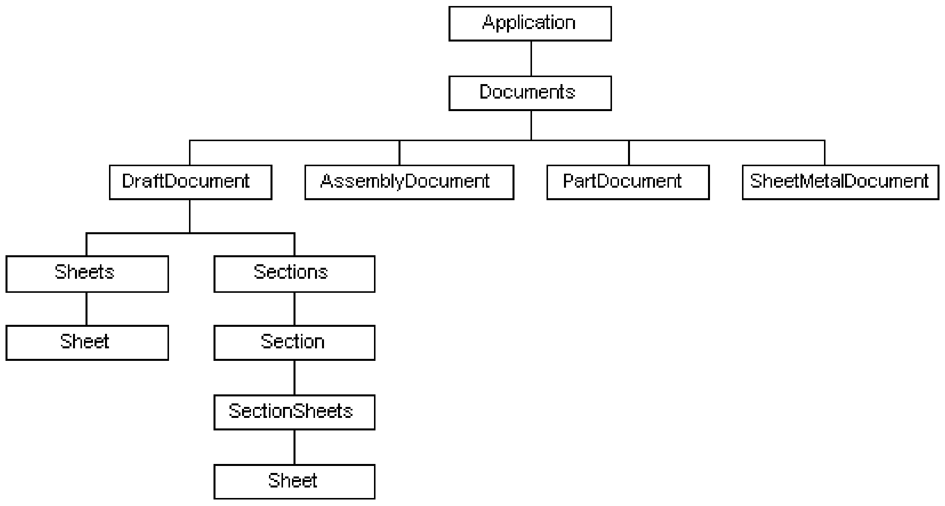
C H A P T E R 8

Working with Documents

This chapter contains a description of documents, and how to access them through COM automation.

Working with Documents—Overview

The term *document* is a standard object name throughout various applications. "Documents" refers to a collection of Document objects that have been opened by the application. The following illustration shows the Document object and how it relates to the rest of the application:



An application can have only one Documents collection, but the collection can contain any number of document objects. In Solid Edge, these document objects can represent any Assembly, Draft, Sheet Metal, Weldment, and Part documents that are currently open in the application. Many properties and methods are common across all document types; other properties and methods are specific to a document type. Part, Sheet Metal, and Weldment documents share a common automation interface.

Each type of Solid Edge document object has its own methods and properties, along with several that are common across all types of documents. For more information on the properties and methods of the documents collection and document objects, see the AssemblyDocument, DraftDocument, SheetMetalDocument, WeldmentDocument, and PartDocument objects in the Programming with Solid Edge on-line Help.

Working with Different Solid Edge Document Types

The following lists the classes (also called ProgIDs) associated with Solid Edge and each type of document. These class names are used in the Visual Basic functions CreateObject and GetObject. They are also used in the Add method of the Document object.

- Solid Edge Application—SolidEdge.Application
- Part Document—SolidEdge.PartDocument
- Sheet Metal Document—SolidEdge.SheetMetalDocument
- Assembly Document—SolidEdge.AssemblyDocument
- Weldment Document—SolidEdge.WeldmentDocument
- Draft Document—SolidEdge.DraftDocument

The following syntax starts Solid Edge and creates a Part document, a Sheet Metal document, an Assembly document, and a Draft document:

```
Set objApp = CreateObject("SolidEdge.Application")
objApp.Visible = True
Set objDocs = objApp.Documents
objDocs.Add("SolidEdge.PartDocument")
objDocs.Add("SolidEdge.SheetMetalDocument")
objDocs.Add("SolidEdge.AssemblyDocument")
objDocs.Add("SolidEdge.DraftDocument")
```

The following syntax starts Solid Edge, creating an Assembly document, and returns an AssemblyDocument object. The Application object is then retrieved using the Application property of the document object:

```
objDoc = CreateObject("SolidEdge.AssemblyDocument")
objApp = objDoc.Application
```

Sample Program—Opening and Printing a Document

The following program opens a Part document, prints the document, and then closes it.

```
 'Declare the program variables.
 Dim objApp As Object
 Dim objDocs As Object
 Dim objDoc As Object
 Dim IntNumCopies As Integer
 Dim Orientation As PrinterObjectConstants
 Dim PaperSize As PrinterObjectConstants

 'Connect to a running instance of Solid Edge.
 Set objApp = GetObject(, "SolidEdge.Application")

 'Access the Documents collection object.
 Set objDocs = objApp.Documents

 'Open an existing document.
 Set objDoc = objDocs.Open("c:\My Documents\Drawing
Files\block.par")

 'Print two copies of the document on the default paper, using
 'letter size paper with a landscape orientation.
 IntNumCopies = 2
 Orientation = vbPRORLandscape
 PaperSize = vbPRPSLetter
 Call objDoc.PrintOut _
   (NumCopies:=IntNumCopies, _
   Orientation:=Orientation, _
   PaperSize:=PaperSize)

 'Close the document.
 Call objDoc.Close
```

CHAPTER 9

Working with Part and Sheet Metal Documents

This chapter describes how to use Part and Sheet Metal documents.

Part Document Anatomy

The Models Collection

The PartDocument and SheetMetalDocument objects support a Models collection. A Model contains a set of Features that make up a single solid (or non-overlapping solid regions—a disjoint solid).

Reference Planes

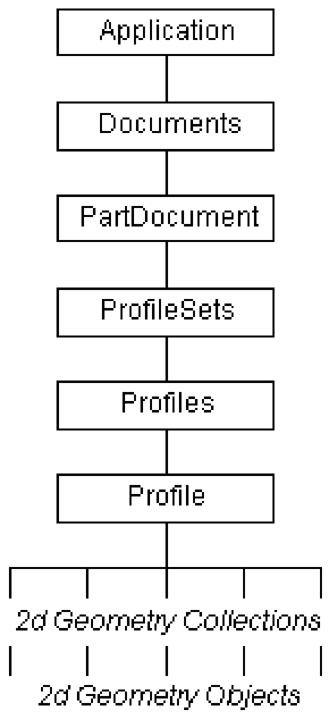
When you model a part, a reference plane (the RefPlane object) must exist before you can create a profile. The corresponding collection object, RefPlanes, provides several methods to enable you to place reference planes. These methods roughly correspond to the reference plane commands that are available in the interactive environment.

- AddAngularByAngle—Creates angular and perpendicular reference planes. The perpendicular reference plane is a special case of the angular reference plane where the angle is pi/2 radians (90 degrees).
- AddNormalToCurve and AddNormalToCurveAtDistance—Create reference planes that are normal to a part edge. With AddNormalToCurve, if the edge is a closed curve, the plane is placed at the curve's start point. AddNormalToCurveAtDistance places the plane at a specified offset from the start point.
- AddParallelByDistance—Creates coincident and parallel reference planes. A coincident reference plane is a parallel reference plane where the offset value is zero.
- AddParallelByTangent—Creates parallel reference planes that are tangent to a curve.
- AddBy3Points—Creates reference planes associative to three points you specify.

Profiles

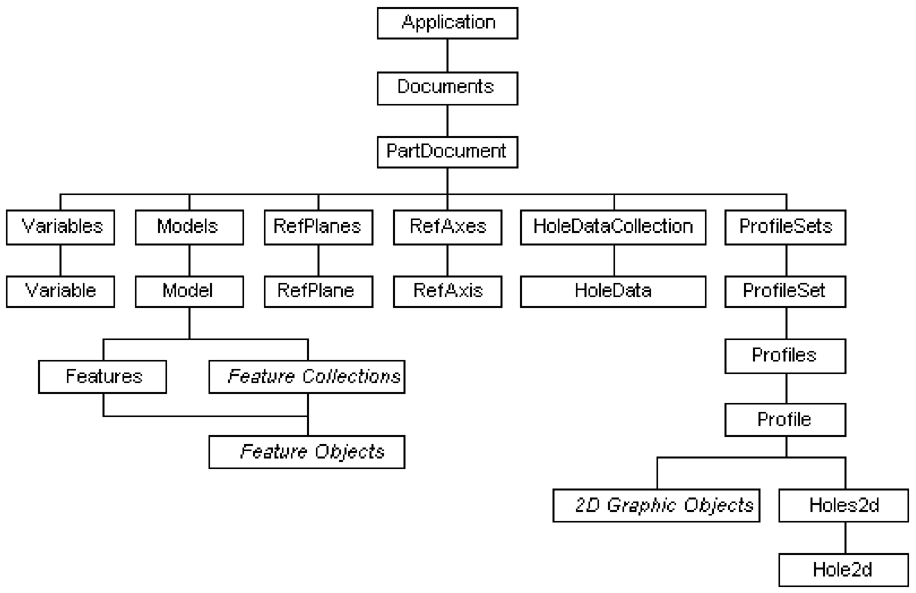
With many types of features, one of the first steps in the construction process is to draw a two-dimensional profile. It is the projection of this profile through a third dimension that defines the shape of the feature.

The workflow for modeling a feature through automation is the same as the workflow in the interactive environment. For profile-dependent features, you draw the profile and then project or revolve it. In the automation environment, the profile is a required input to the add method for certain types of features. In addition, profile automation includes the ability to create, query, and modify profiles. The object hierarchy for the Profile object is as follows:


Application
Documents
PartDocument
ProfileSets
Profiles
Profile
2d Geometry Collections
2d Geometry Objects

Working with the Part Object Hierarchy

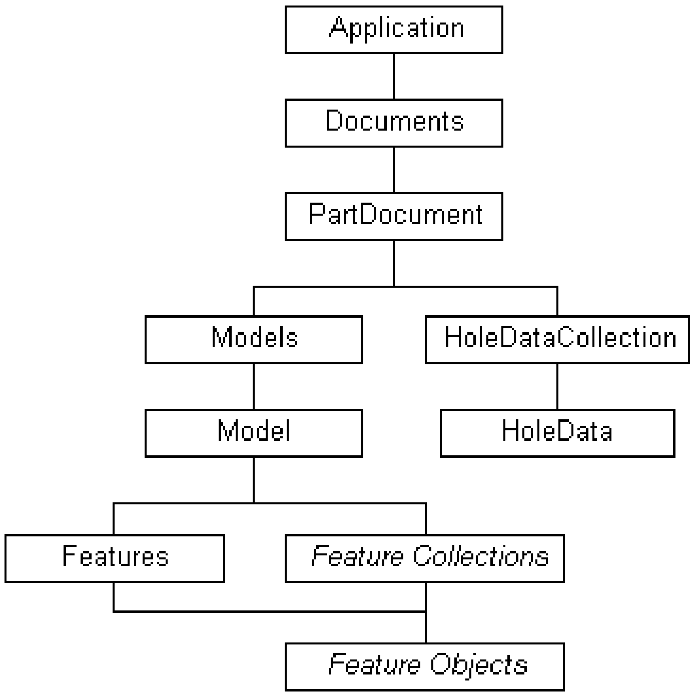
A simplified object hierarchy for the Solid Edge Part environment is as follows:



The PartDocument supports a Models collection. A Model is a group of graphics. In Solid Edge, a Model consists of a set of Features that make up a single solid (which *may* consist of nonoverlapping solid regions, a disjoint solid). In addition to the objects shown in this hierarchy diagram, the PartDocument object supports the following methods/properties: AttachedPropertyTables, AttributeQuery, Constructions, CoordinateSystems, DocumentEvents, FamilyMembers, HighlightSets, Properties, PropertyTableDefinitions, RoutingSlip, SelectSet, Sketches, SummaryInfo, UnitsOfMeasure, and Windows, among others.

Creating a Base Feature

When you create a model interactively, you always begin by creating a *base feature*. You then add subsequent features to this base feature to completely define the model. When you create a model using automation, the workflow is identical. Using add methods on the Models collection, you first create a base feature commonly using either an extruded or revolved protrusion. The simplified hierarchical diagram is as follows:



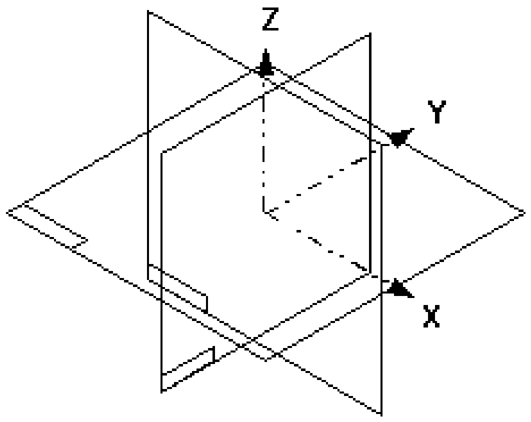
The Model Object

Using the Add method on the Models collection creates a Model object. Use an Add method on the Models collection once to create the base feature, and then use the Add method on the Features collection to create subsequent features. The Model object acts as the parent of the features that define the part. You access the individual features through the Model object.

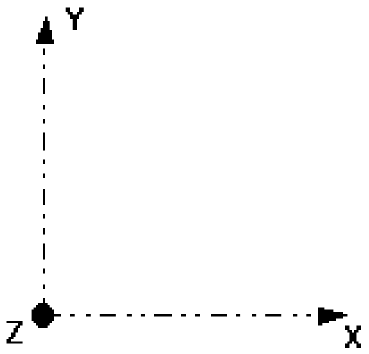
Understanding the Modeling Coordinate System

When you work interactively in Solid Edge, there is no need to be aware of a coordinate system. This is because you create profiles and features relative to the initial reference planes and existing geometry. When modeling non-interactively, however, it is often easier to identify specific locations in space to position profiles and features rather than to define relationships to existing geometry. Understanding the coordinate system is necessary to correctly place and orient profiles and features.

Solid Edge uses the Cartesian coordinate system. The units used when expressing coordinates in the system are always meters. The following illustrations show the Solid Edge coordinate system. As viewed from standard isometric view, the coordinate system is as follows:



As viewed from a top view, positive x is to the right, positive y is up, and positive z is pointing toward you. The origin of the coordinate system is at the intersection of the three base reference planes as follows:



Creating Profiles

A profile consists of one or more wireframe elements. If the profile consists of more than one element, the elements must be end-connected. You do this by adding relations between the elements' endpoints.

Creating a Profile Through Automation

To create a profile in the automation environment, follow these steps:

1. Create an empty Profile object.

2. Place geometry to define the shape of the profile. The collections that support the add methods are on the Profile object.

3. Place relationships on the geometry. You can use any of the add methods supported by the Relations2d collection object. The only required relations are key point relations between the endpoints of elements that you intend to connect.

4. Place required dimensions on the geometry.

5. Use the End method to validate the profile. Depending on the validation criteria specified, the system checks to verify that the profile is valid. After validation, a profile is available that you can use it as input to the feature add methods.

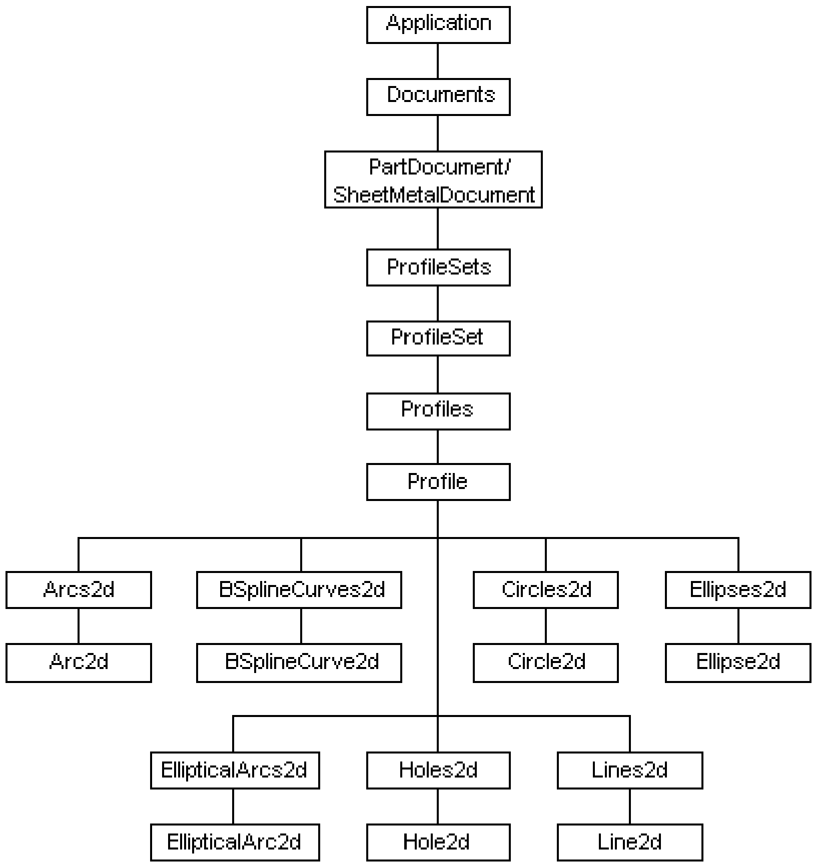
Creating Multiple Disjoint Solid Regions

When interactively creating an extruded protrusion base feature (that is, the first feature in the feature tree), you are not limited to using a single closed shape; you can use multiple closed shapes. All the shapes are extruded using the same extents to create a single extruded protrusion feature. The result is several disjoint solid regions.

You can also perform this function through automation. When drawing the geometry in the blank Profile object, you can create multiple closed shapes. The End method evaluates the geometry, and if there are multiple shapes, it breaks them up so that there is a single closed shape per Profile object. This set of Profile objects is contained in a single Profiles collection object. Each Profiles collection is owned by a ProfileSet object. All the profiles that define a profile-based feature, which in most cases is a single profile, are contained in a single ProfileSet.

Working with 2D Geometry

The Solid Edge automation model allows you to place many different two-dimensional geometry objects. Through automation, you can place and manipulate objects such as arcs, b-spline curves, circles, ellipses, elliptical arcs, hole centers, and lines. The object hierarchy for 2-D geometry is as follows:



To create a 2-D geometry object, first access a Profile object. The Profile object owns the 2-D graphic object collections, and it is through add methods on these collections that you create geometry. For example, to create a line, you could use the AddBy2Points method, which is available through the Lines2d collection.

Note The Parent property of the 2-D geometry object is the Profile object, not the collection. The collection provides a way to create objects and iterate through them.

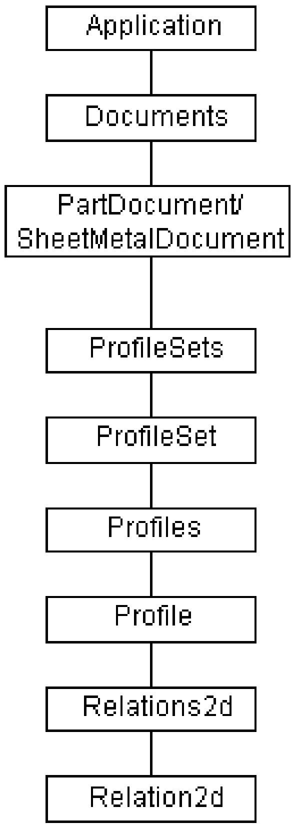
When a 2-D geometry object is created, it is assigned a name. This name, which is stored in the Name property, consists of two parts: the object type (such as Line2d, Arc2d, or Circle2d) and a unique integer. Each object's name is therefore unique, and because it never changes, you can always use the Name property to reference a 2-D graphic object.

Solid Edge allows you to establish and maintain relationships on the 2-D elements that you draw in the Profile environment. These relationships control the size, shape, and position of an element in relation to other elements.

Working with 2D Relationships

When an element changes, it is the relationships that drive the update of related elements. For example, if you have drawn a polygon with two lines parallel to one another and you modify the polygon, the two lines remain parallel. You can also change elements that have dimensions. If a driving dimension measures the radius of an arc, you can edit the value of the dimension to change the radius of the arc.

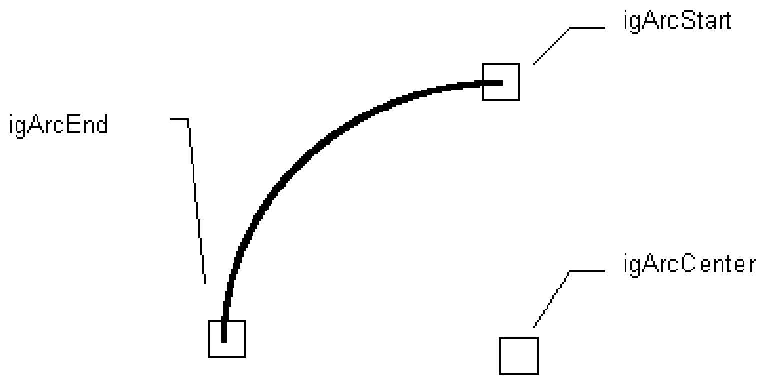
The object hierarchy for relationships is as follows:



The Relations2d object provides the methods for placing relationships and for iterating through all the relationships that exist on the associated profile. To establish a relationship between elements, use an add method on the Relations2d collection on the profile.

The objects for which you want to define the relationship must already exist on the profile. There is an add method on the Relation2d object for each type of relationship that can be defined. Once a relationship is defined, properties and methods on the Relation2d object allow you to edit existing relationships or query for information about relationships.

Many relationships are placed at a specific location on an element. For example, when you place a key point relationship to link the endpoints of two lines, you select one end of each line. The programming interface for placing relationships provides the ability to select specific points on an element by using predefined key points for each geometrical element. The key point to use is specified by using key point indexes. The following illustration shows the key points for an Arc element:



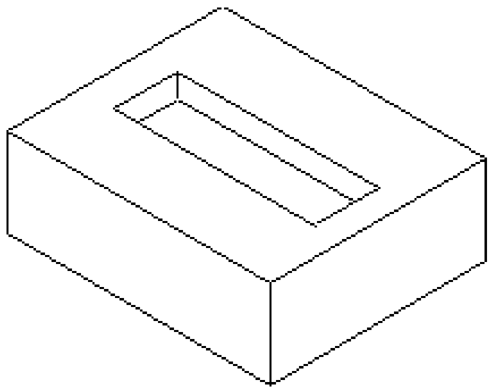
For more information on the methods to place, edit, and query relationships, see the Programming with Solid Edge on-line Help. For a complete list of defined types for different relationships and key point index constants for different elements, see the list of constants for ObjectType and KeypointIndexConstants available in the Solid Edge Constants type library using the Visual Basic Object Browser.

Creating Solids

In Solid Edge, all geometry is created relative to existing geometry in the file. For example, when you create a base protrusion feature, the feature depends on the profile, and the profile depends on the reference plane. Relationships between these geometry elements are referred to as parent/child relationships. In this case, the reference plane is the parent of the profile, and the profile is the parent of the protrusion feature.

If a parent is modified in any way, its children are automatically updated to maintain the correct relationships. While this parent/child relationship is inherently a part of designing in the interactive environment, it is even more apparent when you create features and solids through the automation interface.

To better understand the process of creating solid geometry using automation, compare the steps required to create the following cutout feature interactively with the steps to create it through automation:



Creating the Cutout Interactively

1. Define the profile plane by selecting a face and defining an edge and origin.
2. Create the profile by sketching wireframe geometry and defining geometric and dimensional relationships.
3. Define the side of the profile from which material is to be removed.
4. Define the extents of the feature.

Creating the Cutout Through Automation

The steps for creating this feature through automation are similar to those used to create it interactively. However, many elements of feature creation in the interactive environment are automatically inferred or are trivial for the user to define. You must directly address these elements when creating features through automation.

5. Determine the top face of the solid where the profile is to be created.

6. Determine the appropriate edges of the face to use to control the orientation of the reference plane.

7. Create a reference plane on the face of the solid.

8. Create an empty profile object.

9. Determine the location in 3-D space where the profile elements are to be created, and compute the corresponding location in the 2-D reference plane or profile space.

10. Create the 2-D wireframe geometry, and constrain it as needed to define the shape of the profile.

11. Complete the profile by performing validation to determine whether it meets the expected criteria.

12. Determine the side of the profile from which material is to be removed.

13. Determine the extent of the cutout. In this case, the cutout is finite, so the direction of the cutout needs to be determined. In other types of extents, this can also involve finding faces of the solid to specify the feature extents.

14. Create the cutout feature.

Although the automation workflow is similar to the interactive workflow, creating a feature through automation is not trivial. Fundamentally, creating a feature depends on providing the parent geometry and values for the feature to reference. In the previous example, Steps 1 through 9 create the parent geometry and determine input values for the feature. The feature is created in Step 10. Creating and collecting the parents of the feature is the most difficult part of defining a feature through automation.

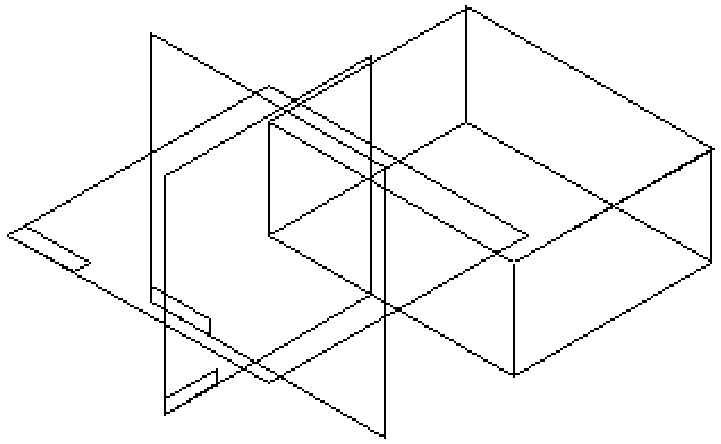
The functions needed to perform these steps can be broken into several categories.

- Querying the solid, its faces, and its edges.
- Accessing information from the solid.
- Creating reference planes.
- Creating profiles.
- Placing and constraining 2-D geometry to define the shape of the profile.
- Determining sides and directions for input to the feature functions.
- Creating the feature.

Sample Program—Creating a Feature

The following program creates a cutout feature. This program uses many separate components of the automation model to form a complete workflow for creating a feature. The following program components are used: querying, working with reference planes, working with profiles, and placing features.

This program assumes that a base feature already exists in the file. The base feature is a rectangular block, 6x5x2 inches and oriented as follows:



The corner of the block is at the intersection of the three base reference planes. This places the corner of the block at the coordinate (0,0,0), the 6-inch dimension in the positive x direction, the 5-inch dimension in the positive y direction, and the 2-inch dimension in the positive z direction. The cutout to be created is 4 x 1½ inches and ½ inch deep. It is centered on the top face of the block.

```
Private Sub Command1_Click()

'Declare the variables.
Dim objApp As Object
Dim objModel As Object
Dim objFaces As Object
Dim objTopFace As Object
Dim objFaceEdges As Object
Dim objRefPlane As Object
Dim objEdge As Object
Dim objXEdge As Object
Dim objProfileSet As Object
Dim objProfile As Object
Dim objLines As Object
Dim objL1 As Object, objL2 As Object
Dim objL3 As Object, objL4 As Object
Dim objRelations As Object
Dim dblStartPoint(3) As Double
Dim dblEndPoint(3) As Double
Dim dblHighY As Double
Dim dblLowY As Double
Dim dblProfileX As Double
Dim dblProfileY As Double
Dim dblPocketXSize As Double
Dim dblPocketYSize As Double
Dim intStatus As Integer
Dim i As Integer

'Enable error handling.
On Error Resume Next

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")
If Err Then
  MsgBox "Solid Edge must be running."
  Exit Sub
End If

'Determines the top face of the base feature.

'Reference the single model in the file.
Set objModel = objApp.ActiveDocument.Models(1)
If Err Then
  MsgBox "A model must exist in the file."
  Exit Sub
End If

'Reference the faces intersected by the defined ray.

Set objFaces = objModel.Body.FacesByRay _
  (0.01, 0.01, 0.01, 0, 0, 1)

'Save the top face (the first face intersected).
Set objTopFace = objFaces(1)
If Err Then 'no intersection occurred
  MsgBox "The base feature is not positioned correctly."
  Exit Sub
End If

'Turn off error handling
On Error GoTo 0

'Determine the edge of the face to use for
'defining the x axis of the reference plane by
'finding the horizontal edge with the smallest
'y values.

'Reference all the edges of the top face.
Set objFaceEdges = objTopFace.Edges

'Initialize y to a very small value.
dblLowY = -9999999

'Iterate through the edges.
For i = 1 To objFaceEdges.Count

'Reference the geometry object of the current edge.
Set objEdge = objFaceEdges.Item(i)

'Reference the endpoints of the edge.
Call objEdge.GetEndPoints(dblStartPoint, dblEndPoint)

'Check to see if y coordinates match (in tolerance) to see
'if the line is vertical.
If WithinTol(dblStartPoint(1), _
  dblEndPoint(1), _
  0.0000001) Then
  'Save the edge with the smallest y value.
  If dblLowY < dblStartPoint(1) Then
  'Save small y value and corresponding edge.
  dblLowY = dblStartPoint(1)
  Set objXEdge = objFaceEdges(i)
  End If
```

```
 End If
 Next I

 'Create the reference plane.
 Set objRefPlane =
objApp.ActiveDocument.RefPlanes.AddParallelByDistance _
   (ParentPlane:=objTopFace, _
   Distance:=0, _
   NormalSide:=igNormalSide, _
   Pivot:=objXEdge, _
   PivotOrigin:=igPivotEnd, _
   Local:=True)

 'Create the 2d profile.

 'Create a ProfileSet object.
 Set objProfileSet = objApp.ActiveDocument.ProfileSets.Add

 'Create an empty Profile object.
 Set objProfile = objProfileSet.Profiles.Add(objRefPlane)

 'Reference the corresponding location on the profile for
 'coordinate (1, 1.75, 2). All units must be converted to
 'database units (meters).
 Call objProfile.Convert3Dcoordinate _
   (1 * 0.0254, _
   1.75 * 0.0254, _
   2 * 0.0254, _
   dblProfileX, _
   dblProfileY)

 'Reference the Lines2d collection from the profile.
 Set objLines = objProfile.Lines2d

 'Determine the size of the pocket (in meters).
 dblPocketXSize = 4 * 0.0254
 dblPocketYSize = 1.5 * 0.0254

 'Draw lines that define the shape of the pocket.
 Set objL1 = objLines.AddBy2Points _
   (dblProfileX, _
   dblProfileY, _
   dblProfileX + dblPocketXSize, _
   dblProfileY)

 Set objL2 = objLines.AddBy2Points _
   (dblProfileX + dblPocketXSize, _
   dblProfileY, _
   dblProfileX + dblPocketXSize, _
   dblProfileY + dblPocketYSize)

 Set objL3 = objLines.AddBy2Points _
   (dblProfileX + dblPocketXSize, _
   dblProfileY + dblPocketYSize, _
   dblProfileX, _
   dblProfileY + dblPocketYSize)

 Set objL4 = objLines.AddBy2Points _
   (dblProfileX, _
   dblProfileY + dblPocketYSize, _
   dblProfileX, _
   dblProfileY)

 'Reference the Relations2d collection so you can add
relationships.
 Set objRelations = objProfile.Relations2d
```

```
'Add keypoint relationships to connect the endpoints of the lines.
Call objRelations.AddKeypoint _
  (objL1, igLineEnd, _
  objL2, igLineStart)

Call objRelations.AddKeypoint _
  (objL2, igLineEnd, _
  objL3, igLineStart)

Call objRelations.AddKeypoint _
  (objL3, igLineEnd, _
  objL4, igLineStart)

Call objRelations.AddKeypoint _
  (objL4, igLineEnd, _
  objL1, igLineStart)

'Complete the profile and validate it.
intStatus = objProfile.End(igProfileClosed)
If intStatus <> 0 Then
  MsgBox "profile validation failed."
  Exit Sub
End If

'Place the feature.
Set objFeature = objModel.ExtrudedCutouts.AddFinite _
  (Profile:=objProfile, _
  ProfileSide:=igLeft, _
  ProfilePlaneSide:=igLeft, _
  Depth:=0.5 * 0.0254)

'Turn off the profile display.
objProfile.Visible = False

End Sub

Public Function WithinTol (Val1 As Double, _
  Val2 As Double, _
  Tol As Double) As Boolean

'Compare input values within specified tolerance.
If Abs(Val1 - Val2) <= Tol Then
  WithinTol = True
Else
  WithinTol = False
End If
End Function
```

Sample Program—Creating 2-D Graphic Objects

The following program creates 2-D geometry objects and uses the collection to iterate through the objects by index or by name. The program assumes that Solid Edge is running and is in the Profile environment.

In this example, twenty lines are drawn on a profile. The start point of each line is random. The endpoint of each line is the profile's origin. The program uses three different methods to iterate through the Lines2d collection.

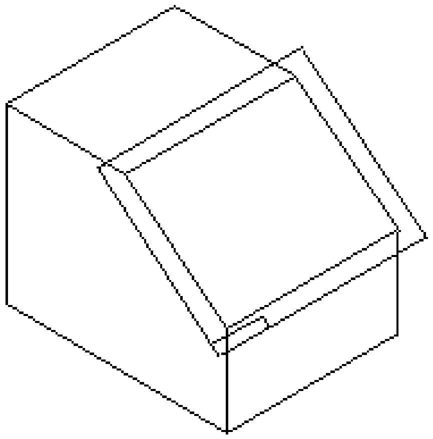
- First, the collections enumeration technique is used to iterate through the Line objects in the Lines2d collection by means of a "For Each" statement. With this technique, each line in the collection is automatically accessed and moved using the Move method. This is the most efficient of these three techniques.
- Second, the collection is traversed using a For loop. The Item property on the Lines2d collection is used to access each line in the collection by the object's Index property. As each line is accessed, the angle of the line is changed. The Item property is the default property for a collection. Therefore, the collection.Item(I) call can be shortened to collection(I). Both calls have the same result.
- The third technique also uses the Item property, but instead of using the Index of the object, it uses each object's Name to access it in the collection. The Item property can use either the object's Index or the object's Name to access the object. To hold a reference to an object for later use, use the Name property. The Name is a safer reference than the Index because an object's Name never changes, but its Index can change as objects are added to and removed from a collection.

```
Dim objApp As Object
Dim objLine2d As Object
Dim objProfile As Object
Dim objProfSets As Object
Dim objRefPlane As Object
Dim objLines As Object
Dim objLine As Object
Dim i As Integer
Dim pnt2x As Double
Dim pnt2y As Double
Dim vAngle As Double
Dim vName As String

'Connect to a running instance of Solid Edge
Set objApp = GetObject(, "SolidEdge.Application")

'Make sure the active environment is Profile.
If objApp.ActiveEnvironment <> "Profile" Then
  MsgBox "You must be in Profile environment."
  End
End If

'Access the current Profile.
Set objProfSets = objApp.ActiveDocument.ProfileSets
Set objProfile = objProfSets(objProfSets.Count).Profiles(1)

'Invoke random number generation.
Randomize
```

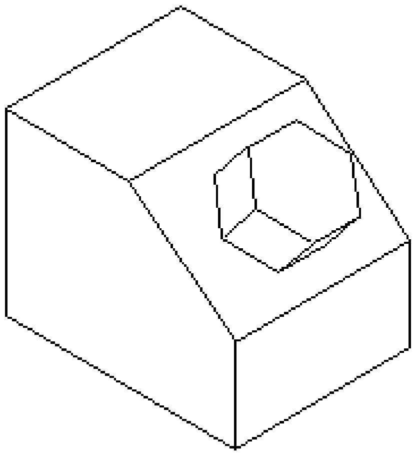
```
 'Initialize the base point to (0,0).
 pnt2x = 0
 pnt2y = 0

 'Reference the Lines2d collection.
 Set objLines = objProfile.Lines2d

 'Draw 20 lines.
 For i = 1 To 20
   Call objLines.AddBy2Points(Rnd / 10, Rnd / 10, pnt2x, pnt2y)
 Next i

 'Iterate through the Lines2d collection and move each line.
 For Each objLine In objProfile.Lines2d
   Call objLine.Move(0#, 0#, 0.01, 0.01)
 Next objLine

 'Use the Count attribute on the Lines2d collection to iterate
through the
 'collection, changing the angle of each line.
 For i = 1 To objLines.Count
   vAngle = objLines(i).Angle
   objLines(i).Angle = vAngle + 0.2
 Next i

 'Finally, use the shortened form of the Item attribute
 'via the Name value to iterate through the collection,
 'changing the length of each line.
 For i = 1 To objLines.Count
   vName = objLines(i).Name
   objLines(vName).Length = 0.05
 Next i
```

Sample Program—Creating a Profile

The following example shows how to create a profile that is used to place an extruded protrusion feature. The program assumes that the face and edge of the model have been retrieved and a reference plane has been created on the face. The initial part and the existing reference plane are shown in the following illustration:



The finished part is as follows:



The following program creates the profile and the protrusion feature. The RefPlane variable is assumed to be a public variable that is referencing the reference plane on the face. The size and position of the initial part is known; the center of the face is located at the coordinate (3.5, 2.5, 3.5).

```
Private Sub Command1_Click()
'Declare the variables.
Dim objApp As Object
Dim objDoc As Object
Dim objRefPlane As Object
Dim objProfileSet As Object
Dim objProfile As Object
Dim XOrigin As Double, YOrigin As Double
Dim ToSide As Double
Dim ToCorner As Double
Dim SideLength As Double
Dim objLines As Object
Dim objRelations As Object
Dim objHexLines(1 To 6) As Object
Dim Status As Long

'Define the constants.
Const PI = 3.14159265358979
Const HexSize = 2.5

'Define variables.
Set objApp = GetObject(, "SolidEdge.Application")
Set objDoc = objApp.ActiveDocument

'Set a reference to the RefPlane object on which the profile
'is to be constructed.
Set objRefPlane = objDoc.RefPlanes(4)

'Create a ProfileSet object.
Set objProfileSet = objDoc.ProfileSets.Add

'Create the blank Profile object using the known
'reference plane.
Set objProfile = objProfileSet.Profiles.Add(objRefPlane)

'Determine the location on the reference plane
'to draw the profile.
Call objProfile.Convert3DCoordinate( _
  x3d:=3.5 * 0.0254, _
  y3d:=2.5 * 0.0254, _
  z3d:=3.5 * 0.0254, _
  x2d:=XOrigin, _
  y2d:=YOrigin)

'Calculate some sizes used to draw the hexagon.
ToSide = (HexSize * 0.0254) / 2
ToCorner = ToSide / Cos(30 * (PI / 180))
SideLength = ToSide * Sin(30 * (PI / 180))

'Reference drawing object collections from the Profile.
Set objLines = objProfile.Lines2d
Set objRelations = objProfile.Relations2d

'Place lines to draw the hexagon.
Set objHexLines(1) = objLines.AddBy2Points( _
  XOrigin + ToCorner, _
  YOrigin, _
  XOrigin + SideLength, _
  YOrigin + ToSide)

Set objHexLines(2) = objLines.AddBy2Points( _
  XOrigin + SideLength, _
  YOrigin + ToSide, _
  XOrigin - SideLength, _
  YOrigin + ToSide)

Set objHexLines(3) = objLines.AddBy2Points( _
  XOrigin - SideLength, _
  YOrigin + ToSide, _
  XOrigin - ToCorner, _
  YOrigin)

Set objHexLines(4) = objLines.AddBy2Points( _
  XOrigin - ToCorner, _
  YOrigin, _
  XOrigin - SideLength, _
  YOrigin - ToSide)
```

```
Set objHexLines(5) = objLines.AddBy2Points( _
  XOrigin - SideLength, _
  YOrigin - ToSide, _
  XOrigin + SideLength, _
  YOrigin - ToSide)

Set objHexLines(6) = objLines.AddBy2Points( _
  XOrigin + SideLength, _
  YOrigin - ToSide, _
  XOrigin + ToCorner, YOrigin)

'Place key point relationships to connect the ends.
Call objRelations.AddKeypoint( _
  objHexLines(1), igLineEnd, _
  objHexLines(2), igLineStart)

Call objRelations.AddKeypoint( _
  objHexLines(2), igLineEnd, _
  objHexLines(3), igLineStart)

Call objRelations.AddKeypoint( _
  objHexLines(3), igLineEnd, _
  objHexLines(4), igLineStart)

Call objRelations.AddKeypoint( _
  objHexLines(4), igLineEnd, _
  objHexLines(5), igLineStart)

Call objRelations.AddKeypoint( _
  objHexLines(5), igLineEnd, _
  objHexLines(6), igLineStart)

Call objRelations.AddKeypoint( _
  objHexLines(6), igLineEnd, _
  objHexLines(1), igLineStart)

'Validate the profile.
Status = objProfile.End(igProfileClosed)

If Status <> 0 Then
  MsgBox "Profile Validation failure: " & Status
  Exit Sub
End If

'Create the feature.
Call objDoc.Models(1).ExtrudedProtrusions.AddFinite( _
  Profile:=objProfile, _
  ProfileSide:=igLeft, _
  ProfilePlaneSide:=igRight, _
  Depth:=1.5 * 0.0254)
```

This example follows the steps previously outlined:

Step 1: Create an empty Profile object.

First create a ProfileSet object, and then use the Add method on its Profiles collection object. Once you have created the Profile object, you have access to the 2-D geometry and relationships to define the shape of the profile.

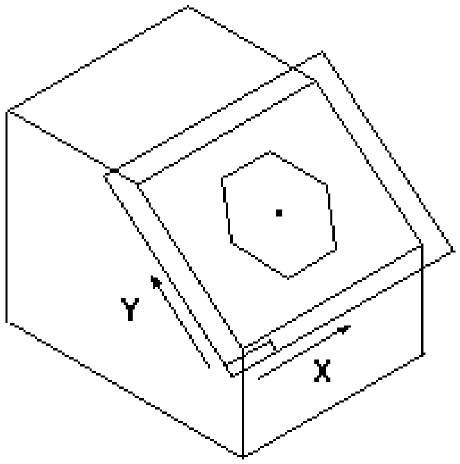
Step 2. Place geometry to define the shape of the profile.

In this example, the requirement is to draw a hexagon with its center at the center of the face. The difficulty in doing this is determining the relationship between the 2-D profile space and the 3-D model space. You do not have any control over the origin of the profile; it is automatically defined by the system.

In the interactive environment, you position the profile relative to the solid by placing dimensions. While this is possible in automation, dimensionally constraining the profile to the edges of the solid can be a difficult process, since you would need to determine the edges to which to constrain. The technique described here avoids this problem and allows you to create an under-constrained—but dimensionally correct—part.

Because the size and position of the initial solid in this example is known, the center of the face relative to 3-D model space (3.5, 2.5, 3.5) can be inferred. A profile is a 2-D object and represents a 2-D planar space. The Convert3DCoordinate method of the Profile object allows you to convert from 3-D model space to 2-D profile space. The input 3-D point is projected onto the reference plane, and the 2-D point in profile space is output.

The reference plane determines the axes of the profile. The small box in the corner of the reference plane represents the origin, and the x axis is in the long direction of the box. The following illustration shows the x and y axes for the reference plane and the calculated center point of the profile:



The hexagon is drawn relative to this center point. The system determines whether the profile elements are connected by checking to see if key point relationships exist between the endpoints of the elements.

Steps 3 and 4: Place relationships and dimensions on the geometry.

The program uses the AddKeyPoint method to fully constrain the geometry. In this example, dimensions are omitted.

Step 5: Validate the profile.

The program uses the End method on the Profile object, using a single argument, igProfileClosed (the profile must be closed), to define the validation criteria to use. The End method returns 0 if the validation succeeds, 1 if the validation fails. Once the profile is complete, it can be used as input to many of the feature add methods. In this example, an extruded protrusion feature is added.

Modifying and Querying Profiles

You can modify and query geometry contained in a profile. The OrderedGeometry method supported on the Profile object provides a way to iterate through the 2-D graphic objects in the profile. You can use the methods and properties of the 2-D graphic objects to directly manipulate the geometry in the profile. You can also use these methods and properties to access information about the geometry.

Another way to modify a profile is to edit the values of driving dimensions that control the geometry in the profile. The preferred way to do this is to access the Variable table.

Running Macros in the Profile Environment

In some cases, it is not necessary to automate an entire modeling process. Automating the entire process can limit the program's flexibility. For example, if you have parts that frequently have hexagonal-shaped features on them, you can increase your productivity by automating the creation of these features. The problem is that you might need both extruded and revolved protrusions, and also cutouts. This means you must use four different commands or an interface that allows the user to choose the type of feature. It can also be difficult to interact with the user to identify where a feature is to be placed.

One simple solution is to create a program that runs in the Profile environment. The interactive user selects the feature, face, and edge to define the profile plane, and then runs a program to draw the profile. In this way, the program can be used to create any feature that uses a profile. Once the profile is created, it behaves the same as any interactively placed geometry.

Writing a program to run from the Profile environment is easier than writing a program to perform the entire workflow of creating a feature. When you run from the Profile environment, the Solid Edge feature command does much of the work. You determine the Profile object on which to draw geometry, sketch the geometry, and add relationships.

Sample Program—Running a Macro

The following program illustrates how to automate using a macro. In this program, a square profile is drawn in the Profile environment:

```
 'Declare the variables.
 Dim objApp As Object
 Dim objProfile As Object
 Dim objProfileSets As Object
 Dim objLines As Object
 Dim objRelations As Object
 Dim objL(1 To 4) As Object

 'Set a constant for the size of the square.
 Const SIZE = 4 * 0.0254

 'Connect to a running instance of Solid Edge.
 Set objApp = GetObject(, "SolidEdge.Application")

 'Check to make sure the user is in the Profile environment.
 If objApp.ActiveEnvironment <> "Profile" Then
   MsgBox "This macro must be run from the Profile environment."
   End
 End If

 'Determine the profile on which to draw.
 Set objProfileSets = objApp.ActiveDocument.ProfileSets
 Set objProfile = objProfileSets(objProfileSets.Count).Profiles(1)

 'Set references to the collections used.
 Set objLines = objProfile.Lines2d
 Set objRelations = objProfile.Relations2d

 'Draw the geometry.
 Set objL(1) = objLines.AddBy2Points(0, 0, SIZE, 0)
 Set objL(2) = objLines.AddBy2Points(SIZE, 0, SIZE, SIZE)
 Set objL(3) = objLines.AddBy2Points(SIZE, SIZE, 0, SIZE)
 Set objL(4) = objLines.AddBy2Points(0, SIZE, 0, 0)

 'Add key point relationships between the ends of the lines.
 Call objRelations.AddKeypoint(objL(1), igLineEnd, objL(2),
igLineStart)
 Call objRelations.AddKeypoint(objL(2), igLineEnd, objL(3),
igLineStart)
 Call objRelations.AddKeypoint(objL(3), igLineEnd, objL(4),
igLineStart)
 Call objRelations.AddKeypoint(objL(4), igLineEnd, objL(1),
igLineStart)
```

The sample code connects to Solid Edge and verifies that the user is in the Profile environment. If the user is not in the Profile environment, a message is displayed, and the program exits.

Being in the Profile environment means that the feature command has already created a blank Profile object on which to sketch. In addition, a new ProfileSet object also exists. This ProfileSet will *usually* be the last one in the ProfileSets collection. The program sets a reference to the Profile object in this ProfileSet.

Note The workflow used in this example for connecting to the current Profile object works only when the user is creating a *new* feature. If the user edits an existing feature, this workflow could create a reference to the wrong profile.

Once you have accessed the Profile object, you can place geometry and apply relationships. In this example, a box is placed at the origin of the Profile object.

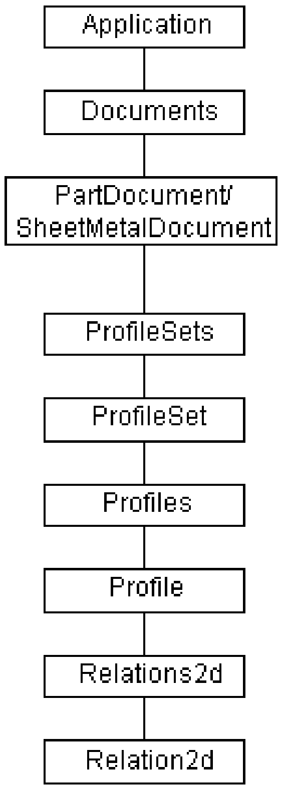
You may want to let the user specify where to place the geometry; the Solid Edge mouse control provides access to mouse input. In this way, the interactive user can use IntelliSketch to specify points relative to existing Profile geometry and part edges.

Working with 2-D Relationships—Overview

Solid Edge allows you to establish and maintain relationships on the 2-D elements that you draw in the Profile environment. These relationships control the size, shape, and position of an element in relation to other elements.

When an element changes, it is the relationships that drive the update of related elements. For example, if you have drawn a polygon with two lines parallel to one another and you modify the polygon, the two lines remain parallel. You can also change elements that have dimensions. If a driving dimension measures the radius of an arc, you can edit the value of the dimension to change the radius of the arc.

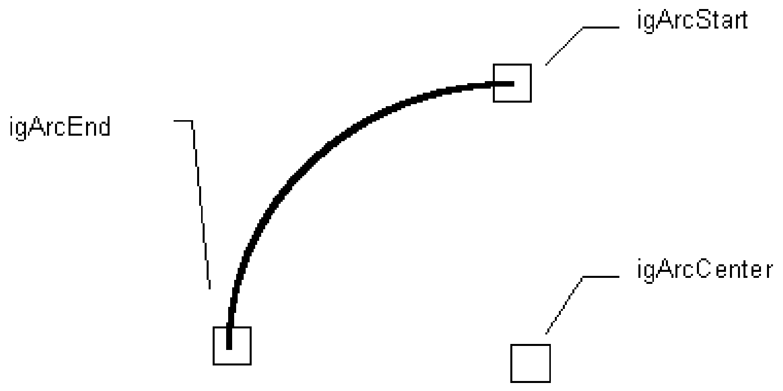
The object hierarchy for relationships is as follows:



The Relations2d object provides the methods for placing relationships and for iterating through all the relationships that exist on the associated profile. To establish a relationship between elements, use an add method on the Relations2d collection on the profile.

The objects for which you want to define the relationship must already exist on the profile. There is an add method on the Relation2d object for each type of relationship that can be defined. Once a relationship is defined, properties and methods on the Relation2d object allow you to edit existing relationships or query for information about relationships.

Many relationships are placed at a specific location on an element. For example, when you place a key point relationship to link the endpoints of two lines, you select one end of each line. The programming interface for placing relationships provides the ability to select specific points on an element by using predefined key points for each geometrical element. The key point to use is specified by using key point indexes. The following illustration shows the key points for an Arc element:



For more information on the methods to place, edit, and query relationships, see the Programming with Solid Edge on-line Help. For a complete list of defined types for different relationships and key point index constants for different elements, see the list of constants for ObjectType and KeypointIndexConstants available in the Solid Edge Constants type library using the Visual Basic Object Browser.

Sample Program—Adding Relationships

The following program places four lines and then adds key point and parallel relationships to create a closed parallelogram. You can learn the most from this example by stepping through the code and watching the results one step at a time. The original four lines are placed so that their ends are not connected. As each of the key point relationships is added, the endpoints of the lines are connected. When the parallel relationships are added between the sides, each side aligns itself with the opposite side.

```
'Declare the program variables.
Dim objApp As Object
Dim objProfileSets As Object
Dim objProfile As Object
Dim objLines As Object
Dim objRelations As Object
Dim L(1 To 4) As Object

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Make sure Profile is the active environment.
If objApp.ActiveEnvironment <> "Profile" Then
  MsgBox "This macro must be run from the Profile environment."
  End
End If

'Access the Profile object.
Set objProfileSets = objApp.ActiveDocument.ProfileSets
Set objProfile = objProfileSets(objProfileSets.Count).Profiles(1)

'Set references to the necessary collections.
Set objLines = objProfile.Lines2d
Set objRelations = objProfile.Relations2d

'Draw the geometry.
Set L(1) = objLines.AddBy2Points(0, 0, 0.1, 0)
Set L(2) = objLines.AddBy2Points(0.1, 0, 0.1, 0.2)
Set L(3) = objLines.AddBy2Points(0.1, 0.2, 0, 0.2)
Set L(4) = objLines.AddBy2Points(0, 0.2, 0, 0)

'Add keypoint relationships between the ends of the lines.
Call objRelations.AddKeypoint(L(1), igLineEnd, _
  L(2), igLineStart)
Call objRelations.AddKeypoint(L(2), igLineEnd, _
  L(3), igLineStart)
Call objRelations.AddKeypoint(L(3), igLineEnd, _
  L(4), igLineStart)
Call objRelations.AddKeypoint(L(4), igLineEnd, _
  L(1), igLineStart)

'Add parallel relationships between opposing lines.
Call objRelations.AddParallel(L(1), L(3))
Call objRelations.AddParallel(L(2), L(4))
End
```

Sample Program—Querying for Existing Relationships

It is sometimes useful to determine the relationships that are already defined for a specific object. For example, suppose you want to place horizontal relationships on all lines in a selection set. Rather than place a horizontal relationship on each object in the selection set, you can first check to see if the line already has a relationship that defines its orientation. To determine what relationships are already defined for a 2-D object, use the Relationships property of the object. This property returns a Relationships2d collection that contains all of the relationships on that object.

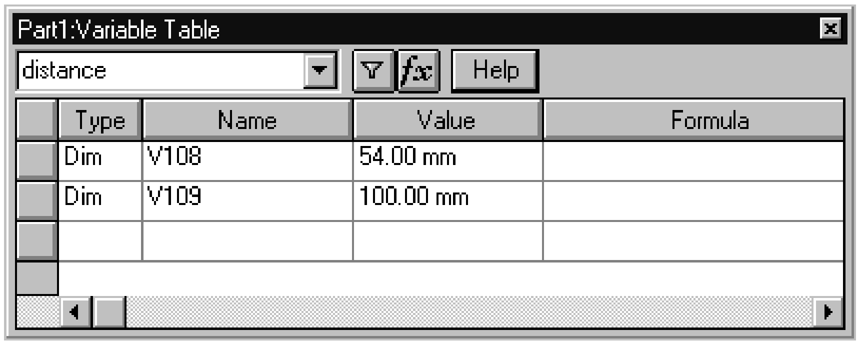
The following program shows how to access existing relationships:

```
 'Declare the program variables.
 Dim objApp As Object
 Dim objSelectSet As Object
 Dim objRelations2d As Object 'All relationships.
 Dim objRelation2d As Object 'The first relationship in the
collection.
 Dim objLineRelations As Object 'The collection of relationships on
each line.
 Dim objLineRelation As Object 'A relationship on a line.

 'Connect to a running instance of Solid Edge.
 Set objApp = GetObject(, "SolidEdge.Application")

 'Access the SelectSet object.
 Set objSelectSet = objApp.ActiveDocument.SelectSet

 'Make sure the selection set is not empty.
 If objSelectSet.Count = 0 Then
   MsgBox "You must select the geometry first."
   End
 End If

 'Access the Relations2d collection from the parent profile.
 Set objRelations2d = objSelectSet(1).Parent.Relations2d

 'Iterate through each element in the selection set.
 For Each object In objSelectSet
   'Look for Line2d objects.
   If object.Type = igLine2d Then
     'Get the Relations collection from the line
     Set objLineRelations = object.Relationships
     'If there are no relationships on the line,
     'add a horizontal one.
     If objLineRelations.Count = 0 Then
     objRelations2d.AddHorizontal object
     Else
     'Check to make sure no positional
     'relationship exists.
     For Each objLineRelation In objLineRelations
     If _
     objLineRelation.Type <> igHorizontalRelation2d _
     And _
     objLineRelation.Type <> igParallelRelation2d _
     And _
     objLineRelation.Type <> igPerpendicularRelation2d _
     And _
     objLineRelation.Type <> igVerticalRelation2d _
   Then
     'Add horizontal relationship.
     objRelations2d.AddHorizontal (object)
     End If
```

```
    Next objLineRelation
    End If
  End If
Next object
```

Working with Variables—Overview

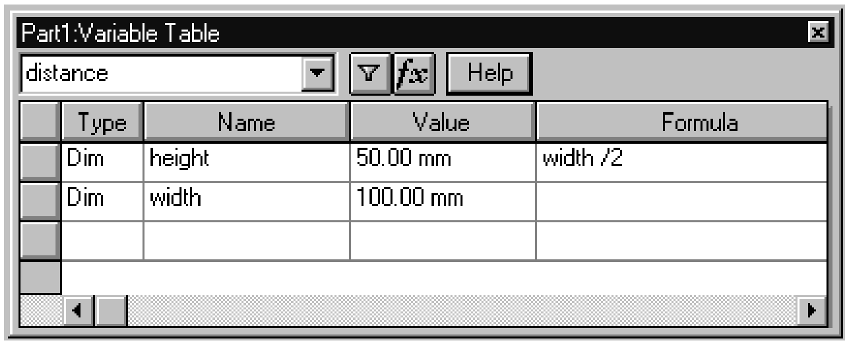
The variable system allows you to define relationships between variables and dimensions using equations, external functions, and spreadsheets. For example, you can construct a rectangular profile using four lines, and place two dimensions to control the height and width. The following illustration shows the Variable Table with these two dimensions displayed. (Every dimension is automatically available as a variable.)



Type	Name	Value	Formula
Dim	V108	54.00 mm	
Dim	V109	100.00 mm	

Using these variables, you can make the height a function of the width by entering a formula. For example, you could specify that the height is always one-half of the width. Once you have defined this formula, the height is automatically updated when the width changes.

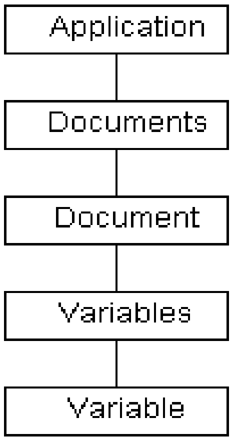
All variables have names; it is through these names that you can reference them. In the preceding illustration, the names automatically assigned by the system are V108 and V109. You can rename these dimensions; in the following illustration, V108 has been renamed to "height," and V109 has been renamed to "width."



Type	Name	Value	Formula
Dim	height	50.00 mm	width /2
Dim	width	100.00 mm	

Every variable has a value. This can be a static value or the result of a formula. Along with the value, a unit type is also stored for each variable. The unit type specifies what type of measurement unit the value represents. In this example, both of the variables use distance units. You can create Variables for other unit types such as area and angle. Values are displayed using this unit type and the unit readout settings.

The object hierarchy for variables is as follows:


Application
Documents
Document
Variables
Variable

Sample Program—Creating and Accessing Variable Objects

All variable automation is accessed through the Variables collection and Variable objects. The Variables collection serves two purposes: it allows you to create and access variable objects, and it allows you to work with dimensions as variables.

The Variables collection supports an Add method to create new Variable objects. It also supports the standard methods for iterating through the members of the collection. The following program connects to the Variables collection, creates three new variables, and lists them.

Note When debugging programs that interact with the Variable Table, it helps to have the Variable Table displayed while stepping through the program. The Variable Table shows the results as they are executed.

```
 'Declare the program variables.
 Dim objApp As Object
 Dim objVariables As Object
 Dim objVariable As Object

 'Connect to a running instance of Solid Edge.
 Set objApp = GetObject(, "SolidEdge.Application")

 'Access the Variables collection.
 Set objVariables = objApp.ActiveDocument.Variables

 'Create a variable with distance unit type.
 Call objVariables.Add("Var1", "1.25", igUnitDistance)

 'Create a variable without specifying the unit type (defaults to
distance).
 Call objVariables.Add("Var2", "3.56 cm")

 'Create a variable with area unit type.
 Call objVariables.Add("Var3", "Var1 * Var2", igUnitArea)

 'Iterate through all the variables, printing their name and value.
 For Each objVariable In objVariables
 Debug.Print objVariable.Name & " = " & objVariable.Value
 Next
```

Units with variables work the same as units in the system. Units are stored using internal values and then are appropriately converted for display. For example, all length units are stored internally as meters. When these units are displayed in the Variable Table, they are converted to the units specified in the Properties dialog box.

Note When iterating through the Variables collection, Variable objects—not Dimension objects—are returned. To iterate through dimensions, use the Dimensions collection.

Sample Program—Accessing Dimensions through the Variable Table

When working interactively with the Variable Table, both variables and dimensions are displayed in the table. This enables you to create formulas using the values of dimensions and also have formulas that drive the values of dimensions. In this workflow, there is no apparent difference between variables and dimensions. Internally, however, variables and dimensions are two distinct types of objects that have their own unique collections, properties, and methods.

The Variables collection allows you to work with dimensions in the context of variables through several methods on the collection. These methods include Edit, GetFormula, GetName, PutName, Query, and Translate. The following program uses dimensions through the Variables collection. The program assumes that Solid Edge is running and in the Profile environment.

```
'Declare the program variables.
Dim objApp As Object
Dim objVariables As Object
Dim objVariable As Object
Dim objLines As Object
Dim objTempLine As Object
Dim objDims As Object
Dim objDim1 As Object
Dim objDim2 As Object
Dim Dim2Name As String
Dim Formula As String
Dim objNamedDims As Object
Dim objNamedDim As Object
Dim objProfileSets As Object
Dim objProfile As Object

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Access the current Profile object.
Set objProfileSets = objApp.ActiveDocument.ProfileSets
Set objProfile = objProfileSets(objProfileSets.Count).Profiles(1)

'Reference the collections used.
Set objVariables = objApp.ActiveDocument.Variables
Set objLines = objProfile.Lines2d
Set objDims = objProfile.Dimensions

'Create a line.
Set objTempLine = objLines.AddBy2Points(0, 0, 0.1, 0.1)

'Place a dimension on the line to control its length.
Set objDim1 = objDims.AddLength(objTempLine)

'Make the dimension a driving dimension.
objDim1.Constraint = True

'Create a second line.
Set objTempLine = objLines.AddBy2Points(0, 0.1, 0.1, 0.2)

'Place a dimension on the line to control its length.
Set objDim2 = objDims.AddLength(objTempLine)

'Make the dimension a driving dimension.
objDim2.Constraint = True
```

```
 'Assign a name to the dimension placed on the first line.
 Call objVariables.PutName(objDim1, "Dimension1")

 'Retrieve the system name of the second dimension, and display
 'it in the debug window.
 Dim2Name = objVariables.GetName(objDim2)
 Debug.Print "Dimension Name = " & Dim2Name

 'Edit the formula of the second dimension to be half
 'the value of the first dimension.
 Call objVariables.Edit(Dim2Name, "Dimension1/2.0")

 'Retrieve the formula from the dimension, and print it to the
 'debug window to verify.
 Formula = objVariables.GetFormula(Dim2Name)
 Debug.Print "Formula = " & Formula

 'This demonstrates the ability to reference a dimension object by
its name.
 Set objDim1 = objVariables.Translate("Dimension1")

 'To verify a dimension object was returned, change its
 'TrackDistance property to cause the dimension to change.
 objDim1.TrackDistance = objDim1.TrackDistance * 2

 'Use the Query method to list all all user-defined
 'variables and user-named Dimension objects and
 'display in the debug window.
 Set objNamedDims = objVariables.Query("*")
 For Each objNamedDim In objNamedDims
   Debug.Print objVariables.GetName(objNamedDim)
 Next
```

Sample Program—Variable Objects

In addition to the properties and methods on the Variables collection, properties and methods are also available on the Variable objects. These properties and methods read and set variable names, define formulas, set values, and specify units of measure. The following program shows how to work with Variable objects:

```
'Declare the program variables.
Dim objApp As Object
Dim objVariables As Object
Dim objVar1 As Object

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Access the Variables collection.
Set objVariables = objApp.ActiveDocument.Variables

'Add a variable, and print its value to the debug window.
Set objVar1 = objVariables.Add("NewVar", "1.5")
Debug.Print "NewVar = " & objVar1.Value

'Change the formula of the variable to a function.
objVar1.Formula = "Sin(0.1)"

'Change the name of the variable.
objVar1.Name = "NewName"

'Change the value of the variable. This will not change
'the value of the variable.
objVar1.Value = 123

'Change the formula of the variable to a static value.
'This causes the formula to be removed and sets the value.
objVar1.Formula = "456"

'Change the value of the variable. It works now.
objVar1.Value = 789

'Delete the variable.
objVar1.Delete
```

Note If a variable's value is defined using a formula, the value of the variable can be changed only by editing the formula. You can remove a formula by setting the value of the formula to a static value. In addition, when you set the Value property of a variable, it is assumed to be in database units, such as meters for distance. Use the Formula property for user-defined units.

Using External Sources

Two external sources are available to drive the values of variables and dimensions: Visual Basic functions and subroutines and Excel spreadsheets.

Visual Basic Functions and Subroutines

Interactively, when using functions and subroutines to drive a variable's formula, you use the Function Wizard to define the link to the function. You can also set up this link through automation as shown in the following example. The following is a listing of a .BAS file that contains a function and a subroutine. For this example, the .BAS file is called *UserFunc.BAS*.

```
Function AddThree(InValue As Double) As Double
AddThree = InValue + 3
End Function

Sub AddMultiply(InOne As Double, InTwo As Double,
  ByRef OutOne As Double, ByRef OutTwo As Double)
  OutOne = InOne + InTwo
  OutTwo = InOne * InTwo
End Sub
```

The following program shows how to use user-defined functions from an external .BAS file.

```
'Declare the program variables.
Dim objApp As Object
Dim objVariables As Object
Dim objVar1 As Object
Dim objVar2 As Object

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Reference the Variables collection.
Set objVariables = objApp.ActiveDocument.Variables

'Add four variables.
Set objVar1 = objVariables.Add("A", "0")
Call objVariables.Add("B", "34")
Call objVariables.Add("C", "0")
Set objVar2 = Variables.Add("D", "0")

'Modify the formula of the variable to call the
'external function AddThree. The input is the
'value of variable B and the result will be
'assigned the variable A.
objVar1.Formula = "C:\Temp\UserFunc.AddThree (B)"

'Modify the formula of variable D to use the external
'subroutine AddMultiply. The input is the variables
'A and B, and the output is C and D.
Var2.Formula = "C:\Temp\UserFunc.AddMultiply (A,B,C,D)"
```

Use the following notation when creating a formula that contains an external function or subroutine:

```
module.function (argument1, argument2, ... )
```

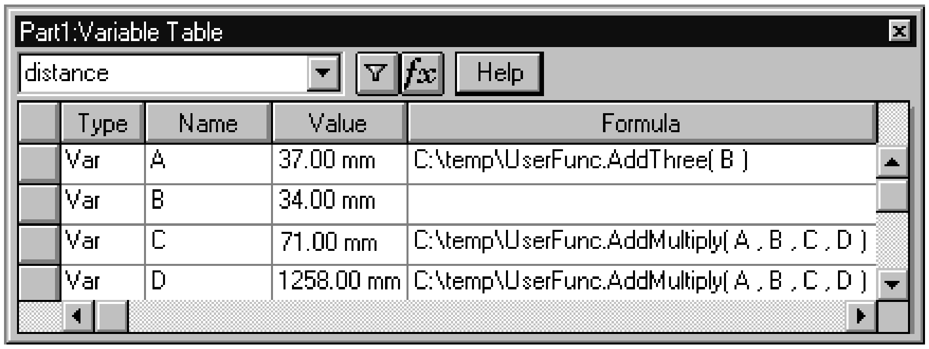
Specify the module name without the .BAS extension. Specify the arguments of the function after the function name.

In the previous example, first an external function is used. This function requires a single argument for which the variable B has been specified. The resulting value of the function is assigned to the variable A since the formula is being added to that variable object.

In the next step, the program assigns an external subroutine as the formula of a variable. Using a subroutine is slightly different than using a function, since a subroutine does not explicitly return a value as a function does. A subroutine can return values through one or more of its arguments.

When calling an external subroutine through the variable table, its arguments must be either input or output and cannot serve both purposes. In this example, the subroutine has two input arguments, InOne and InTwo, and two output arguments, OutOne and OutTwo. Whether a variable is input or output is specified in the subroutine definition using the ByVal keyword for input variables and the ByRef keyword for output variables.

The formula specifies that the subroutine AddMultiply is to be called with the variables A and B as input, and C and D as output. This formula is being defined for the variable object referenced by the object variable Var2, which is named D. However, because the variable named C is an output of this subroutine, the formula is also assigned to that variable. The following illustration shows the results in the variable table after running the program.



Type	Name	Value	Formula
Var	A	37.00 mm	C:\temp\UserFunc.AddThree(B)
Var	B	34.00 mm	
Var	C	71.00 mm	C:\temp\UserFunc.AddMultiply(A , B , C , D)
Var	D	1258.00 mm	C:\temp\UserFunc.AddMultiply(A , B , C , D)

Querying the Solid Model

A solid model contains information such as features, faces, and edges. Extracting specific information programmatically from a solid model can be difficult. For example, to create a cutout feature, you specify a face of the solid on which to sketch the profile. While it is trivial to identify this face interactively, it can be a difficult task to simulate in a non-interactive program. Several methods are available to obtain useful information from a model.

Lists of Features

You can find all of the features in a solid by using the collections in the object hierarchy. To perform an operation on all of the features of a specific type, iterate through the collection of that feature type. To review all of the features in a model regardless of type, use the Features collection, which contains all the features in the model.

This example uses the Chamfers collection to iterate through all the chamfers in the model. It checks each one to see if it is defined as a 45-degree chamfer and changes the setback of each 45-degree chamfer to 0.25 inches.

```
 Dim objApp As Object
 Dim objChamfers As Object
 Dim i As Integer

 'Connect to a running instance of Solid Edge.
 Set objApp = GetObject(, "SolidEdge.Application")

 'Access the Chamfers collection.
 Set objChamfers = objApp.ActiveDocument.Models(1).Chamfers

 'Iterate through all chamfers in the collection.
 For i = 1 To objChamfers.Count
   'Check that the chamfer is a 45-degree setback type.
   If objChamfers(i).ChamferType = igChamfer45degSetback Then
   'Change the setback value of the chamfer.
   objChamfers(i).ChamferSetbackValue1 = 0.25 * 0.0254
   '(ChamferSetbackValue1 must be converted to database units,
meters.)
   End If
 Next i
```

The next example uses the Features collection on the Model object to turn on the dimension display for all features in the model.

```
 Dim objApp As Object
 Dim objFeatures As Object
 Dim objFeature As Object

 'Connect to a running instance of Solid Edge.
 Set objApp = GetObject(, "SolidEdge.Application")

 'Access the Features collection.
 Set objFeatures = objApp.ActiveDocument.Models(1).Features

 'Iterate through all the features in the collection.
```

```
For Each objFeature In objFeatures
  'Turn on the dimension display.
  objFeature.ShowDimensions = True
Next
```

Getting Definition Information from a Feature

Each type of feature has a set of properties that define it. Using these properties and methods, you can extract the defining information from the feature. Below is a simple example that uses a property of a cutout feature to determine its extent. Depending on the extent of the cutout, it uses another property to determine the depth of the cutout.

```
'Declare the program variables.
Dim objApp As Object
Dim objCutout As Object
Dim Depth As Double

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Access the Cutout.
Set objCutout = _
  objApp.ActiveDocument.Models(1).ExtrudedCutouts(1)

'If the cutout has a finite extent,
If objCutout.ExtentType = igFinite Then

  'store the depth of the cutout in the Depth variable,
  Depth = objCutout.Depth

  'and show the results.
  MsgBox "The depth of the cutout is " & Depth & "."
End If
```

Getting Geometric Information from the Model and Features

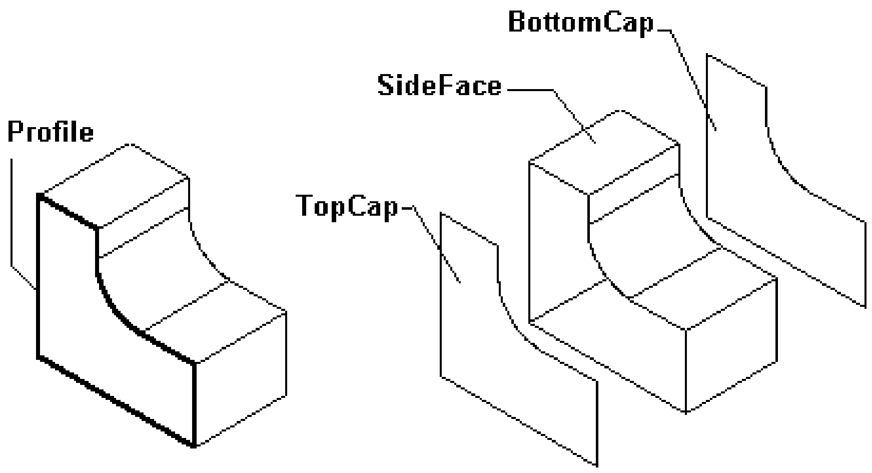
A model is the result of placing a series of features. At its lowest level, the model is a set of faces that enclose a volume. Two faces connect at an *edge*. It is frequently necessary to go beyond the feature information and extract geometric information from the faces and edges of the model. Many feature constructions require the input of faces and/or edges of the model.

Properties on the Model object and the individual feature objects allow you to access faces and edges from the model. The Faces, FacesByRay, and Edges properties are supported by the Model object and all feature objects. For most profile-based features, the TopCap, BottomCap, and SideFaces properties are supported.

The Geometry property on the Faces and Edges returns various 3-D wireframe and surface objects. The type of object returned depends on the geometry of the face or edge. You can determine the type returned through the Type property of the associated object. All of the objects returned by the Geometry property support a Type property. You can use this property to determine what type of 3-D geometry the edge or face is, and consequently what methods and properties it supports. See the Programming with Solid Edge on-line Help file for specific information about the properties and methods supported for each object.

Note Because these edges and faces are actually defined by the solid model, only the methods and properties that allow you to extract information from the geometry are supported. Any properties or methods that modify the geometry are not supported in Solid Edge.

Most of the profile-based features also support properties that allow you to access specific faces from the solid. These properties are BottomCap, TopCap, and SideFaces. The faces are located on the feature as follows:



Except for features created with a symmetric extent, the top cap is always in the same plane as the profile.

The following example shows how to use these methods. In this example, a ray is defined and intersected with the model. The result is a Faces collection that contains all of the faces intersected. The first face from this collection is assigned to a Face variable. This Face object is used to reference a single item.

The next step uses the Edges method of the Face object to access all the edges of the face. One of the edges from this collection is assigned to an Edge object.

The Geometry property returns a 3-D geometry object from the edge. The type of geometry object returned depends on the geometry of the edge, so the type can vary; it can be a line, an arc, a curve, or whatever geometry exists in the model. This sample verifies that the geometry is a line and uses Line object methods to obtain the endpoints of the line.

```
'Declare the program variables.
Dim objApp As Object
Dim objFaces As Object
Dim objFace As Object
Dim objEdges As Object
```

```
 Dim objEdge As Object
 Dim objLine As Object
 Dim StartPoint(3) As Double
 Dim EndPoint(3) As Double
 Dim objFeature As Object

 'Connect to a running instance of Solid Edge.
 Set objApp = GetObject(, "SolidEdge.Application")

 'Access the first Extruded Protrusion in the Model.
 Set objFeature =
objApp.ActiveDocument.Models(1).ExtrudedProtrusions(1)

 'Access the faces on that Feature that intersect with
 'the defined vector.
 Set objFaces = objFeature.FacesByRay(Xorigin:=0, _
   Yorigin:=0, _
   Zorigin:=1, _
   Xdir:=0, _
   Ydir:=0, _
   Zdir:=-1)

 'Access the first face intersected.
 Set objFace = objFaces(1)

 'Access all the edges of the face.
 Set objEdges = objFace.Edges

 'Access the first edge in the collection.
 Set objEdge = objEdges(1)

 'If the edge is a line,
 If objEdge.Geometry.Type = igLine Then

 'Determine the StartPoint and EndPoint of the line.
 Call objEdge.GetEndPoints(StartPoint, EndPoint)

 'Store the array values in variables.
 X1 = StartPoint(0)
 Y1 = StartPoint(1)
 Z1 = StartPoint(2)
 X2 = EndPoint(0)
 Y2 = EndPoint(1)
 Z2 = EndPoint(2)

 'And then format them to 2-decimal point accuracy.
 X1 = Format(X1, ".00")
 Y1 = Format(Y1, ".00")
 Z1 = Format(Z1, ".00")
 X2 = Format(X2, ".00")
 Y2 = Format(Y2, ".00")
 Z2 = Format(Z2, ".00")

 'Use the coordinates as needed.
   Text1.Text = X1 & ", " & Y1 & ", " & Z1
   Text2.Text = X2 & ", " & Y2 & ", " & Z2
 End If
```

Modifying the Solid Model

There are several ways to modify a model through automation.

The simplest and most common modification is to change the dimensions of a feature. You can do this either by editing the properties of the feature or by editing the parent profile. For example, to edit the depth of a cutout, change the value of the feature's Depth property. To change the size of the cutout, change the profile geometry or the dimensions that control the profile.

Either workflow is acceptable. However, the recommended workflow is to edit dimensional values using the variable table entries that correspond to the feature's dimensions. For information on the Variable Table and how to create parametric libraries, Chapter 13, *Working with Variables*.

In addition to using the properties of the part to change dimensional values, you can also use the properties of the feature to control other attributes. Some of these properties are unique to each feature, and others are common properties shared by all features. The following properties are commonly shared by all features:

- The ability to display the dimensions associated with the feature and its profile.
- The ability to suppress the feature.
- The ability to change its order of placement among the features of the model.

Programming Families of Parts

A family of parts is a series of similar parts in different sizes or with slightly different features. For example, a standard hex head bolt comes in many diameters, lengths, and thread specifications. All of these potentially unique bolts are simply variations of a single design. That is, they are all members of the same family.

Solid Edge provides a robust interface for users to construct families of parts. However, another way to do it through automation is to interactively create a single member of the family in Solid Edge, and then automatically edit the dimensional values of the part to create any other member in the family. The primary advantage to this method is that you create the graphics of the part interactively a single time. Once the part is created, it can be modified through the variable table. Using programming tools and Microsoft Office products, you can provide an interface for editing the parts. If needed, you can easily limit designers to a predefined set of dimensions.

One of the sample programs delivered with Solid Edge demonstrates this. The *Modifying Graphics from Excel Data* sample, which is described in Appendix B, *Sample Programs,* illustrates a family of parts application that uses the variable table to create a link between a part and an Excel spreadsheet.

CHAPTER 10

Modeling with Surfaces

This chapter introduces automation using the surfacing approach to modeling.

Surface Modeling

Surface modeling is characterized by using control points to define 2-D and 3-D curves, and these curves are used to define surfaces. With surface-based features, edges drive the model and curves are a major part of the model's definition.

When the surface model forms a valid, enclosed volume, the surface model can be stitched together into a solid model, and if required, solid modeling features can be added.

Sample Program—Blending Surfaces

The following program assumes that the active document is a part document containing two surfaces that can be blended. For example, a valid model could be two intersecting planar construction surfaces.

```
 Dim objApp As SolidEdgeFramework.Application
 Dim objDoc As SolidEdgePart.PartDocument
 Dim objModel As Model
 Dim objConstructions As Constructions
 Dim objBody As Body
 Dim objFaces As Faces
 Dim objFace1 As Face
 Dim objFace2 As Face
 Dim objRounds As Rounds
 Dim objSurfaceBlend As Round
 Dim objBlendType As BlendShapeConstants
 Dim dBlendValue As Double
 Dim strName As String
 Dim Status As Long
 Dim retType As Long

 On Error Resume Next

 RetSuccess = True
 strRetStatusMsg = ""

 ' Create/get the application with specific settings
 Set objApp = GetObject(, "SolidEdge.Application")
 If Err Or objApp Is Nothing Then
    strRetStatusMsg = "Could not get application."
    GoTo ErrorHandler
 End If

 ' open the document mentioned in the doc files.
 Set objDoc = objApp.Documents.Open(strDocFiles)
 If Err Or objDoc Is Nothing Then
    strRetStatusMsg = "Could not open document " & strDocFiles
    GoTo ErrorHandler
 End If

 ' get the first face
 Set objFace1 = objDoc.Constructions(1).Body.Shells(1).Faces(1)
 If Err Or objFace1 Is Nothing Then
    strRetStatusMsg = "Failed to get first face object."
    GoTo ErrorHandler
 End If

 ' get the second face
 Set objFace2 = objDoc.Constructions(2).Body.Shells(1).Faces(1)
 If Err Or objFace2 Is Nothing Then
    strRetStatusMsg = "Failed to get second face object."
    GoTo ErrorHandler
 End If

 Set objRounds = objDoc.Constructions(1).Rounds
 If Err Or objRounds Is Nothing Then
    strRetStatusMsg = "Failed to round faces object."
    GoTo ErrorHandler
 End If

 Set objSurfaceBlend = objRounds.AddSurfaceBlend(objFace1, igLeft,
objFace2, igRight, 0.0254, False, False)
 If Err Or objSurfaceBlend Is Nothing Then
    strRetStatusMsg = "Failed to create surface blend."
```

```
   GoTo ErrorHandler
End If

' Status
Status = objSurfaceBlend.Status
If Err Or Status <> igFeatureOK Then
   strRetStatusMsg = "Failed to get Status on SurfaceBlend."
   GoTo ErrorHandler
End If

' now get the blend type
objBlendType = objSurfaceBlend.BlendShape

' set the blend type to G2
objSurfaceBlend.BlendShape = igBlendShapeG2Continuous
If Err Then
   strRetStatusMsg = "Failed to set BlendShape."
   GoTo ErrorHandler
End If

' now get the blend value
dBlendValue = objSurfaceBlend.BlendShapeValue

' now set the blend value
objSurfaceBlend.BlendShapeValue = 2
If Err Then
   strRetStatusMsg = "Failed to set BlendShapeValue."
   GoTo ErrorHandler
End If
```

Sample Program—Creating a Parting Surface

The following program creates a block protrusion and extracts a closed loop of edges from one face as the parting line. It then creates a parting surface that passes through the parting line.

```
 Private Sub Form_Load()
 Dim objApp As Object
 Dim objDoc As Object
 Dim objModel As Object
 Dim objProfArr(1 To 2) As Object
 Dim objLines As Object
 Dim objRelns1 As Object
 Dim objThnWls As Object
 Dim objFaces As Object
 Dim objFace As Object
 Dim objEdges As Object
 Dim objEdgeArr() As Object
 Dim nIndex As Integer
 Dim nEdges As Integer

 Dim objConstructions As Constructions
 Dim objConstructionModel As ConstructionModel
 Dim objPartingSurfaces As SolidEdgePart.PartingSurfaces
 Dim objPartingSurface As SolidEdgePart.PartingSurface

 Set objApp = GetObject(, "SolidEdge.Application")
 Set objDoc = objApp.ActiveDocument

 ' get the constructions collection
 Set objConstructions = objDoc.Constructions

 Set objPartingSurfaces = objConstructions.PartingSurfaces
 If Err Or objPartingSurfaces Is Nothing Then
  strRetStatusMsg = "Could not get PartingSurface collection."
 End If

 ' Draw the Profile
 Set objProfArr(1) =
objDoc.ProfileSets.Add.Profiles.Add(pRefPlaneDisp:=objDoc.RefPlanes
(1))Set objLines = objProfArr(1).Lines2d
 Call objLines.AddBy2Points(X1:=0, Y1:=0, X2:=0.04, Y2:=0)
 Call objLines.AddBy2Points(X1:=0.04, Y1:=0, X2:=0.04, Y2:=0.04)
 Call objLines.AddBy2Points(X1:=0.04, Y1:=0.04, X2:=0, Y2:=0.04)
 Call objLines.AddBy2Points(X1:=0, Y1:=0.04, X2:=0, Y2:=0)

 ' Relate the Lines to make the Profile closed
 Set objRelns1 = objProfArr(1).Relations2d
 Call objRelns1.AddKeypoint(Object1:=objLines(1),
Index1:=igLineEnd, Object2:=objLines(2), Index2:=igLineStart)
 Call objRelns1.AddKeypoint(Object1:=objLines(2),
Index1:=igLineEnd, Object2:=objLines(3), Index2:=igLineStart)
 Call objRelns1.AddKeypoint(Object1:=objLines(3),
Index1:=igLineEnd, Object2:=objLines(4), Index2:=igLineStart)
 Call objRelns1.AddKeypoint(Object1:=objLines(4),
Index1:=igLineEnd, Object2:=objLines(1), Index2:=igLineStart)

 Set objModel =
objDoc.Models.AddFiniteExtrudedProtrusion(NumberOfProfiles:=1, _
     ProfileArray:=objProfArr, ProfilePlaneSide:=igRight, _
     ExtrusionDistance:=0.04)
 objProfArr(1).Visible = False

 ' Check the status of Base Feature
 If objModel.ExtrudedProtrusions(1).Status <> igFeatureOK Then
```

```
  MsgBox ("Error in the Creation of Base Protrusion Feature
object")
 End If

 ' Get all faces
 Set objFaces =
objModel.ExtrudedProtrusions(1.Faces(FaceType:=igQueryAll)

 ' Get all edges of the first face
 Set objFace = objFaces(1)
 Set objEdges = objFace.Edges

 nEdges = objEdges.Count

 ReDim objEdgeArr(1 To nEdges)

 For nIndex = 1 To nEdges
   Set objEdgeArr(nIndex) = objEdges(nIndex)
 Next nIndex

 ' Create the parting surface
 Set objPartingSurface =
objPartingSurfaces.Add(NumberOfEdges:=nEdges,
EdgeArray:=objEdgeArr, _
 pReferencePlane:=objDoc.RefPlanes(2), Distance:=0.01,
SideType:=igLeft)

 ' Check the status of the Feature
 If Err Or objPartingSurface Is Nothing Then
  strRetStatusMsg = "Failed to create a PartingSurface."
 End If

 'Set objFaces = objPartingSurface.Faces(FaceType:=igQueryAll)
 'Set objFace = objFaces(1)
 'Set objEdges = objFace.Edges

 If objPartingSurface.SideType <> igLeft Then
  strRetStatusMsg = "SideType dose not match."
 End If
```

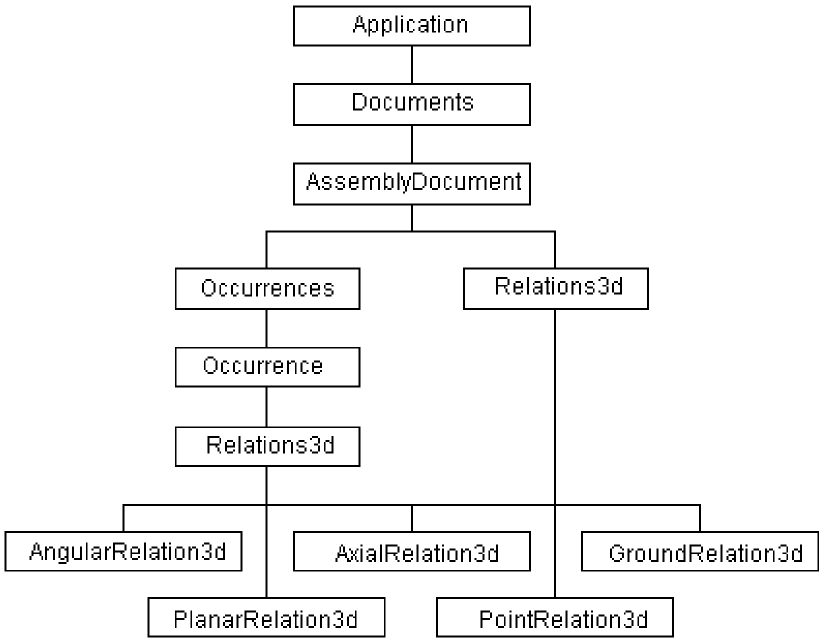
C H A P T E R 11

Working with Assemblies

This chapter describes the automation interface of the Solid Edge Assembly environment.

Assembly Document Anatomy

An assembly is a document that is a container for OLE links to other documents that contain parts or other assemblies. An assembly document is used exclusively for assemblies and has its own automation interface. This programming interface allows you to place parts into an assembly and examine existing parts and subassemblies and their relationships. The assembly automation model is as follows:



The Assembly Coordinate System

When you work interactively in Solid Edge, there is no need to be aware of a coordinate system. This is because you can place parts and subassemblies relative to existing parts and subassemblies. When modeling programmatically, however, it is often easier to position parts and subassemblies by specifying locations in space rather than by defining relationships to existing geometry.

The Solid Edge Part and Assembly environments use the Cartesian coordinate system. Solid Edge uses a uniform set of measurement definitions, a system referred to as internal or database units. The internal unit for distance is a meter; it follows then that units used when expressing coordinates are always meters.

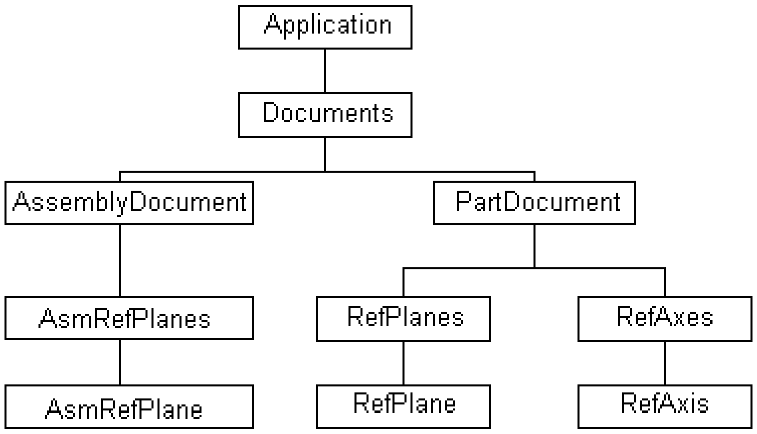
Working with Assembly Reference Planes

An assembly reference plane (the AsmRefPlane object) enables you to place relationships and manage assembly display. The corresponding collection object, AsmRefPlanes, provides two methods to enable you to place reference planes. These methods correspond to the reference plane commands that are available in the interactive environment.

- AddAngularByAngle—Creates angular and perpendicular reference planes. The perpendicular reference plane is a special case of the angular reference plane where the angle is pi/2 radians (90 degrees).
- AddParallelByDistance—Creates coincident and parallel reference planes. A coincident reference plane is a parallel reference plane where the offset value is zero.

Creating Reference Elements

Reference elements—reference planes and reference axes—are important parts of 3–D design. Most features require that you define a 2–D profile and project that profile through space to shape the feature. Creating a representative profile often requires working with a reference plane. For revolved features, a reference axis is also needed. Methods for creating these reference elements are similar to the interactive commands that perform the same functions. The hierarchical chart for reference elements is as follows:



Working with Reference Axes

A reference axis defines the axis of revolution for a revolved feature. Reference axes are usually created in the Profile environment when a user defines the profile of the revolution. Two objects—the collection object, RefAxes, and the instance object, RefAxis—are available to enable you to manipulate reference axes in your models.

Sample Program—Creating Reference Elements

The following program connects to a running instance of Solid Edge, creates an Assembly document and places an assembly reference plane using the AddAngularByAngle method. Then the program creates a Part document and places a reference plane using the AddParallelByDistance method.

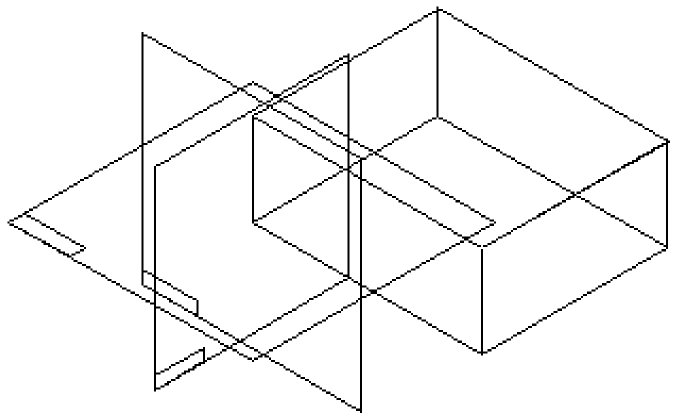
```
 'Declare the program variables.
 Dim objApp As Object
 Dim objDocs As Object
 Dim objAssyDoc As Object
 Dim objAsmRefPlanes As Object
 Dim objAsmRefPlane As Object
 Dim objPPlane As Object
 Dim objPartDoc
 Dim objRefPlanes As Object
 Dim objRefPlane As Object
 Const PI = 3.14159265358979

 'Connect to a running instance of Solid Edge.
 Set objApp = GetObject(, "SolidEdge.Application")

 'Access the Documents collection object.
 Set objDocs = objApp.Documents

 'Add an Assembly document.
 Set objAssyDoc = objDocs.Add("SolidEdge.AssemblyDocument")

 'Access the AsmRefPlanes collection object.
 Set objAsmRefPlanes = objAssyDoc.AsmRefPlanes

 'Create a reference plane at an angle to a
 'principal reference plane.
 Set objPPlane = objAssyDoc.AsmRefPlanes(2)
 Set objAsmRefPlane = objAsmRefPlanes.AddAngularByAngle( _
   ParentPlane:=objPPlane, _
   Angle:=(2 * PI / 3), _
   Pivot:=objAssyDoc.AsmRefPlanes(1), _
   PivotOrigin:=igPivotEnd, _
   NormalSide:=igNormalSide, _
   Local:=True)

 'Add a Part document.
 Set objPartDoc = objDocs.Add("SolidEdge.PartDocument")

 'Access the RefPlanes collection object.
 Set objRefPlanes = objPartDoc.RefPlanes

 'Create a global reference plane parallel to the top reference
plane.
 Set objRefPlane = _
   objRefPlanes.AddParallelByDistance(ParentPlane:=objRefPlanes(1),
_
   Distance:=0.1, _
   NormalSide:=igNormalSide,
   Local:=False)
```

Placing Occurrences

The automation interface for the assembly environment allows you to place parts and subassemblies into an assembly. This is handled by the AddByFilename method, which is provided on the Occurrences collection object. Parts and subassemblies are differentiated by the Subassembly property on each Occurrence object. The following example shows how to place a part into an assembly. This example accesses the Occurrences collection from the active document and places the part.

```
 'Declare the program variables.
 Dim objApp As Object
 Dim objOccurrences As Object
 Dim objOccurrence1 As Object

 'Connect to a running instance of Solid Edge.
 Set objApp = GetObject(, "SolidEdge.Application")

 'Access the Occurrences collection object.
 Set objOccurrences = objApp.ActiveDocument.Occurrences

 'Add an Occurrence object.
 Set objOccurrene1 = objOccurrences.AddByFilename("c:\Drawing
Files\Block.par")
```

Parts or subassemblies are initially placed into the assembly at the same location and position they maintain in their original files. The following illustration shows a block and its position relative to the three initial global reference planes. The block is positioned so its corner is at the coordinate (0,0,0). When this part is placed into an assembly using the AddByFilename method, it is placed in the same location and orientation in the assembly file as it existed in the original part file. Subassemblies follow the same rules.

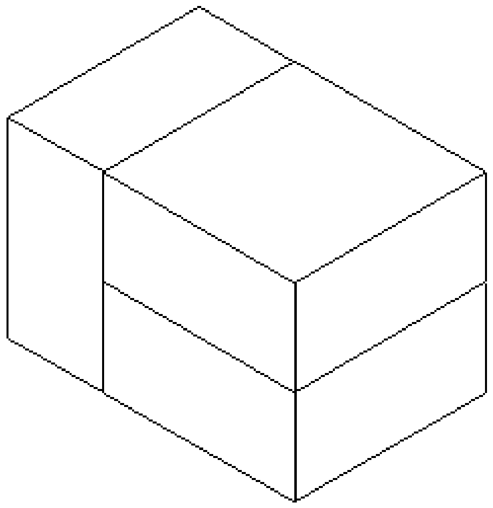


Manipulating Occurrences

Because occurrences are placed in the same relative location and orientation in which they were initially created, you will typically change the part or subassembly's position and orientation after placement.

Note These methods apply only to grounded occurrences. Occurrences that are placed with relationships to other occurrences have their location and orientation defined by their relationships to the other occurrences.

To show how to use these methods, consider a block with dimensions of 100 mm in the x axis, 100 mm in the y axis, and 50 mm in the z axis. Assume that you need to place three of these parts to result in the following assembly:

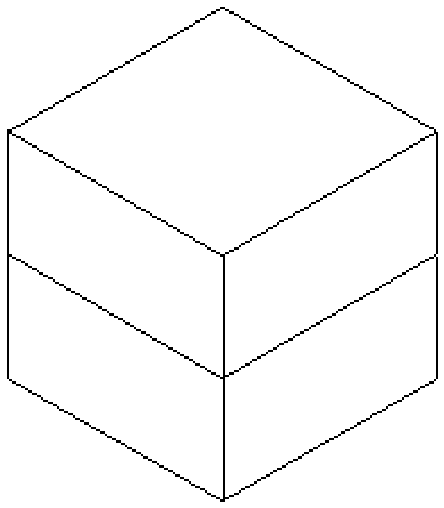


The following program creates this assembly:

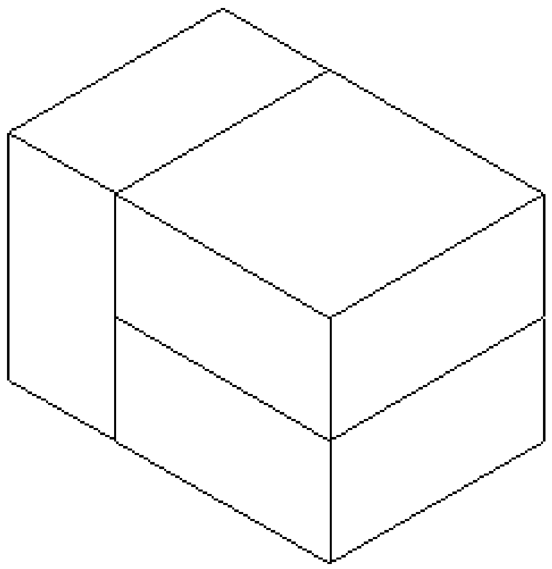
```
'Declare the program variables.
Dim objApp As Object
Dim objOccurrences As Object
Dim objTemp As Object

'Create constant (pi) for converting angles.
Const PI = 3.14159265358979

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Access the Occurrences collection object.
Set objOccurrences = objApp.ActiveDocument.Occurrences

'Place the first block.
Call objOccurrences.AddByFilename("Block.par")

'Place the second block.
Set objTemp = objOccurrences.AddByFilename("Block.par")

'It is currently in the same position and orientation as the first
'block, so reposition it.
Call objTemp.Move(0, 0, 0.05)

'Place the third block.
```

```
Set objTemp = objOccurrences.AddByFilename("Block.par")

'Rotate the third block to a vertical position.
Call objTemp.Rotate(0, 0, 0, 0, 1, 0, -PI / 2)

'Reposition the third block.
Call objTemp.Move(-0.075, 0, 0.025)
```

The following illustration shows the assembly after placing the second block and moving it into place:



And after placing the third block, rotating it, and moving it into place, the assembly is as follows:



This example positions the blocks using the Move method. You can also use the SetOrigin method, which is available on the Occurrence object, to move occurrences. SetOrigin works together with GetOrigin; GetOrigin returns the coordinates of the occurrence origin with respect to the assembly origin, and SetOrigin positions the occurrence's origin relative to the assembly's origin.

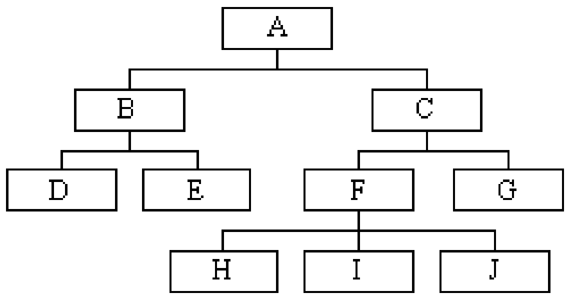
Using Occurrence Attributes

The following properties are available on Occurrence objects to return information about it and to set its characteristics:

- Locatable—Determines whether the Occurrence can be located interactively in the assembly. The other properties set and retrieve information about the Occurrence.
- OccurrenceDocument—Returns the COM Automation object for the document. This object can be either a PartDocument (for a part) or an AssemblyDocument (for a subassembly).
- Subassembly—Indicates if the occurrence is a subassembly.

Some other supported properties include OccurrenceFileName, Quantity, ReferenceOnly, and Status. For information on these properties, see the Programming with Solid Edge on-line Help file.

The following example shows how to use the OccurrenceDocument and Subassembly properties. Using recursion, this program returns all of the components of an assembly no matter how many levels or parts it contains. The program finds every part and subassembly in an assembly and ensures that its display is turned on. To understand the program logic, assume that your assembly is structured as follows:



The first subroutine, cmdDisplay_Click, starts the process by calling the subroutine DisplayOn and passing in the current file (Part A) as input. DisplayOn iterates through the parts in A, which are B and C. It determines whether B is a subassembly, checks if its display is on, and calls the subroutine DisplayOn, passing in B as input. DisplayOn iterates through the parts in subassembly B. In this case, they are not subassemblies so each one is checked to make sure its display is on and then the subroutine is exited. This causes program control to return to where DisplayOn is processing the parts in A. It has finished B and now moves to C where the process continues.

Using recursion, this simple program traces through each subassembly to the individual parts. This technique can be used whenever you need to access each part in an assembly.

```
Private Sub Command1_Click()
  Dim objApp As Object
```

```
'Connect to a running instance of Solid Edge.
  Set objApp = GetObject(, "SolidEdge.Application")

'Call the function with the current document as input.
  Call DisplayOn(objApp.ActiveDocument)
End Sub


Public Sub DisplayOn(Document As Object)
  'Declare variables.
  Dim objOccurrences As Object
  Dim objOccurrence As Object

  'Reference the Parts collection object.
  Set objOccurrences = Document.Occurrences

  'Iterate through each part in the current document.
  For Each objOccurrence In objOccurrences
  'Check to see if the current attachment is
  'a subassembly.
  If objOccurrence.Subassembly Then
    'Turn on the display of the subassembly.
  objOccurrence.Visible = True
    'Call this function with the subassembly as input.
    Call DisplayOn(objOccurrence.OccurrenceDocument)
  Else
    'Turn on the display of the part.
    objOccurrence.Visible = True
  End If
  Next
End Sub
```

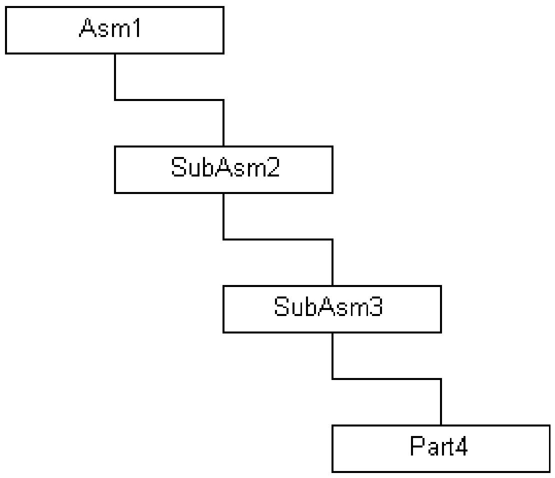
Working with Assembly Reference Objects

When you work with an assembly document interactively, you can work directly with occurrences and part geometry in subassemblies. For example, you can place relationships between occurrences nested in subassemblies, you can measure distances between faces of occurrences in subassemblies, you can in-place-activate an occurrence within a sub-assembly, and you can apply face styles to occurrences within subassemblies.

Because you can use the Occurrences collection to access occurrences nested in subassemblies, and because you can access the OccurrenceDocument representing a PartDocument and access geometry within the part, it appears simple to use the automation interface to work with occurrences in subassemblies just as you would through the graphical user interface. However, this appearance is deceptive. When you work with occurrences in subassemblies, and when you work with geometry of parts in occurrences (however deeply nested), use the Reference object to create references to part geometry and to nested occurrences from the top-level assembly. Then use the Reference object to place relationships, measure distances, in-place-activate nested occurrences, apply face styles, and so forth.

You can create Reference objects with the AssemblyDocument.CreateReference method. This method has two input parameters: an occurrence (which must be an Occurrence object), and an entity, which can be one of several different types of objects.

Consider the following assembly structure:



Part4 is a member of SubAsm3, SubAsm3 is a member of SubAsm2, and SubAsm2 is a member of Asm1.

Suppose you want a Reference object that represents Part4 in Asm1 (a top-level assembly reference to Part4). You must first find the Occurrence object corresponding to Part4. The AssemblyDocument has an Occurrences collection that represents all of the parts and sub-assemblies in the top-level assembly. So there is an Occurrence in Asm1 that represents SubAsm2.

```
Dim objSubAsm2Occurrence As Occurrence
Set objSubAsm2Occurrence = objAsm1.Occurrences(1)
```

If an Occurrence represents a subassembly, then you can access the subassembly document with the OccurrenceDocument method of the Occurrence object. In this case, this method will return objSubAsm3.

```
Dim objSubAsm3 As AssemblyDocument
Set objSubAsm3 = objSubAsm2Occurrence.OccurrenceDocument
```

This document will also have an Occurrences collection. By recursively accessing the AssemblyDocument represented by each nested Occurrence object, you can navigate down the assembly structure to the Occurrence representing any deeply-nested part (objPart4Occurrence in the example).

To create a Reference object in the top-level Assembly document, go back up this tree, creating Reference objects as you go. First, in the most deeply-nested subassembly document that contains the Occurrence you want, create a Reference to the part. The object supporting the CreateReference method is the AssemblyDocument that owns the Occurrence object whose OccurrenceDocument (a PartDocument) supports the part model (a Model object). The first input parameter is objPart4Occurrence. In this case, the second input parameter is the Model object in objPart4. In this example, objPart4 is a PartDocument. However, you've found it as on an Occurrence object, so get to the PartDocument through this Occurrence:

```
Dim objPart4Model as Model
objPart4Model =
objPart4Occurrence.OccurrenceDocument.Models(1)
```

Now you have enough information to create a Reference object:

```
Dim objPart4Ref as Reference
objPart4Ref =
objSubAsm3.CreateReference(objPart4Occurrence,
objPart4Model)
```

Unfortunately, this Reference object only makes sense to SubAsm3 (it is the road map to Part4 within SubAsm3. However, you can introduce this Reference to SubAsm2 by creating a Reference to it within SubAsm2:

```
Dim objP4RefRef as Reference
Set objP4RefRef =
objSubAsm2.CreateReference(objSubAsm3Occurrence,
objPart4Ref
```

One more step, and Asm1 recognizes the reference to Part4:

```
Dim objP4RefRefRef as Reference
Set objP4RefRefRef =
objAsm1.CreateReference(objSubAsm2Occurrence, objP4RefRef)
```

Analyzing Existing Assembly Relationships

When interactively placing parts in an assembly, you define relationships between parts to control their relative positions. Using the automation interface for the Assembly environment, you can access and modify properties of the assembly relationships.

Relationship objects are accessible through two collections: Relations3d on the AssemblyDocument object and Relations3d on each Part object. The Relations3d collection on the AssemblyDocument allows you to iterate through all relationships in the document. The Relations3d collection on each Part object allows you to iterate through the relationships defined for that specific part.

There are five types of 3-D relationships: AngularRelation3d, AxialRelation3d, GroundRelation3d, PlanarRelation3d, and PointRelation3d. These do not directly correlate to the interactive commands that place relationships. The relationships are as follows:

- AngularRelation3d—Defines an angular relationship between two objects.
- AxialRelation3d—Defines a relationship between conical faces. This is an axial align in the interactive interface.
- GroundRelation3d—Defines a ground constraint.
- PointRelation3d—Applies a connect relationship between points (vertices) of the points in an assembly.
- PlanarRelation3d—Defines a relationship between two planar faces. This includes both mates and planar aligns.

The following example shows how to use some of these relationship objects. This sample finds all of the PlanarRelation3d objects that define mates and modifies their offset values.

```
'Declare the program variables.
Dim objApp As Object
Dim Offset As Double
Dim objRelations As Object
Dim objRelation As Object

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Set the value of the offset.
Offset = 0.05

'Access the Relations3d collection.
Set objRelations = objApp.ActiveDocument.Relations3d

'Iterate through each relationship in the collection.
For Each objRelation In objRelations

  'If the relationship is a PlanarRelation3d, then
  If objRelation.Type = igPlanarRelation3d Then

    'Check to see if it is align relationship.
  'If it is an align relationship, do nothing.
```

```
    If objRelation.NormalsAligned Then
    Else
    'Otherwise, apply the offset.
    objRelation.Offset = objRelation.Offset + Offset
    End If
  End If
Next
```

Adding New Assembly Relationships

There are five methods to define assembly relationships through the automation interface: AddAngular, AddAxial, AddGround, AddPlanar, and AddPoint. These do not exactly correspond with the assembly relationship commands that are available interactively. However, they do correspond to the relationships that the interactive commands create.

For example, the AddPlanar method can be used to create either a Mate or an Align. The inputs to the AddPlanar method are two reference objects which are described below (but they correspond to the faces being mated or aligned), a Boolean that specifies whether or not the normals to the faces are aligned (this determines whether the faces are mated or aligned), and constraining points on each face (that correspond to the locations where you would click to locate the faces when you work interactively).

The following sample demonstrates the AddAxial method. This produces the same relationship that the interactive Align command produces when you align cylindrical faces. The inputs to this method are similar to those for the AddPlanar method. The first two inputs are reference objects that represent the cylindrical faces being aligned, and the third input is the Boolean that specifies whether normals to these faces are aligned. This method does not have input parameters for the constraining points the AddPlanar method uses.

To programmatically create the relationships that the Insert interactive command creates, you would use the AddPlanar and AddAxial methods. This would define the two cylindrical faces whose axes are aligned, and it would define the two planar faces that are mated. To remove the final degree of freedom, you would edit the axial relationship and set its FixedRotate property to True.

To create a Connect relationship, use the AddPoint method. The first input parameter is a reference object corresponding to the face or edge on the first part; the second input parameter is a constant that defines which key point from the input geometry defines the connection point (for example, CenterPoint, EndPoint, MidPoint, and so forth); and the third and fourth input parameters describe the same characteristics of the second part.

Within this general description, there are some important refinements. The methods previously described refer to reference objects, which correspond to part geometry. Each Assembly relationship must store a means of retrieving the geometric Part information that defines it. When using the AddPlanar method, for example, you need to pass in references to two planar faces (or reference planes).

The AssemblyDocument object has a CreateReference method whose job is to create the reference objects. The CreateReference method takes as input an Occurrence (an object that represents a member document of the assembly—which in this case would be a part document) and an Entity. The Entity can be an Edge, Face, or RefPlane object from the Occurrence document. The Reference object stores a path to the geometric representations necessary to construct the relationships.

To create assembly relationships via the automation interface, Occurrence objects (the Part and Subassembly models that comprise the assembly) must be placed in the Assembly document. You do this with the AssemblyDocument.Occurrances.AddByFilename method. This places the Occurrence in the assembly with a ground relationship. Therefore, (except for the first Occurrence added to the assembly) before any other relationships can be applied between this Occurrence and others in the assembly, the ground relationship must be deleted.

```
 Dim objApp As Application
 Dim objDoc As AssemblyDocument
 Dim objScrew As Occurrence
 Dim objScrewConicalFace As Face
 Dim objReferenceToConeInScrew As Reference
 Dim objNut As Occurrence
 Dim objNutConicalFace As Face
 Dim objReferenceToConeInNut As Reference
 Dim objFace As Face
 Dim objFaces As Object
 Dim objGroundRel As GroundRelation3d
 Dim objRelNuttoScrew As AxialRelation3d

 Set objApp = GetObject(, "SolidEdge.Application")
 Set objDoc = objApp.ActiveDocument

 Set objScrew = objDoc.Occurrences.AddByFilename("Screw.par")
 Set objFaces =
objScrew.OccurrenceDocument.Models(1).RevolvedProtrusions(1).SideFa
ces
 For Each objFace In objFaces
   If objFace.Geometry.Type = igCone Or objFace.Geometry.Type =
igCylinder Then
     Set objScrewConicalFace = objFace
     Exit For
   End If
 Next objFace

 Set objReferenceToConeInScrew =
objDoc.CreateReference(Occurrence:=objScrew,
entity:=objScrewConicalFace)

 Set objNut = objDoc.Occurrences.AddByFilename("Nut.par")
 Set objFaces =
objNut.OccurrenceDocument.Models(1).RevolvedCutouts(1).SideFaces
 For Each objFace In objFaces
   If objFace.Geometry.Type = igCone Or objFace.Geometry.Type =
igCylinder Then
     Set objNutConicalFace = objFace
     Exit For
   End If
 Next objFace

 Set objReferenceToConeInNut =
objDoc.CreateReference(Occurrence:=objNut,
entity:=objNutConicalFace)

 'All Occurrences placed through automation are placed "Grounded."
You must
 'delete the ground constraint on the second Occurrence before you
 'can place other relationships.
 Set objGroundRel = objDoc.Relations3d.Item(2)
 Call objGroundRel.Delete

 'Rather than passing literal axes to the AddAxial method, pass
```

```
references to conical faces,
 'just as you select conical faces when you use the interactive
Align command.
 Set objRelNuttoScrew =
objDoc.Relations3d.AddAxial(objReferenceToConeInNut,
objReferenceToConeInScrew, False)
```

C H A P T E R 12

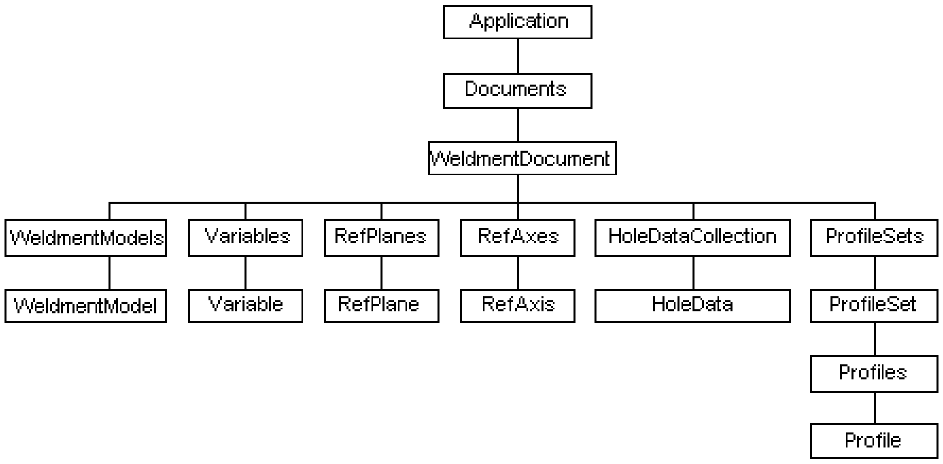
Working with Weldments

This chapter introduces the automation interface of the Solid Edge Weldment environment.

Weldment Document Anatomy

Weldment documents use assembly documents as input to represent weldment features. Two methods on the WeldmentModels automation object allow you to create an insert weldment feature. One requires an assembly document as input, while the other requires an assembly file name as input. Both of these methods have optional parameters that can specify which parts of the input assembly are to be included in the weldment. Both of these methods return an AssemblyWeldment object that represents an insert weldment feature that is created from an assembly. This object supports methods that allow the insert weldment feature to be edited, such as changing the assembly parts that are to be included in the weldment. This object also supports other methods and properties, such as one that updates the weldment based on the current state of its parent assembly.

A simplified weldment automation model is as follows:



C H A P T E R 13

Working with Draft Documents

This chapter describes the automation interface of the Solid Edge Draft environment.

Draft Document Anatomy

The structure of Draft documents differs significantly from other Solid Edge document types. From the DraftDocument object, you access the Sheets collection and then the individual Sheet objects. The Sheets collection contains both working sheets and background sheets.

In addition, DraftDocument supports a Sections object. The Sections object is a collection of Section objects that group Sheets by the characteristics of the data they contain. As users create drawings interactively, data from these drawings is automatically placed in one of three Section objects:

- Section1—Contains Sheet objects (Working Sheets) on which normal 2-D drawing objects (Lines, Arcs, DrawingViews, and so forth) are placed.
- Backgrounds—Contains background sheets, which hold the sheet borders.
- DrawingViews—Contains the Sheet objects on which the 2-D geometry of DrawingView/DraftView objects is placed. For each DrawingView/DraftView object, there is a separate sheet in the DrawingViews section.

Sections are a part of the graphical interface, although they are not immediately apparent. When the interactive user selects View > Background Sheet, Solid Edge internally changes to the Backgrounds section and displays its sheets. Similarly, the View > Working Sheet command allows you to modify the sheets that are in the Sections1 section. When a DrawingView is added, a new sheet is added to the DrawingViews section.

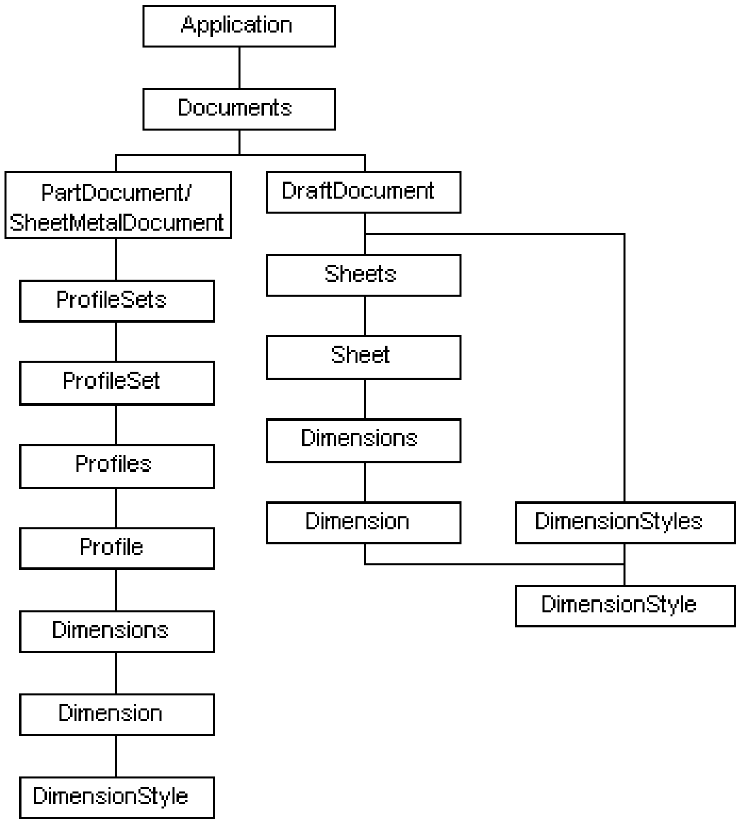
However, it is not possible through the graphical interface to create and manipulate sections directly. Although it is possible through automation to create new Sections, it is not a supported workflow. Although the same information is available on the Sheets collection that is a child of the DraftDocument object, within Sections, the information is separated by its functional characteristics.

Working with Dimensions—Overview

Solid Edge allows you to place and edit dimensions on elements. In the Draft environment, dimension objects primarily communicate characteristics such as size, distance, and angle. In the Profile environment, dimension objects control the size and orientation of geometry.

Dimensions can be linear, radial, or angular. Dimensions supply information about the measurements of elements, such as the angle of a line or the distance between two points. A dimension is related to the element on which it is placed. In the Draft environment, if an element on which a dimension is placed changes, the dimension updates. In the Profile environment, the dimensions control the geometry; if the dimension changes, the geometry updates.

The object hierarchy for dimensions is as follows:



In a Part document, the Dimensions collection is accessed through the Profile object. In the Draft environment, the Dimensions collection is accessed through the Sheet object. The Dimensions collection provides the methods for placing dimensions and for iterating through all the dimensions on the entire sheet or profile.

In a Draft document, the DimensionStyles collection on the document provides the methods for adding dimension styles and for iterating through all the dimension styles in the document.

Linear Dimensions

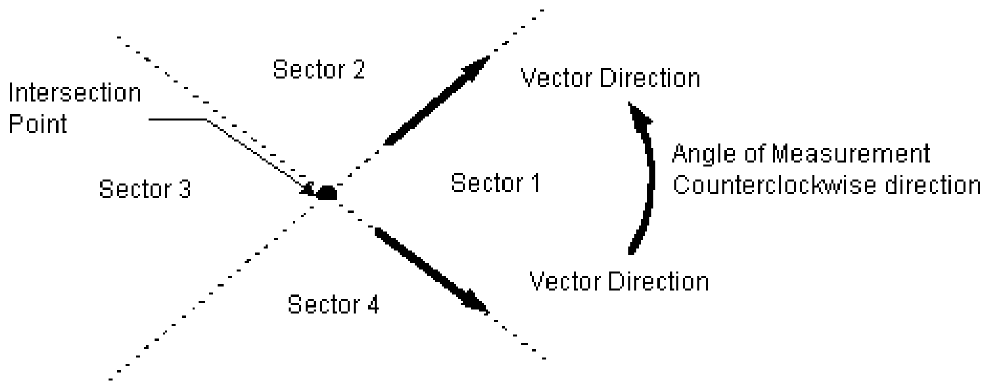
A linear dimension measures the distance between two or more elements, the length of a line, or an arc's length. For a complete description of the properties that define how a linear dimension is placed, see the Programming with Solid Edge on-line Help.

Radial Dimensions

Radial dimensions measure the radius or diameter at a point on the element. These dimensions are similar except that they show the radius or diameter value depending on the type. With the ProjectionArc and TrackAngle properties, you can define the measurement point on the element. For a complete description of the properties, see the Programming with Solid Edge on-line Help.

Angular Dimensions

Angular dimensions measure the angle between two lines or three points. An angular dimension defines two intersecting vectors and four minor sectors. These sectors are distinguished according to whether the angle is measured in the sector where the vector direction goes outward from the intersection point or comes inward, and whether the angle is measured in the clockwise or counterclockwise direction.



The angles are always measured in the counterclockwise direction with both vector directions going outward from the intersection point (sector one condition). To measure in any other angle, certain properties are set so that the dimension object modifies the vector direction and computes the angle.

Placing Dimensions

Two techniques are available for placing dimensions:

- Placing driven dimensions (Draft environment)—Driven dimensions are controlled by the graphic elements to which they refer. If the element changes, the dimensional value updates. A driven dimension measures (that is, documents) the model. You can override the value of a driven dimension by setting the OverrideString.
- Placing driving dimensions (Profile environment)—Driving dimensions control the elements to which they refer. When you edit a driving dimension, the geometry of the element that is related to that dimension is modified.

You can place dimensions only on existing elements. A set of Add methods is provided on the Dimensions collection, one for each type of dimension. The element to which the dimension is attached determines the type of dimension (driving or driven) that will be placed. The Add methods on the Dimensions collection object take minimal input and place the dimensions with specific default values. For a complete description of the add methods and properties available for setting the default values, see the Programming with Solid Edge on-line Help.

When you place dimensions between two elements interactively, the dimensions are measured at a specific location on an element. For example, when you place a dimension between the end points of two lines, you select one end of each line. When you place dimensions through automation, you specify a point on the element and a key point flag to define the dimension.

In the following program, four lines are drawn and connected with key point relationships. The lengths of two of the lines and the distance between two lines are dimensioned. The dimension is set to be a driving dimension so it will control the length and position of the geometry. The sample also shows how to modify a dimension style by changing the units of measurement of one of the dimensions to meters.

```
 'Declare the program variables.
 Dim objApp As Object
 Dim objProfile As Object
 Dim objProfileSets As Object
 Dim objLines As Object
 Dim objRelations As Object
 Dim objDimensions As Object
 Dim L(1 To 4) As Object
 Dim A(1 To 4) As Object
 Dim D1 As Object

 'Connect to a running instance of Solid Edge.
 Set objApp = GetObject(, "SolidEdge.Application")

 'Make sure that the active environment is Profile. Exit if it is
not.
 If objApp.ActiveEnvironment <> "Profile" Then
   MsgBox "This macro must be run from the Profile environment."
   End
 End If
```

```
'Reference the profile on which to place the geometry.
Set objProfileSets = objApp.ActiveDocument.ProfileSets
Set objProfile = objProfileSets(objProfileSets.Count).Profiles(1)

'Reference the collections used.
Set objLines = objProfile.Lines2d
Set objRelations = objProfile.Relations2d
Set objDimensions = objProfile.Dimensions

'Draw the geometry.
Set L(1) = objLines.AddBy2Points(0, 0, 0.1, 0)
Set L(2) = objLines.AddBy2Points(0.1, 0, 0.1, 0.1)
Set L(3) = objLines.AddBy2Points(0.1, 0.1, 0, 0.05)
Set L(4) = objLines.AddBy2Points(0, 0.05, 0, 0)

'Add endpoint relationships between the lines.
Call objRelations.AddKeypoint(L(1), igLineEnd, L(2), igLineStart)
Call objRelations.AddKeypoint(L(2), igLineEnd, L(3), igLineStart)
Call objRelations.AddKeypoint(L(3), igLineEnd, L(4), igLineStart)
Call objRelations.AddKeypoint(L(4), igLineEnd, L(1), igLineStart)

'Add dimensions, and change the dimension units to meters.
Set D1 = objDimensions.AddLength(object:=L(2))
D1.Constraint = True
D1.Style.PrimaryUnits = igDimStyleLinearMeters

Set D1 = objDimensions.AddLength(object:=L(4))
D1.Constraint = True
D1.Style.PrimaryUnits = igDimStyleLinearMeters

Set D1 = objDimensions.AddDistanceBetweenObjects( _
  Object1:=L(2), X1:=0.1, Y1:=0.1, z1:=0, _
  KeyPoint1:=False, _
  Object2:=L(3), X2:=0, Y2:=0.05, z2:=0, _
  KeyPoint2:=False)
D1.Constraint = True
D1.Style.PrimaryUnits = igDimStyleLinearMeters
End
```

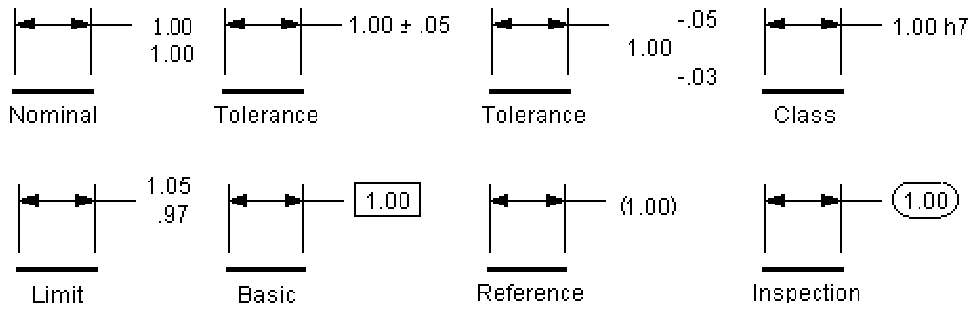
Displaying Dimensions

Dimensions can measure elements in different ways and are positioned relative to the elements being measured. When you create a dimension, default values are set for the properties to position the dimensions properly. Most of these default values are derived from the dimensional style associated with the dimension. In addition, the following values are set:

- BreakDistance is set to 0.5, which centers the dimension text.
- BreakPosition is set to igDimBreakPositionCenter.
- TrackDistance is set to be 10 times the dimension text size in case of angular dimensions. In all other cases, it is set to two times the dimension text size.

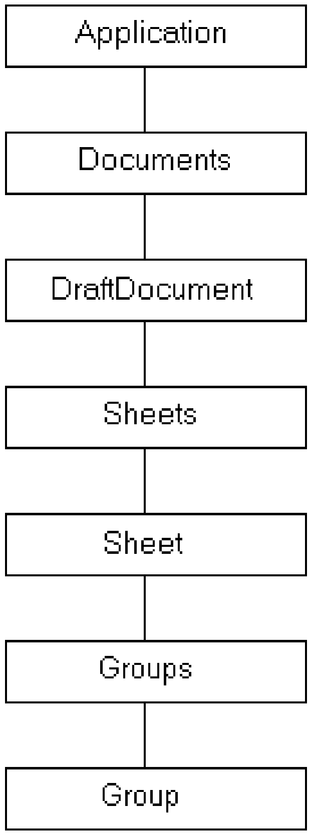
The DisplayType Property

Dimension values can be displayed in different ways. The DisplayType property and the Tolerance value allow you to set the display you need.



Working with Groups—Overview

A Group object binds elements, such as 2D geometry objects and dimensions, on a drawing sheet. You can then locate, select, and manipulate the elements as a unit. Grouped elements are usually related, such as the holes and center lines of a bolt hole pattern. The hierarchy for the Group object is as follows:



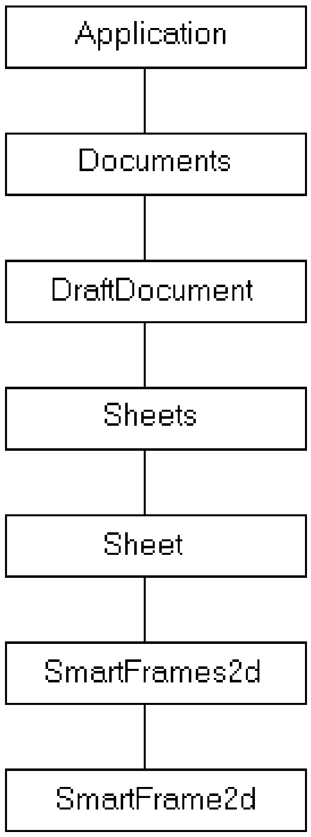
Working with SmartFrames—Overview

SmartFrames are shapes (rectangles or ellipses) on a sheet that enclose embedded or linked object(s) and have some intelligence about how to deal with the data in that frame. SmartFrames provide control over the way automation objects are displayed and manipulated on a Solid Edge sheet. SmartFrames have intelligence about their contained objects that includes the following features:

- A transformation matrix to convert between the framed object’s local coordinate system and the containing document’s coordinate system.
- Methods to manipulate the contained object, such as scale, crop, move, or rotate.
- Frame symbology that shows the state of the framed object such as linked, embedded, or a link that needs updating.
- Link update rules (such as automatic and manual).
- In-place activation rules.
- Knowledge about favorite commands.
- Knowledge about a preferred file location or extension used in first associating the file to a frame.
- Knowledge for converting between links, embeddings, and native data.

When using Solid Edge, you may sometimes find it useful to reference data that exists in a format other than a Solid Edge file. For example, while in the Solid Edge drawing environment, you might want to link to a portion of a Microsoft Excel spreadsheet. Solid Edge supports this cross-referencing through the implementation of SmartFrames. A SmartFrame is a Solid Edge object that contains a view of an embedded or linked object.

The object hierarchy for SmartFrames is as follows:


Application
Documents
DraftDocument
Sheets
Sheet
SmartFrames2d
SmartFrame2d

Sample Program—Creating a SmartFrame

Initially, you can create an empty SmartFrame without specifying an object to be linked or embedded. A SmartFrame style must be specified, or you can use the default style for a sheet. A SmartFrame style has properties that affect how the object within the SmartFrame can be manipulated. For example, a SmartFrame that is based on a reference file style can either align the origin of the external file with the Solid Edge file or provide an option to scale the contents.

When you create a SmartFrame, four solid black lines are drawn to represent the frame. Once you have created the SmartFrame, you can select and manipulate the object as you would other Solid Edge objects.

You can create and manipulate SmartFrame objects through the automation interface using the methods that are associated with the SmartFrames2d collection object. In the following example, the AddBy2Points method creates a SmartFrame. The first argument of AddBy2Points specifies a style to be applied to the SmartFrame. In this case, the style is set to a blank string (""), so the default style is applied.

```
 'Declare the program variables.
 Dim objApp As Object
 Dim objSFrames As Object
 Dim objSFrame As Object

 'Connect to a running instance of Solid Edge.
 Set objApp = GetObject(, "SolidEdge.Application")

 'Reference the SmartFrames2d collection.
 Set objSFrames = objApp.ActiveDocument.ActiveSheet.SmartFrames2d

 'Create a SmartFrame2d object by two points.
 Set objSFrame = objSFrames.AddBy2Points("", 0.02, 0.02, 0.07,
0.07)

 'Add a description to the SmartFrame.
 objSFrame.Description = "myframe"
```

You can also use the AddByOrigin method to create a SmartFrame object. With AddByOrigin, you specify an origin and offset parameters for the top, bottom, left, and right sides of the frame.

Sample Program—Linking and Embedding

You can link or embed objects to existing SmartFrames using the CreateEmbed method. The syntax for this method is as follows:

```
Call SmartFrame.CreateEmbed("c:\temp\myfile.doc")
```

You can also link to a document with the CreateLink method.

```
Call SmartFrame.CreateLink("c:\temp\myfile.doc ")
```

In the following example, the program searches all of the members of the SmartFrames collection, looking for the SmartFrame with the Description of "myframe." Once this object is found, CreateEmbed is called to embed the specified file (in this case, c:\temp\myfile.doc).

```
Dim objApp As Object
Dim objSFrames As Object
Dim objSFrame As Object
Dim NumFrames As Integer

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Reference the SmartFrames2d collection.
Set objSFrames = objApp.ActiveDocument.ActiveSheet.SmartFrames2d

'Set NumFrames equal to the Count property of objSFrames.
NumFrames = objSFrames.Count

'Return the frame with the description "myframe".
If NumFrames > 0 Then
  For I = 1 To NumFrames
    Set objSFrame = objSFrames.Item(I)
  If objSFrame.Description = "myframe" Then
  Exit For
  End If
  Next I
End If

'Embed document within the identified SmartFrame.
Call objSFrame.CreateEmbed("c:\temp\myfile.doc")
```

Manipulating SmartFrames

Once you have linked or embedded data, there are several ways through automation to manipulate a SmartFrame. For example, you can cut, copy, and paste the data. You can also edit the contained object, change the SmartFrame styles and properties, and perform other operations. Property values that are set for the SmartFrame style determine what types of manipulations are permitted. For the Visual Basic user, many properties are accessible on the SmartFrame object itself that control the SmartFrame. For example, the following syntax shows how to change the size of the SmartFrame, make its contents visible, and ensure that it cannot be selected.

```
Call SFrame.ChangeCrop(0.05, 0.0, 0.0, 0.07)
SFrame.ContentsVisible = True
SFrame.ProtectFromSelection = True
```

Another related property is the Object property. This property returns the object that is contained in the SmartFrame. For example,

```
Dim activexObject as Object
Set activexObject = SFrame.Object
```

If the object is a type that supports its own automation interface, you can call its native methods and properties. For example, if the object is an Excel spreadsheet, you can call properties and methods exposed for Excel spreadsheet objects.

Sample Program—Using SmartFrame Styles

When you create a SmartFrame in Solid Edge, you also have the option to specify a style to be associated with that frame. With most Solid Edge objects, the SmartFrame style determines characteristics such as color and line weight.

A SmartFrame style can also determine certain behaviors of the SmartFrame. For example, you can create a SmartFrame style that specifies a default command for the contained object, determines that the contained object is read-only, and specifies that the user cannot move the SmartFrame. You can also create specific styles for specific types of objects. For example, you can create a SmartFrame style for linking or embedding data from an Excel spreadsheet.

You can retrieve existing SmartFrame styles from the SmartFrame2dStyles collection object accessed from the Document object. The following syntax shows how to locate a specific SmartFrame style object from the collection based on its Name property:

```
 'Declare the program variables.
 Dim objApp As Application
 Dim objSFStyleCollection As Object
 Dim objSFStyle As Object
 Dim NumStyles As Integer
 Dim I As Integer


 'Connect to a running instance of Solid Edge.
 Set objApp = GetObject(, "SolidEdge.Application")

 'Reference the SmartFrames2dStyles collection.
 Set objSFStyleCollection =
objApp.ActiveDocument.SmartFrame2dStyles

 'Set NumStyles equal to the Count property of the collection
object.
 NumStyles = objSFStyleCollection.Count

 'Find a specific style object by name.
 If NumStyles > 0 Then
   For I = 1 To NumStyles
     Set objSFStyle = objSFStyleCollection.Item(I)
   If objSFStyle.Name = "Style1" Then
   Exit For
   End If
   Next I
 End If

 'Create a new SmartFrame style using the Add method as follows:
 If NumStyles < 1 Then
   Set objSFStyle = objSFStyleCollection.Add("Style1", "parent")
 End If
```

You can use the SmartFrame style name when you create a SmartFrame object to associate a style with a SmartFrame object. The following syntax creates a SmartFrame object with a specific style:

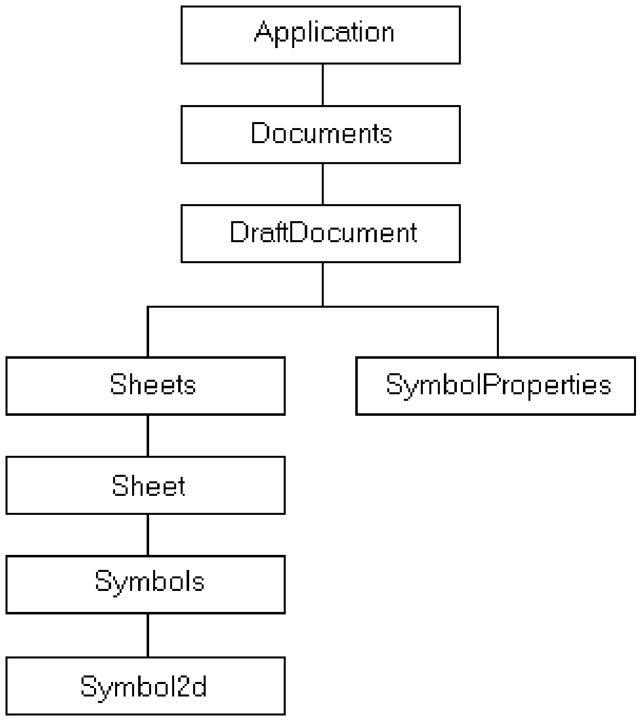
```
Set SFrame = SFrameCollection.AddBy2Points(SFStyle.Name, 0.02,
0.02, 0.07, 0.07)
```

Working with Symbols—Overview

Symbols are documents that contain graphic elements. You can place these documents at a specified scale, position, and orientation. The document that contains the graphic elements is the *source* document; the document into which the source is placed is the *container* document. A source document is represented in a container document by a symbol. The symbol references the source document as the COM object. Using symbols, you can store a drawing of a nut, bolt, or screw in one document and place it in several documents at a user-defined size. In addition, symbols have the following benefits:

- Save memory when placing multiple instances of the same source document in the same container.
- Automatically update the container document when modified.
- Maintain the properties defined in the source document.

On the Insert menu, click Object to place a symbol in the interactive environment. When using Solid Edge though automation, you can place a symbol using the methods associated with the Symbols collection. The object hierarchy for symbols is as follows:



The Symbols collection object provides methods that enable you to place new symbols and to query for information about existing ones. The Symbol2d object provides methods and properties to enable you to review or manipulate the symbol geometry, the attachment between the symbol and the source document, and the user properties. You can also move and copy symbols.

Creating Symbol Source Documents

You can place a symbol from any source document that is implemented as an ActiveX object. For example, a source document could be a Microsoft Word file, an Excel spreadsheet, or a Solid Edge document.

Sample Program—Placing Symbols

When you place a symbol, you must specify an insertion type. The insertion type affects the way the symbol is updated. Three options are available:

- Linked—The symbol and the initial source document are directly connected. The symbol is automatically updated when its source document is edited. The source document is external to the container. It is a separate file that is visible with Explorer.
- Embedded—A copy of the initial source document is stored in the container. The symbol is attached to this copy and is automatically updated when the copy is updated. After placement, the symbol is strictly independent of the initial source document.
- Shared Embedded—When placing a symbol more than one time into the same container, the initial source document is copied only one time. The symbols are attached to that copy and are all updated automatically when the copy of the initial source document is updated. After placement, the symbols are strictly independent of the initial source document.

The following program demonstrates the three ways to place a symbol:

```
'Declare the program variables.
Dim objApp As Object
Dim objSheet As Object
Dim objSymbols As Object
Dim objSymbol1 As Object
Dim objSymbol2 As Object
Dim objSymbol3 As Object

'Turn on error checking.
On Error Resume Next

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")
If Err Then
  MsgBox "Solid Edge must be running."
  End
End If

On Error GoTo 0

'Make sure the active environment is Draft.
If objApp.ActiveEnvironment <> "Detail" Then
  MsgBox "You must be in the Drafting environment."
  End
End If

'Reference the active sheet.
Set objSheet = objApp.ActiveDocument.ActiveSheet

'Reference the Symbols collection.
Set objSymbols = objSheet.Symbols

'Create a linked symbol at location x=0.1, y=0.1.
Set objSymbol1 = _
  objSymbols.Add(igOLELinked, "c:\temp\test1.doc", 0.1, 0.1)

'Create an embedded symbol at location x=0.1, y=0.15.
Set objSymbol2 = _
```

```
    objSymbols.Add(igOLEEmbedded, "c:\temp\test2.doc", 0.1, 0.15)

 'Create a shared embedded symbol at location x=0.1, y=0.2.
 Set objSymbol3 = _
   objSymbols.Add(igOLESharedEmbedded, "c:\temp\test3.doc", 0.1,
0.2)
```

Sample Program—Moving and Rotating a Symbol

You can manipulate a symbol much as you would manipulate other objects and elements in a drawing. For example, you can manipulate a symbol by editing its properties or symbology or by using element manipulation commands such as Move, Copy, Scale, and so forth. When manipulated, the symbol is treated as a single element.

The following program manipulates the symbol geometry by referencing the origin of a symbol and setting a new origin. It also sets a new rotation angle.

```
'Declare the program variables.
Dim objApp As Object
Dim objSheet As Object
Dim objSymbols As Object
Dim objSymbol As Object
Dim x As Double
Dim y As Double

'Turn on error checking.
On Error Resume Next

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")
If Err Then
  MsgBox "Solid Edge must be running."
  End
End If

On Error GoTo 0

'Make sure the active environment is Draft.
If objApp.ActiveEnvironment <> "Detail" Then
  MsgBox "You must be in the Drafting environment."
  End
End If

'Reference the active sheet.
Set objSheet = objApp.ActiveDocument.ActiveSheet

'Reference the Symbols collection.
Set objSymbols = objSheet.Symbols

'Reference the first symbol in the collection.
Set objSymbol = objSymbols(1)

'Retrieve the origin of the symbol.
Call objSymbol.GetOrigin(x, y)

'Modify the symbol's origin.
objSymbol.SetOrigin x + 0.1, y + 0.1

'Set the angle of rotation to 45 degrees (in radians).
objSymbol.Angle = 45 * (3.14159265358979 / 180)
```

Sample Program—Retrieving Symbol Properties

The following program shows how to access the path and name of a linked symbol. It also shows how to access the Class property, which tells what type of file is referenced by the symbol.

```
'Declare the program variables.
Dim objApp As Object
Dim objSheet As Object
Dim objSymbols As Object
Dim objSymbol As Object
Dim InsertionType As Integer
Dim SourcePathName As String

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Make sure the active environment is Draft.
If objApp.ActiveEnvironment <> "Detail" Then
  MsgBox "You must be in the drafting environment."
  End
End If

'Reference the active sheet.
Set objSheet = objApp.ActiveDocument.ActiveSheet

'Reference the Symbols collection.
Set objSymbols = objSheet.Symbols

'Reference the first symbol in the collection.
Set objSymbol = objSymbols(1)

'Reference the OLE type.
InsertionType = objSymbol.OLEType

'Retrieve the attachment type.
If InsertionType = igOLELinked Then
  SourcePathName = objSymbol.SourceDoc
End If

'Display the type of file to a message box.
MsgBox "Symbol is " & objSymbol.Class & " File."
End
```

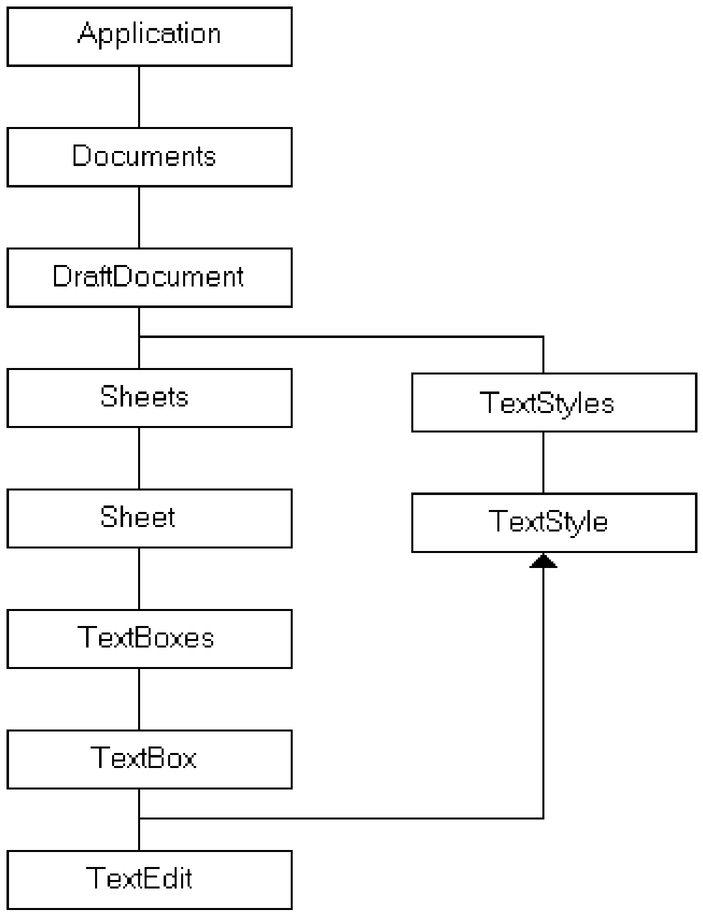
Sample Program—Accessing the Dispatch Interface of a Source Document

The following program shows how you can access the source document dispatch interface. The source document is modified by way of its automation interface. In this example, the symbol is a Microsoft Word document, so the syntax that modifies the content of the symbol is part of the Microsoft Word automation interface. This syntax varies depending on the type of document represented by the symbol. When the file is saved, the attached symbol is updated.

```
'Declare the program variables.
Dim objApp As Object
Dim objSheet As Object
Dim objSymbols As Object
Dim objSymbol As Object
Dim SourceDispatch As Object

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Make sure the active environment is Draft.
If objApp.ActiveEnvironment <> "Detail" Then
  MsgBox "You must be in the drafting environment."
  End
End If

'Reference the active sheet.
Set objSheet = objApp.ActiveDocument.ActiveSheet

'Reference the Symbols collection.
Set objSymbols = objSheet.Symbols

'Reference the first symbol in the collection.
Set objSymbol = objSymbols(1)

'Reference the source document dispatch interface.
Set SourceDispatch = objSymbol.Object

'Open the source document to modify it.
objSymbol.DoVerb igOLEOpen

'Add some additional text to the document.
SourceDispatch.Range.InsertParagraphBefore
SourceDispatch.Range.InsertBefore "New Text"

'Save and close the file.
SourceDispatch.Save
SourceDispatch.Close

'Exit Word.
SourceDispatch.Application.Quit
End
```

Working with Text—Overview

Solid Edge allows you to place and edit text boxes. Through the automation interface, this is handled by means of the TextBox object. The hierarchy for the TextBox object is as follows:



The object hierarchy shows the objects specific to text boxes. The TextBoxes collection on the sheet provides the methods for placing TextBox objects. It also allows for iterating through all the TextBox objects that exist on the sheet. The TextStyles collection on the document provides the methods for adding text styles and also allows you to iterate through all the text styles in the document. The TextEdit object provides methods to edit the text contained in a TextBox object.

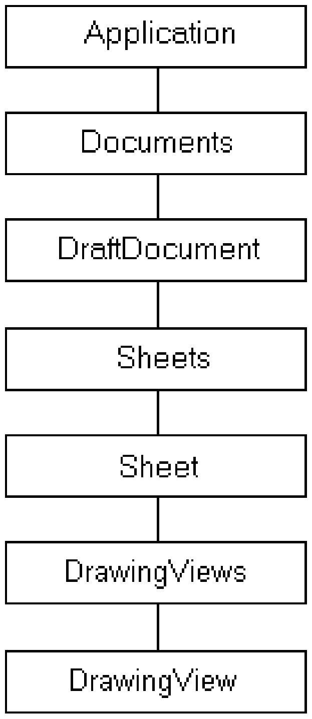
Sample Program—Placing a Text Box

Several add methods on the TextBoxes collection enable you to place a text box on a drawing sheet by specifying the origin, height, width, and/or rotation angle of the box. In the following example, the AddByTwoPoints method is used to create a TextBox object. Once the text box is created, text is added, and TextEdit is called to bold selected characters.

```
'Declare the program variables.
Dim objApp As Object
Dim objSheet As Object
Dim objTextBoxes As Object
Dim objTextBox1 As Object
Dim objTextEdit As Object
Dim X1 As Double, X2 As Double
Dim Y1 As Double, Y2 As Double

'Turn on error checking.
On Error Resume Next

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")
If Err Then
  MsgBox "Solid Edge must be running."
  End
End If

'Make sure the active environment is Draft.
If objApp.ActiveEnvironment <> "Detail" Then
  MsgBox "You must be in the Draft environment."
  End
End If

'Reference the Sheet object.
Set objSheet = objApp.ActiveDocument.ActiveSheet

'Reference the TextBoxes collection.
Set objTextBoxes = objSheet.TextBoxes

'Use the AddByTwoPoints method to create a text box.
'The points are defined by the X1, Y1, X2, Y2 variables.
'The z values of the point are defined as zero.
X1 = 0.02: Y1 = 0.02
X2 = 0.14: Y2 = 0.14

Set objTextBox1 = objTextBoxes.AddByTwoPoints _
  (X1, Y1, 0, X2, Y2, 0)

'Place the text in the text box.
objTextBox1.Text = "Testing TextBox in Solid Edge."

'Reference the new TextEdit object.
Set objTextEdit = objTextBox1.Edit

'Bold the word "TextBox."
'First, select the range of characters in the editor.
Call objTextEdit.SetSelect(8, 15, seTextSelectRange)

'Then set the bold flag.
objTextEdit.Bold = True
```

Working with Drawing Views—Overview

A DrawingView object is a 2-D representation of a 3-D part or assembly model. A drawing view is used to display design space geometry in document space. A view of design space is enclosed by the drawing view border (a handle that allows manipulation of the drawing view). Only one part or assembly document can be used as the basis for drawing views in a draft document. The hierarchy for the DrawingView object is as follows:



C H A P T E R 14

Managing Documents

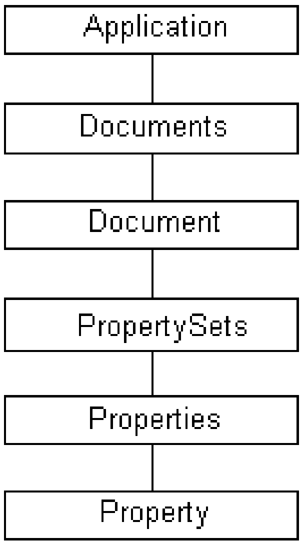
This chapter contains information to help you manage your documents with Solid Edge automation.

Working with Property Sets—Overview

Property sets are a mechanism for grouping and storing attribute information. This attribute information is made available to the end user in the following ways:

- Through an application's user interface.
- Through an automation interface.

The automation interface is provided through Solid Edge to allow end user programs access to the attribute information that is stored in a document. The object hierarchy for property sets is as follows:



There are three levels of objects for properties. The highest level, the PropertySets object, is a collection of property set objects. This collection object provides an index to the property sets stored in a document. The second level, the Properties object, is a representation of the property set object. This object is the parent object of all the properties stored in the property set. The third level, the Property object, represents the individual property stored in the property set.

To access a property, you traverse the object hierarchy starting at the document level. From the Document object, you can use the property named Properties, which actually returns a PropertySets object. Once you have access to this object, you can query for the number of property sets that are contained in the collection, the parent of the collection, the application which contains the collection, or a specific property set. Once the Properties object is located, you can access individual properties.

Solid Edge supports five property sets: summary information, extended summary information, project information, document information, and mechanical modeling. In addition, there is a Custom property set which gives access to all user-created properties defined through the Custom pane on the Properties dialog box.

Summary Information Property Set

The document summary information property set is the standard common property set. The stream name is SummaryInformation. The following properties are contained in this property set:

- Application name
- Author
- Comments
- Creation date
- Keywords
- Last author
- Last print date (not maintained)
- Last save date
- Number of characters (not maintained)
- Number of pages (not maintained)
- Number of words (not maintained)
- Revision number (not maintained)
- Security (not maintained)
- Subject
- Template
- Title
- Total editing time (not maintained)

Note Solid Edge does not update the values for the properties listed as "not maintained." You can, however, access these properties through automation and manually maintain their values.

Extended Summary Information Property Set

The extended summary information property set consists of properties that are of a summary or statistical nature, but that are not included in the Summary Information property set. The stream name is ExtendedSummaryInformation. The following properties are contained in this property set:

- CreationLocale
- Name of Saving Application

- Status
- Username

Project Information Property Set

The project information property set consists of project-related properties. The stream name is ProjectInformation. The following properties are contained in this property set:

- Document Number
- Project Name
- Revision

Document Summary Information Property Set

The document summary information property set consists of document-related properties. The stream name is DocumentSummaryInformation. The following properties are contained in this property set:

- Category
- Company
- Format (not maintained)
- Manager
- Number of bytes (not maintained)
- Number of hidden slides (not maintained)
- Number of lines (not maintained)
- Number of multimedia clips (not maintained)
- Number of notes (not maintained)
- Number of paragraphs (not maintained)
- Number of slides (not maintained)

Note Solid Edge does not update the values for the properties listed as "not maintained." However, you can access these properties through automation and manually maintain the values.

Mechanical Modeling Property Set

The mechanical modeling property set consists of mechanical modeling properties. The stream name is MechanicalModeling. The Material property is contained in this property set.

Sample Program—Accessing the Subject Property

The following example accesses the value of the Subject property from the summary information property set:

```
'Declare variables
Dim objApp As Object
Dim objDoc As Object
Dim objProp As Object
Dim objSummaryInfoPropSet As Object
Dim objSubjectProperty As Object

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Access the active document.
Set objDoc = objApp.ActiveDocument

'Initialize the PropertySets collection object.
Set objProp = objDoc.Properties

'Access the SummaryInformation properties collection
Set objSummaryInfoPropSet = objProp("SummaryInformation")

'Retrieve the Subject property.
Set objSubjectProperty = objSummaryInfoPropSet("Subject")

'Show the value of the property.
MsgBox "Subject is " & objSubjectProperty.Value & "."
```

Sample Program—Reviewing All Properties

The following example iterates through each Properties collection in the PropertySets object and displays every property in the document to the debug window.

```
'Declare the program variables.
Dim objApp As Object
Dim objDoc As Object
Dim objPropCollection As Object
Dim i As Integer

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Access the PropertySets collection object.
Set objPropCollection = objApp.ActiveDocument.Properties

'Iterate through each Properties object in the
'PropertySets collection.
For Each objProps In objPropCollection
  Debug.Print "Printing Properties " & objProps.Name

  'Iterate through each Property object in the
  'Properties collection.
  For i = 1 To objProps.Count
    Debug.Print " " & objProps(i).Name & "=" & objProps(i).Value
  Next i
  Debug.Print " "
Next
```

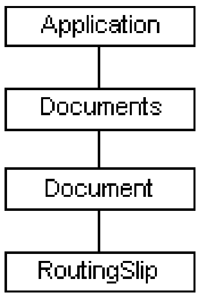
Working with Routing Slips—Overview

As users share information in workgroups and across enterprises, the need for simple, easy-to-use commands to route documents from one user to another over standard mail protocols is needed. Solid Edge allows you to create a routing slip for a document that specifies recipients of this document to be sent through electronic mail in a specified way. The interactive user can attach a routing slip by clicking Add Routing Slip on the File menu.

Routing slips allow you to distribute a document to either a distribution list or a series of reviewers and to have the document returned to you. For one-after-another routing, optional status messages keep the originator informed of the document progression. A routing slip is saved as part of the document; it requires an electronic mail system that is compliant with the Messaging Application Programming Interface (MAPI).

Routing slips can become even more powerful through the use of task automation. For example, you can compile the list of users for the routing slip from a database, such as Microsoft Access.

The hierarchy chart for the Routing Slip object is as follows:



The RoutingSlip object is a dependent of the Document object. For Solid Edge, this includes the PartDocument, SheetMetalDocument, AssemblyDocument, and DraftDocument objects.

Sample Program—Editing and Sending a Routing Slip

The following example shows how to access the RoutingSlip object, define information for the routing slip, and route it:

```
Dim objApp As Object
Dim objRoutingSlip As Object

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Access the RoutingSlip object of the active document.
Set objRoutingSlip = objApp.ActiveDocument.RoutingSlip

'Fill out the routing slip.
With objRoutingSlip
  .Recipients = Array("Melanie Baeske", "Melanie Baeske")
  .Subject = "Document for Review"
  .Message = "Review this document and add your comments"
  .Delivery = igOneAfterAnother
  .ReturnWhenDone = True
  .TrackStatus = True
  .AskForApproval = True

  'Route the document.
  .Route
End With
```

Sample Program—Approving a Document

The following syntax shows how a recipient can act on a document with a routing slip:

```
Dim objApp As Object
Dim objRoutingSlip As Object

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Access the RoutingSlip object for the active document.
Set objRoutingSlip = objApp.ActiveDocument.RoutingSlip

If objRoutingSlip.Status <> igInvalidSlip _
  And Not objRoutingSlip.HasRouted Then
  objRoutingSlip.Approve = True
  objRoutingSlip.Route
End If
```

C H A P T E R 15

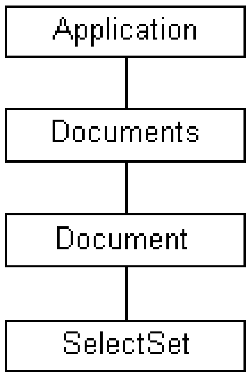
External Clients

This chapter contains information on using external clients with Solid Edge.

Working with Selection Sets—Overview

Solid Edge allows you to select multiple objects at one time. For example, you can select multiple features in the Part environment, or multiple parts and subassemblies in the Assembly environment. This temporary collection of objects is referred to as a selection set. Selection sets provide a way for an operation to be performed simultaneously on multiple elements.

Just as you can create a selection set interactively, you can also create one using automation. With automation, you use the SelectSet object to create a selection set. The object hierarchy for the SelectSet object is as follows:



Sample Program—Collecting Selected Objects

With a selection set, you can create commands that work in an object-action sequence. For example, when you delete a feature, you first select the feature and then perform the delete action. This is an object-action sequence because you first identify the objects and then perform an action on them.

The following syntax shows this methodology by using the selection set to collect selected objects. The Depth property of each object in the selection set is then modified to a common value. If a feature is selected that does not support the Depth property, a message box is displayed notifying the user, and the processing of the other features continues.

```
'Declare the program variables.
Dim objApp As Object
Dim objSelectSet As Object
Dim objFeature As Object

'Enable error handling.
On Error Resume Next

'Connect to running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")
If Err Then
  MsgBox "Solid Edge must be running."
  End
End If

'Access the SelectSet collection.
Set objSelectSet = objApp.ActiveDocument.SelectSet

'Use the Count property to make sure the selection
'set is not empty.
If objSelectSet.Count = 0 Then
  MsgBox "You must select the features to process."
  End
End If

'Process each feature in the selection set.
For Each objFeature In objSelectSet
  'Make sure the feature has a finite extent.
  If objFeature.ExtentType = igFinite Then
    'Change the depth of the feature.
  objFeature.Depth = 0.01
  End If

  If Err Then
    'Display and clear the error.
  MsgBox "Feature " & objFeature.Name & _
    " doesn’t support the depth property."
    Err.Clear
  End If
Next objFeature
```

Sample Program—Adding To/Removing From Selection Sets

The SelectSet collection object also supports methods to add and remove objects from the selection set as well as several methods that affect all objects in the selection set. These are the Delete method and two Clipboard-related methods: Copy and Cut.

The following syntax shows how to use these methods by adding all the round features in a model to the selection set. It then uses the Delete method to delete them from the model.

```
'Declare variables.
Dim objApp As Object
Dim objSelectSet As Object
Dim objRounds As Object
Dim objRound As Object

'Connect to running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Access the Rounds and SelectSet collections.
Set objRounds = objApp.ActiveDocument.Models(1).Rounds
Set objSelectSet = objApp.ActiveDocument.SelectSet

'Process each round in the Rounds collection.
For Each objRound In objRounds
  objSelectSet.Add objRound
Next

'Delete all the geometry in the selection set.
objSelectSet.Delete
```

Visual C++ Program Syntax

```
 // system includes
 #include <objbase.h>
 #include <comdef.h>
 #include <iostream.h>


 // Import all the Solid Edge type libraries. This will create a
.tli and .tlh file
 // for each of these type libraries, that will wrap the code to
call into the
 // methods and properties of the objects in the type libraries.

 #import "constant.tlb"
 #import "framewrk.tlb"
 #import "fwksupp.tlb"
 #import "geometry.tlb"
 #import "part.tlb"
 #import "assembly.tlb"
 #import "draft.tlb"

 // Use the typelib namespaces.

 using namespace SolidEdgeConstants;
 using namespace SolidEdgeFramework;
 using namespace SolidEdgeFrameworkSupport;
 using namespace SolidEdgeGeometry;
 using namespace SolidEdgePart;
 using namespace SolidEdgeAssembly;
 using namespace SolidEdgeDraft;


 // Error handling macro. Every function that calls this macro
needs to
 // have a label called "wrapup" declared, to jump to in case of
error.
 // This is where any cleanup should be done or resources freed
before
 // exiting the function.
 #define HandleError(hr, message) \
 \
 if FAILED(hr) \
 { \
   cerr << "Error: 0x" << hex << hr << dec << " Line: " << __LINE__
<< " Message: " << message << endl; \
   goto wrapup; \
 }\
 \




 // NOTES:
 // -----
 //
 // 1. "RunSEAutomation()" is a driver function that attaches to
 // an instance of Solid Edge and gets an interface pointer to the
 // Solid Edge application. The real client code that does anything
 // useful is encapsulated within this function. The "main()"
function
 // simply initializes COM, calls "RunSEAutomation()" and un-
initializes
 // COM. See the "Warning" section below for more information on
this.
 //
 // 2. We have chosen to use the easiet means of calling methods on
the
 // Solid Edge automation objects. While there are MFC
COleDispatchDriver
 // classes that can wrap IDispatch interfaces, it is significantly
 // easier to use the "#import" technique to import entire typelibs
into
 // client code, which automatically creates all the wrapper
classes and
 // their corresponding implementation of "IDispatch->Invoke()"
 // using smart pointers.
 //
 // 3. Some familiarity with COM smart pointers is assumed. For
more
 // information/help:
 // a) See topics "Compiler COM Support Classes" & "Compiler COM
 // Support: Overview" in MSDN.
 // b) See Visual Studio include file "comdef.h.".
 //
 // When you #import a typelib, VC++ automatically creates .tli and
 // .tlh files. The former define smart pointers for each and every
interface
 // defined in the typelib (both vtable and dispinterface), using
 // the "_COM_SMARTPTR_TYPEDEF" macro. If there is an interface of
type
 // "IFoo" in the typelib, the smart pointer associated with that
is
 // named "IFooPtr" (by adding "Ptr" to the interface name). The
 // smart-pointer implementation basically encapsulates the real
COM
 // interface pointers and eliminates the need to call the AddRef,
Release,
 // QueryInterface methods that all interfaces support. In
addition, they
 // hide the CoCreateInstance call for creating a new COM object.
Since
 // these smart pointers are also know the UUID for the interface
they
 // are wrapping, simply equating two smart pointers will call
 // QueryInterface on the interface on the right hand side for the
 // UUID of the interface supported by the smart pointer on the
left
 // hand side (much like VB).
 //
 //
 // For example the following code (error checking omitted) :
 //
 // {
 // IUnknown *pUnk;
 // IAnyInterface *pAnyInterface;
 // [some code to get pAnyInterface]
 // pAnyInterface->QueryInterface(IID_IUnknown, (LPVOID**) &pUnk);
 // pUnk->Release();
 // pAnyInterface->Release();
 // }
 //
 // can be replaced by
 //
 // {
 // IUnknownPtr pUnk;
 // IAnyInterfacePtr pAnyInterface;
 // [some code to get pAnyInterface]
 // pUnk = pAnyInterface; // does the QI internally on pUnk
 // } // destructors on smart pointers "pAnyInterface" and "pUnk"
 // automatically call Release
 //
 // 4. Ensure that the following directories are in your include
```

```
path:
 // a) The directory containing the Solid Edge typelibs
 // b) The directories containing the Visual Studio includes
 //
 // 5. Try to keep the "#import" statements in the standard pch
header
 // so that all the .cpp files in the project automatically have
access
 // to the smart pointers generated from the typelibs.
 //
 // 6. Smart pointers handle error returns by converting error
 // HRESULTs into "_com_error" exceptions. The "_com_error" class
 / encapsulates the HRESULT error code. Since these objects throw
 // exceptions, you will need a try-catch within your code, as
shown
 // in function "RunSEAutomation()". However, if you use the "raw"
 // versions of the interface functions that are returned, you can
 // avoid exceptions, and deal with regular HRESULTs instead. For
 // more information, read the MSDN articles mentioned above.
 //
 // 7. The compiler support implementation converts properties into
 // Get/Put pairs. But the property is also usable directly, as in
 // VB. For example,the "Visible" property on the
 // Application object is usable in the following ways:
 //
 // ApplicationPtr pApp;
 // [get the app pointer]
 // pApp->Visible = VARIANT_TRUE; // this is VB like syntax
 // pApp->PutVisible(VARIANT_TRUE); // this is the equivalent C++
like syntax
 //
 // However, methods are called as usual, such as "pApp-
>Activate()".
 //
 // 8. When Solid Edge creates typelibs, it tries to make each one
 // of them self-contained with respect to the constants that are
 // used by the objects within that typelib. This will allow users
to
 // browse for constants used in a particular typelib within that
same
 // typelib, without having to bring up another typelib in a
typelib
 // browser. But a side effect of this when we are using compiler
 // support #import of typelibs is that we have to explicitly
qualify
 // the constant as coming from a particular typelib (because more
than
 // one has the same constant). In most such cases, we have to
scope
 // the constant to the type library where the
method/property/object
 // resides, because that is how the compiler will expect it to be
 // declared. If that does not work, scope it to
SolidEdgeConstants. The
 // latter contains ALL the constants.
 //
 // 9. Unfortunately, parameters of type SAFEARRAY don't have
compiler support
 // classes, unlike VARIANT, whose corresponding compiler support
class
 // is "_variant_t", or BSTR, whose corresponding class is
"_bstr_t". So
 // SafeArrays have to be managed using the various SafeArray APIs
that Visual
 // C++ provides to manage the creation/manipulation/deletion of
SafeArrays.
 //
 //
 // WARNING:
 // -------
 //
 // 1. You will find interfaces of type "_<SomeInterface>" defined
in the
 // typelib. These are vtable interfaces that support the
corresponding
 // dispatch versions. Although these do show up in the typelib and
 // "#import" generates smart pointers for these, clients MUST NOT
use
 // these in their code, for two reasons:
 // a) These interfaces are intentionally not marshalled so that
any
 // out-of-proc client (exe) cannot use these interfaces.
 // b) These are private to Solid Edge and subject to change from
version
 // to version, so client code can behave unpredictably when these
are
 // used, including causing crashes.
 //
 // The vtable interfaces that COULD be used however, don't have
 // an "underbar" prefix, and can be used.
 //
 // For example:
 //
 // - Don't use "_IApplicationAutoPtr", but use "ApplicationPtr"
 // - Don't use "_IPartDocumentAutoPtr", but use "PartDocumentPtr"
 // - Don't use "_IDMDBodyPtr", but use "BodyPtr"
 // - Can use "ISEDocumentEventsPtr"
 // - Can use "ISEMousePtr"
 //
 // 2. The function "main()" only does initialization and
uninitialization
 // of the COM runtime. The function it calls (i.e.
"RunSEAutomation()")
 // does all the work. The reason is that we do NOT want to use any
smart
 // pointers within "main()". What happens is this: smart pointers
are
 // created on the stack and therefore destructed at the end of the
 // function scope. If there are any smart pointers declared in
"main()",
 // then they will be destroyed AFTER the COM runtime has been
uninitialized.
 // This is a no-no, because un-initializing the COM runtime has to
be
 // the VERY last thing we do. If we use any COM objects/interfaces
 // after un-initializing COM, it can cause runtime crashes. The
other
 // alternative is to use a local scope in "main()" so that all
 // smartpointers are declared within it and are therefore
destroyed
 // before CoUninitialize is called.
 //
 // 3. Be careful when you mix smart pointers and non-smart
pointers (i.e.
 // straight COM interface pointers). In this case, you have to be
aware
 // of the AddRefs and Releases going on in the background and
 // may have to manually insert code to do some AddRefs and
Releases to
 // be COM-compliant.
 //
 void RunSEAutomation();
 HRESULT CreateAssemblyUsingPartFile(ApplicationPtr pSEApp);
 HRESULT CreateDrawingUsingAssemblyFile(ApplicationPtr pSEApp);


 // The entry-point function

 void main()
 {
   bool initialized = false;
   HRESULT hr = NOERROR;

   // Very first thing to do is to initialize the Component Object
   // Model (COM) runtime.
   hr = CoInitialize(NULL);
   HandleError(hr, "Failed to initialize COM runtime");

   // Very important. CoInitialize and CoUninitialize have to be
called in pairs.
   initialized = true;

   // Now get down to business
   RunSEAutomation();


 wrapup:

   // Make sure to un-initialize on the way out
   if (initialized)
   {
   // If we have initialized the COM runtime, we now have to
uninitialize it.
   CoUninitialize();
   }

   return;
 }

 // This function shows you how to connect to Solid Edge. After
connecting
 // to Solid Edge, it runs some useful client automation code, by
calling
 // "CreateAssemblyUsingPartFile()" and
"CreateDrawingUsingAssemblyFile()".

 void RunSEAutomation()
 {
   HRESULT hr = NOERROR;

   ApplicationPtr pSEApp; // Smart pointer for
SolidEdgeFramework::Application


   // Since the compiler support classes throw C++ exceptions when
servers
   // return an error HRESULT, we have to have this try-catch block
here.

   try
   {
   // Try to get a running instance of Solid Edge from the
   // running object table.
   if (FAILED(pSEApp.GetActiveObject("SolidEdge.Application")))
   {
   // Dont have Solid Edge running. Create a new instance.
   hr = pSEApp.CreateInstance("SolidEdge.Application");
   HandleError(hr, "Failed to create an instance of Solid Edge");
   }

   // Now that we are sure Solid Edge is running, create new
assembly
   // and drawing files. Note that any exceptions thrown from
within
   // "CreateAssemblyUsingPartFile" and
"CreateDrawingUsingAssemblyFile"
   // will be caught in this function.

   // First make the application visible.
   pSEApp->Visible = VARIANT_TRUE;

   hr = CreateAssemblyUsingPartFile(pSEApp);
   HandleError(hr, "Failed in CreateAssemblyUsingPartFile");

   hr = CreateDrawingUsingAssemblyFile(pSEApp);
   HandleError(hr, "Failed in CreateDrawingUsingAssemblyFile");

   }
   catch (_com_error &comerr)
   {
   cerr << "_com_error: " << comerr.Error() /* HRESULT */ << " : "
   << comerr.ErrorMessage() /* Error string */ << endl;

   }
   catch (...)
   {
   cerr << "Unexpected exception" << endl;
   }


 wrapup:

   return;
 }



 HRESULT CreateAssemblyUsingPartFile(ApplicationPtr pSEApp)
 {

   HRESULT hr = NOERROR;

   AssemblyDocumentPtr pAsmDoc;

   // First create a new Assembly document (default parameters
   // that are not specified are handled just as in VB)
   pAsmDoc = pSEApp->GetDocuments()-
>Add(L"SolidEdge.AssemblyDocument");

   // Add a new occurrence using a Part file. Creates a grounded
occurrence.
   if (pAsmDoc)
   {
   pAsmDoc->GetOccurrences()->AddByFilename("c:\\block.par");


   // Finally, save the assembly file.
   pAsmDoc->SaveAs("c:\\block.asm");
   }
   else
   {
   hr = E_FAIL;
   }
```

```
    return hr;

  }

  HRESULT CreateDrawingUsingAssemblyFile(ApplicationPtr pSEApp)
  {

    HRESULT hr = NOERROR;

    DraftDocumentPtr pDftDoc;
    ModelLinkPtr pLink;

    // First create a new Draft document (default parameters that
are
    // not specified are handled just as in VB)
    pDftDoc = pSEApp->GetDocuments()-
>Add(L"SolidEdge.DraftDocument");

    if (pDftDoc)
    {
    // Link the newly created assembly file to this draft document
    pLink = pDftDoc->GetModelLinks()->Add("c:\\block.asm");

    if (pLink)
    {
    // Now create a drawing view using the model link above
    pDftDoc->GetActiveSheet()->GetDrawingViews()->Add(pLink,
    SolidEdgeDraft::igTopBackRightView, 1.0, 0.1, 0.1);
    }
    else
    {
    hr = E_FAIL;
    }

    // Finally, save the drawing file.
    pDftDoc->SaveAs("c:\\block.dft");

    }
    else
    {
    hr = E_FAIL;
    }
    return hr;

  }
```

C H A P T E R 16

Add-Ins

This chapter contains information on using add-ins with Solid Edge.

Working with Add-ins—Overview

The Solid Edge API provides an easy-to-use set of interfaces that enable programmers to fully integrate custom commands with Solid Edge. These custom programs are commonly referred to as *add-ins*. Specifically, Solid Edge defines an add-in as a dynamically linked library (DLL) containing a COM-based object that implements the ISolidEdgeAddIn interface. More generally, an add-in is a COM object that is used to provide commands or other value to Solid Edge.

The only restriction placed on an add-in is that the add-in must use standard Windows-based resources. An example of such a resource would be device-independent bitmaps to be added to the Solid Edge graphical user interface. You can create these resources using any of the popular visual programming tools that are available in the Windows programming environment—Visual C++ or Visual Basic, for example.

The following interfaces are available to the add-in programmer:

- ISolidEdgeAddIn—The first interface implemented by an add-in. Provides the initial means of communicating with Solid Edge.
- ISEAddInEvents and DISEAddInEvents—Provides command-level communication between the add-in and Solid Edge.

In addition, several Solid Edge interfaces are available once the add-in is connected to Solid Edge. These include ISEAddIn, ISECommand/DISECommand, ISECommandEvents/DISECommandEvents, ISEMouse/DISEMouse, ISEMouseEvents/DISEMouseEvents, ISEWindowEvents/DISEWindowEvents, and ISolidEdgeBar.

Implementing an Add-in

A Solid Edge add-in has the following requirements:

- The add-in must be a self-registering ActiveX DLL. You must deliver a registry script that registers the DLL and adds Solid Edge-specific information to the system registry.
- The add-in must expose a COM-creatable class from the DLL in the registry.
- The add-in must register the CATID_SolidEdgeAddin as an Implemented Category in its registry setting so that Solid Edge can identify it as an add-in.
- The add-in must implement the ISolidEdgeAddIn interface. The definition of this interface is delivered with the Solid Edge SDK (addins.h). The add-in can implement any additional interfaces, but ISolidEdgeAddIn is the interface that Solid Edge looks for.
- During the OnConnect call (made by Solid Edge on the add-in's ISolidEdgeAddIn interface), the add-in can add commands to one or more Solid Edge environments.
- If a graphical user interface (buttons or toolbars, for example) is associated with the add-in, then the add-in must provide a GUI version to be stored by Solid Edge. If the GUI version changes the next time the add-in is loaded, then Solid Edge will purge the old GUI and re-create it based on the calls to AddCommandBarButton with the OnConnectToEnvironment method. A GUI is an optional component of an add-in; some add-ins, for example, simply monitor Solid Edge events and perform actions based on those activities.
- You must follow COM rules and call AddRef on any Solid Edge pointers that the add-in is holding on to. You must also release the pointers when they are no longer needed. In Visual Basic, AddRef is done automatically by Set SEInterface = <Solid Edge interface>; to release the interface, set the interface to "Nothing."
- For Visual C++ users, a Solid Edge Add-in Wizard exists. The wizard is currently available for download from the Solid Edge web site. The wizard generates fully functional add-ins based on Microsoft's Active Template Library (ATL) for COM.

Working with the ISolidEdgeAddIn Interface

The ISolidEdgeAddIn interface is the first interface that is implemented by an add-in and provides the initial means of communication with Solid Edge. It allows for connection to and disconnection from an add-in. The implementation of this interface is what identifies a COM object as being a Solid Edge add-in.

OnConnection

```
HRESULT OnConnection( IDispatch *pApplication, seConnectMode
ConnectMode, AddIn *pAddIn )
```

Solid Edge passes in a pointer to the dispatch interface of the Solid Edge application that is attempting to connect to the add-in. The add-in uses this pointer to make any necessary calls to the application to connect to Solid Edge event sinks, or to otherwise communicate with Solid Edge to perform whatever tasks the add-in needs when first starting up.

Solid Edge passes in a connect mode that indicates what caused Solid Edge to connect to the add-in. Current modes are as follows:

- seConnectAtStartUp—Loading the add-in at startup.
- seConnectByUser—Loading the add-in at user's request.
- seConnectExternally—Loading the add-in due to an external (programmatic) request.

Solid Edge also passes in a dispatch interface of a Solid Edge Add-in object that provides another channel of communication between the add-in and Solid Edge. An equivalent v-table form of this interface can be obtained by querying the input Add-in's dispatch interface for the ISEAddIn interface (also described in addins.h).

In general, the add-in needs to do very little needs when OnConnection is called. Here are a few basic steps that an add-in may want to perform during connection.

1. Connect to any Solid Edge application event sets the add-in plans on using by providing the appropriate sinks to the application object.
2. Connect to the Solid Edge Add-in object's event set if the add-in plans to add any commands to any environments.
3. Set the GUI version property of the Solid Edge Add-in object.

OnDisconnection

```
HRESULT OnDisconnection( SeDisconnectMode DisconnectMode )
```

Solid Edge passes in a disconnect mode that indicates what caused Solid Edge to disconnect to the add-in. Current modes are as follows:

- SeDisconnectAtShutDown—Unloading at shutdown.
- SeDisconnectByUser—Unloading the add-in due to a user request.
- SeDisconnectExternally—Unloading the add-in due to an external (programmatic) request.

To disconnect, the add-in should do the following:

4. Disconnect from any Solid Edge event sets it may have connected to.
5. Disconnect from the Add-in event set (if connected).
6. Release any other objects or interfaces the add-in may have obtained from the application.
7. Close any storage and/or streams it may have opened in the application's document.
8. Perform any other cleanup such as freeing any resources it may have allocated.

OnConnectToEnvironment

```
HRESULT OnConnectToEnvironment( BSTR EnvCatID, LPDISPATCH
pEnvironment, VARIANT_BOOL* bFirstTime )
```

Solid Edge passes in the category identifier of the environment as a string. If the add-in is registered as supporting multiple environments, the add-in can use the string to determine which environment to which it is being asked to connect.

Solid Edge passes in the dispatch interface of the environment.

Solid Edge passes in the bFirstTime parameter to specify that a Solid Edge environment is connecting to the add-in for the first time. When connecting for the first time, the add-in, if necessary, should add any needed user interface elements (for example, buttons). On exiting, Solid Edge will save any such buttons so they can be restored during the next session.

To connect to a Solid Edge environment, the add-in will perform the following steps in its OnConnectToEnvironment:

1. The add-in should always call the SetAddInInfo method of the add-in interface passed to it during OnConnection if it provides any command bars or command bar buttons in the environment.
2. The add-in uses the bFirstTime parameter to determine if it is the first time the add-in has been loaded into the environment by checking to see if it is VARIANT_TRUE. If it is, the add-in should add any command bar buttons it

needs to carry out its commands by calling the add-in interface's AddCommandBarButton method. If the add-in is not disconnected, and its GUI version has not changed the next time Solid Edge loads the add-in, then Solid Edge will set the parameter to VARIANT_FALSE because Solid Edge will save the data provided it by the add-in the last time the parameter was VARIANT_TRUE. Note that if the add-in's OnDisconnect function is called with a disconnect mode different from seDisconnectAtShutdown, this parameter will be VARIANT_TRUE the next time Solid Edge calls OnConnection. This happens because when an add-in is disconnected by the user or programatically, Solid Edge will purge all GUI modifications made by the add-in from all environments.

3. Add any commands not included in any of the calls to SetAddInInfo by calling the application's AddCommand method. Generally this method is used when a command is being added to the menu but not any command bar.

Note Command bars are persisted by Solid Edge when exiting. When an environment is first loaded, connection to the add-in is performed before the environment's command bars are loaded. This allows an add-in to call SetAddInInfo to supply any glyphs needed by any buttons that were previously saved by Solid Edge.

Add-ins cannot assume the existence of any particular environment, until this function is called with that environment's catid. Any calls with a catid for an environment that does not yet exist will be rejected.

Working with ISEAddInEvents and DISEAddInEvents

When an add-in adds commands to a Solid Edge environment, a system of notifications must exist between the add-in and Solid Edge to enable and disable commands, invoke commands, and provide help for the commands. The ISEAddinEvents interface and its equivalent dispatch interface, DISEAddinEvents, serve this purpose.

One of these two interfaces is implemented by the add-in object and is used by Solid Edge to invoke commands added to Solid Edge by the add-in and to allow the add-in to perform basic user interface updates. The interface contains three methods: OnCommand, OnCommandUpdateUI and OnCommandHelp.

OnCommand

```
HRESULT OnCommand( long nCmdID )
```

Solid Edge calls this method, passing in nCmdID whenever the user invokes an add-in command. The value of the add-in command identifier passed in is the same value the add-in previously gave the AddIn object when it called its SetAddInInfo method.

When OnCommand is called, if the add-in wants to take advantage of the Solid Edge command control or mouse control, it can create a command object using the application's CreateCommand method. CreateCommand returns a DISECommand interface (from which the ISECommand equivalent v-table interface can be obtained).

OnCommandUpdateUI

```
HRESULT OnCommandUpdateUI( long nCmdID, long* pdwCmdFlags, BSTR
Menutext, long *nIDBitmap)
```

Solid Edge calls this method, passing in nCmdID whenever it needs to determine the availability of a command previously added to Solid Edge by the add-in. The value of nCmdID will be one of the values the add-in previously passed in the SetAddInInfo method. The add-in uses the pointer to the dwCmdFlags bit mask to enable/disable the command and to notify Solid Edge to make other GUI changes. The values of the masks are as follows:

- seCmdActive_Enabled—Used to enable the command.
- seCmdActive_Checked—Used to add a check mark on the command's menu item.
- seCmdActive_ChangeText—Used to change the text that appears on the command's menu item.

- seCmdActive_UseDotMark—Used to add a dot mark on the command's menu item.
- seCmdActive_UseBitmap—Used to display the command's menu item as a bitmap.

Menutext can be used to change the text that appears on the menu. In order to change the text, allocate and return the desired text string. nIDBitmap can be used to have a bitmap appear on the menu next to the text.

Note After calling OnCommandUpdateUI, Solid Edge will determine whether seCmdActive_UseBitmap is set and if so, the returned value of nIDBitmap should represent the resource identifier of a bitmap stored in the resource DLL whose handle was passed in the SetAddInInfo method.

This method is called to determine if a command is enabled or disabled. It is called for commands visible on toolbars during idle processing, just before displaying a menu, when an accelerator is pressed, and when the application receives a WM_COMMAND message.

OnCommandHelp

```
HRESULT OnCommandHelp(long hFrameWnd, long uHelpCommand, long
nCmdID )
```

Solid Edge calls this method, passing in nCmdID whenever the user requests help for an add-in command previously added to Solid Edge by the add-in. The value of the add-in command identifier passed in will be one of the values the add-in gave the application when it previously called the SetAddInInfo method. If Solid Edge passes in -1, the add-in should call help for the add-in in general (that is, not help for a specific command).

The handle to the frame window, hFrameWnd as well as an indicator as to the type of help (uHelpCommand) is also passed in. These two parameters can be used in the WinHelp call and valid values of uHelpCommand are documented with the WinHelp function documentation.

Note When a command bar button is added, the dispatch interface of the button is returned. The interface contains help filename and context properties that can be set by the add-in. If set, these properties are used to invoke WinHelp directly from Solid Edge instead of calling OnCommandHelp.

Working with Solid Edge Objects, Interfaces, and Events

ISEAddIn

This interface is passed into the add-in's OnConnection method. The Solid Edge objects that implement this interface are created one per add-in when Solid Edge starts up regardless of whether the add-in is loaded. These objects represent the add-in within Solid Edge. This same interface is exposed by means of the application's add-in's collection object. Because this interface is exposed to automation, some of the functions in the interface can only be called during the connection process, thus ensuring that only the add-in itself makes the calls.

Syntax Examples

- To return the dispatch interface of the application:

```
HRESULT get_Application( IDispatch **Application )
```

- To return the IUnknown of the connectable object that provides the ISEAddInEvents and DISEAddInEvents connection points:

```
HRESULT get_AddInEvents( AddInEvents **AddInEvents )
```

- To determine whether or not the add-in is connected. Connect is set to VARIANT_TRUE if the add-in is connected otherwise VARIANT_FALSE:

```
HRESULT get_Connect( VARIANT_BOOL *Connect )
```

- To programmatically connect to (VARIANT_TRUE) or disconnect from (VARIANT_FALSE) the add-in:

```
HRESULT put_Connect( VARIANT_BOOL Connect )
```

- To access a brief description of the add-in:

```
HRESULT get_Description( BSTR *Description )
```

- To set a brief description of the add-in. The description should be internationalized and also serves as the menu text of a tools pop-up menu that will be created if the add-in adds any commands to an environment. The put_Description can only be called successfully during initial connection:

```
HRESULT put_Description( BSTR Description )
```

- To get the add-in's guid in the string format defined by the Win API StringFromGUID. The CLSIDFromString Win API can convert the string back into its globally unique identifier form:

```
HRESULT get_GUID( BSTR *GUID )
```

- To get the version of the add-in as it relates to the user interface changes it makes in Solid Edge:

```
HRESULT get_GuiVersion( long *GuiVersion )
```

- To set the version of the add-in as it relates to the user interface changes it makes in Solid Edge. An add-in that adds commands to any environment should always set the version in OnConnect. Solid Edge will persist this version when it shuts down. On subsequent runs, a difference in the last persisted version and the version passed to put_GuiVersion will cause Solid Edge to purge any and all menu entries and command bar buttons that were created and saved in the last session. Also, when OnConnectToEnvironment is called, bFirstTime will be set to VARIANT_TRUE when a change in the version is detected:

```
HRESULT put_GuiVersion( long GuiVersion )
```

- To get the dispatch interface of the add-in if it exists. Be sure to call Release() when the interface is no longer needed:

```
HRESULT get_Object( IDispatch **Object )
```

- To set the dispatch interface of the add-in. Solid Edge will AddRef the object when storing it and call Release when it successfully disconnects from the add-in:

```
HRESULT put_Object( IDispatch *Object )
```

- To access the program identifier of the add-in if it has one. Solid Edge will call ProgIDFromCLSID with the clsid of the add-in and return it as the string:

```
HRESULT get_ProgID( BSTR *ProgID )
```

- To determine whether the add-in should appear in the list of add-ins presented to the user for connection and disconnection by Solid Edge. Setting Visible to VARIANT_FALSE will also prevent the add-in from being disconnected programatically:

```
HRESULT get_Visible( VARIANT_BOOL *Visible )
```

- To prevent Solid Edge from presenting the add-in in the list of add-ins presented to the user for connection and disconnection and to prevent the add-in from being disconnected programatically, called with a value of VARIANT_FALSE. Note that add-ins that set Visible to VARIANT_FALSE ensure that the add-in will only be disconnected at shutdown:

```
HRESULT put_Visible( VARIANT_BOOL Visible )
```

To call SetAddInInfo:

```
HRESULT SetAddInInfo( long nInstanceHandle, BSTR EnvCatID, BSTR
CategoryName,
   long nIDColorBitMapResourceMedium,
   long nIDColorBitMapResourceLarge,
   long nIDMonochromeBitMapResourceMedium,
   long nIDMonochromeBitMapResourceLarge,
   long nNumberOfCommands, SAFEARRAY **CommandNames,
   SAFEARRAY **CommandIDs )
```

- **nInstanceHandle** is the HINSTANCE of the add-in's resource DLL, cast to a long.
- **EnvCatID** is the category identifier of the environment to which commands are being added.
- **CategoryName** is a name that the add-in associates with the set of commands it is adding to the environment. The name should be internationalized as it can be presented to the user by Solid Edge.
- **nIDColorBitMapResourceMedium** is the ID of the bitmap resource containing medium-sized images of all the toolbar buttons that the add-in is adding.
- **nIDColorBitMapResourceLarge** is the ID of the bitmap resource containing large-sized images of all the toolbar buttons that the add-in is adding.
- **nIDMonochromeBitMapResourceMedium** is the ID of the bitmap resource containing medium-sized monochrome images of all the toolbar buttons that the add-in is adding.
- **nIDMonochromeBitMapResourceLarge** is the ID of the bitmap resource containing large-sized monochrome images of all the toolbar buttons that the add-in is adding.
- **nNumberOfCommands** is the number of commands being added to the environment.
- **CommandNames** is an array of BSTRs. Each string can contain sub-strings separated by "\n". The substrings are defined as follows:
- Name of the command you are adding. This should not be internationalized and should be tagged in such a way to help ensure uniqueness.
- Text displayed on the menu entry for the command. This substring may contain backslash characters, which Solid Edge (Version 7 or later) will use to create additional pop-up submenus and/or to add a separator preceding the command entry (Version 8 or later). The strings appearing between the backslashes

become the title of the pop-up menu and the last string becomes the entry on the final menu. If the first character of any substring (including the first) is itself a backslash, Solid Edge will add a separator preceding the menu entry.

- Status bar string. This is the string displayed on the status bar.
- Tooltip string. This is the string displayed as the tooltip.
- Macro string. If present, this string becomes the macro associated with the command. Commands that have a macro string will not be invoked by calling OnCommand. Instead, Solid Edge runs the macro.
- Parameter string. If present, this string is passed as an argument to the macro.

Example:

```
"MyAddinCommand1\nSEaddin Sample Command\nDisplays a message
box\nSeaddin Command"
```

The non-internationalized tag for the command is "MyAddinCommand1". "Seaddin Sample Command" will appear as an entry on the addin's pop-up menu created by Solid Edge. "Displays a message box" will appear in the status field of the frame window. "Seaddin Command" is displayed as the tooltip for the command if it is added to a command bar by calling AddCommandBarButton.

Example:

```
"MyAddinCommand1\nSEaddin\ Sample Command\nDisplays a message
box\nSeaddin Command"
```

This example is identical to the one above with one exception. That being that an additional pop-up submenu named "Seaddin" will exist with "Sample Command" being an entry on that pop-up

Example:

```
"MyAddinCommand1\nSEaddin\\ Another Sample Command\nDisplays a
message box\nSeaddin Command"
```

This example is identical to the one above with one exception. Due to the additional backslash, a separator will be inserted preceding the menu entry "Another Sample Command".

- **CommandIDs** on input is a pointer to a SAFEARRAY of identifiers the add-in is to associate with each command being added. The add-in is free to choose any identifier it wishes. The command identifier chosen by the add-in is what is passed in OnCommand, OnCommandUpdateUI and OnCommandHelp.

CommandIDs is also an output of SetAddInInfo. When the function returns, the array contains the runtime command identifier Solid Edge has associated with the command. This identifier is what the operating system will pass in the WM_COMMAND message. It can also be used to add a button for the command using the "Add" method available in the command bar controls' automation interface.

To call AddCommandBarButton:

```
HRESULT AddCommandBarButton( BSTR EnvCatID, BSTR CommandBarName,
   long nCommandID, CommandBarButton
   **CommandBarButton )
```

- EnvCatID is the category identifier of the environment to which a button is being added.
- CommandBarName is the name of the command bar the button will be added to. Solid Edge will create the command bar if it does not exist.
- nCommandID is any of the command identifiers the add-in passed to SetAddInInfo (not the identifier passed back from Solid Edge).
- CommandBarButton is the dispatch interface of the button object that provides for additional programming capabilities. The v-table equivalent interface, ISECommandBarButton can be obtained by querying the returned object for IID_ISECommandBarButton. For example, if the add-in wants to have Solid Edge invoke WinHelp for the command, it can set the help filename and help context properties.

AddCommandBarButton is used by the add-in to have Solid Edge display a button for the command. This routine only needs to be called if the OnConnectToEnvironment argument, bFirstTime is VARIANT_TRUE. Note that this method can be called anytime (that is, Solid Edge does not restrict calls to this routine to emanate from the add-in's OnConnectToEnvironment method). Buttons can also be added via the command bar automation interfaces but there are advantages to using this method.

- Solid Edge will create the command bar if it does not exist.
- Solid Edge can maintain the relationship between the button and the add-in. This allows Solid Edge to remove such buttons when the add-in is disconnected or if in subsequent startups, the add-in no longer exists because it has been uninstalled by the user. It also allows Solid Edge to purge old commands if the GUI version of the add-in has changed.
- One function call as opposed to the many calls needed to add a button via the automation interfaces.

Note Always be sure to Release the returned CommandBarButton interface.

An add-in can set its CommandBarButton OnAction properties (and ParameterText) to a valid Solid Edge "macro" and not connect up to the AddIn event set to listen for OnCommand. When the user selects the command, Solid Edge uses its "Run Macro" subsystem to run the command.

Example:

```
OnAction = "notepad.exe "
ParameterText = "test.txt"
```

Pressing a button with these properties and added by an add-in that is not connected to the AddIn event set will cause Solid Edge to launch Notepad with "test.txt" as the file for Notepad to open.

To call AddCommand:

Use this method instead of SetAddInInfo for those commands without a GUI that goes with it (that is, there are no command bar buttons).

```
HRESULT AddCommand( BSTR EnvCatID, BSTR CmdName, long lCommandID)
```

- **EnvCatID** is the category identifier of the environment to which a command is being added.
- **CmdName** is a string that contains sub-strings separated by a new line character ('\n'). In order, the substrings are:
- Name of the command you are adding. This should not be internationalized and should be tagged in such a way to help ensure uniqueness.
- Text displayed on the menu entry for the command. This substring may contain backslash characters, which Solid Edge (Version 7 or later) will use to create additional pop-up submenus and/or to add a separator preceding the command entry (Version 8 or later). The strings appearing between the backslashes become the title of the pop-up menu and the last string becomes the entry on the final menu. If the first character of any substring (including the first) is itself a backslash, Solid Edge will add a separator preceding the menu entry.
- Status bar string. This is the string displayed on the status bar.
- Tooltip string. This is the string displayed as the tooltip.
- Macro string. If present, this string becomes the macro associated with the command. Commands that have a macro string will not be invoked by calling OnCommand. Instead, Solid Edge runs the macro.
- Parameter string. If present, this string is passed as an argument to the macro.

Example:

```
"MyAddinCommand1\nSEaddin Sample Command\nDisplays a message
box\nSeaddin Command"
```

The non-internationalized tag for the command is "MyAddinCommand1". "Seaddin Sample Command" will appear as an entry on the addin's pop-up menu created by Solid Edge. "Displays a message box" will appear in the status field of the frame window. "Seaddin Command" is displayed as the tooltip for the command if it is added to a command bar by calling AddCommandBarButton.

Example:

```
"MyAddinCommand1\nSEaddin\ Sample Command\nDisplays a message
box\nSeaddin Command"
```

This example is identical to the one above with one exception. That being that an additional pop-up submenu named "Seaddin" will exist with "Sample Command" being an entry on that pop-up

Example:

```
"MyAddinCommand1\nSEaddin\\ Another Sample Command\nDisplays a
message box\nSeaddin Command"
```

This example is identical to the one above with one exception. Due to the additional backslash, a separator will be inserted preceding the menu entry "Another Sample Command".

- lCommandID is the index used by Solid Edge to identify to the add-in which command is being invoked when Solid Edge calls the OnCommand, OnCommandUpdateUI and OnCommandHelp events.

ISECommand/DISECommand

When the application's CreateCommand method is called, it returns the DISECommand dispatch interface of an object (that can be queried for ISECommand using IID_ISECommand to get its equivalent v-table interface). The properties and methods are as follows:

- Mouse—Read-only property that returns the DISEMouse interface object (which can be queried for ISEMouse using IID_ISEMouse to get its equivalent v-table interface) of the mouse control. The object that implements the interface is also a connectable object that provides the DISEMouseEvents and ISEMouseEvents event sets.
- CommandWindow—Read-only property that returns the IUnknown interface of a connectable object that provides the DISECommandWindowEvents and ISECommandWindowEvents event sets.
- Done—Read-write Boolean property used to notify Solid Edge that the command is finished and is to be terminated. Commands that are created with the seTerminateAfterActivation setting do not have to set this property as they will be terminated after Activate is called. Other commands should set this property when their command is finished. The initial (default) value is FALSE.
- OnEditOwnerChange—Read-write Long property used to notify Solid Edge that the command is to be terminated whenever an edit owner change occurs. Commands that set this variable will be relieved of undo transaction calls. Commands that modify multiple edit owners (for example, multiple documents) should set this flag to zero. In such cases, it is up to the command to make any necessary undo transaction calls using the automation interfaces.
- OnEnvironmentChange—Read-write Long property used to notify Solid Edge that the command is to be terminated whenever an environment change occurs.

- Start—Call this method after creating the command and connecting to the command event set. After Start is called and control is returned to Solid Edge, the command, its mouse and window will start sending events to the add-in for processing.

Note To receive the Activate, Deactivate and Terminate events, the add-in must connect up to the command event set before calling Start.

ISECommandEvents/DISECommandEvents

The object returned from CreateCommand provides these event sets. The add-in command will normally provide this event sink and connect the sink to the object acquired by calling CreateCommand. The object passed in implements IID_IConnectionPointContainer so the command can connect the sink using the standard COM connection point interfaces (don't forget to Advise/Unadvise). The member functions of this sink are as follows:

- Activate()—Activates the command.
- Deactivate()—Deactivates the command. Either the user has terminated the command or has invoked another command (that is, the command is being stacked). At this point, if the command has any modal dialog boxes displayed, it should undisplay them until Activate is called.
- Terminate()—Notifies the command that it is being terminated.
- Idle(long lCount, LPBOOL pbMore)—Notifies the command that idle cycles are available. lCount represents the number of cycles that have occurred. The command can set pbMore to FALSE in which case Solid Edge will not give the command any more idle cycles. If a command does not perform any idle processing, it should set this to FALSE.
- KeyDown(unsigned short * KeyCode, short Shift)—Notifies the command of a key down event. The keycode and a shift indicator is passed into this method.
- KeyPress(unsigned short * KeyAscii)—Notifies the command of an ASCII key press event. The ASCII character is passed into this method.
- KeyUp(unsigned short * KeyCode, short Shift)—Notifies the command of a key up event. The keycode and a shift indicator is passed into this method.

Note that the difference between KeyDown and KeyPress is subtle. KeyDown occurs whenever any key is pressed while KeyPress occurs only when an ASCII key is pressed. Also be aware that both of these events "repeat" as long as the user continues to keep a key pressed.

Examples:

- The user presses and releases the SHIFT key. Solid Edge sends the command a KeyDown event followed by a KeyUp event.

- The user presses and releases the ENTER key. Solid Edge sends the command a KeyDown event, a KeyPress, and then a KeyUp event, in that order.
- The user presses the F1 key and continues to hold the key down. Solid Edge sends the command a series of KeyDown events until the user releases the key, at which time Solid Edge sends a KeyUp event.
- The user presses the number 5 key and continues to hold the key down. Solid Edge sends the command a series of KeyDown and KeyPress events until the user releases the key, at which time Solid Edge sends a KeyUp event.

ISEMouse/DISEMouse

This is the interface returned from the command's Mouse property. This interface is used by the add-in command to get and set certain properties used by Solid Edge to help the add-in command manage mouse events. This includes the capability to specify various Solid Edge locate modes and to set locate filters that enable the add-in command to specify what types of objects should be located.

The properties of this interface are as follows:

- ScaleMode—Read-write Long value. Setting ScaleMode to 0 implies that coordinates of the mouse events are in the underlying window coordinate system and 1 implies that they are in design modeling coordinate system.
- EnabledMove—Read-write Boolean that, if set to True, causes Move and Drag in progress events to be fired.
- LastEventWindow—Read-only. Returns the dispatch interface of the window object in which the last mouse event occurred.
- LastUpEventWindow—Read-only. Returns the dispatch interface of the window object in which the last mouse up event occurred.
- LastDownEventWindow—Read-only. Returns the dispatch interface of the window object in which the last mouse down event occurred.
- LastMoveEventWindow—Read-only. Returns the dispatch interface of the window object in which the last mouse move event occurred.
- LastEventShift—Read-only Short that is the state of the CTRL, ALT, and SHIFT keys when the last mouse event occurred. Valid values are those enumerated by the seKey constants.
- LastUpEventShift—Read-only Short that is the state of the CTRL, ALT, and SHIFT keys when the last mouse up event occurred. Valid values are those enumerated by the seKey constants.
- LastDownEventShift—Read-only Short that is the state of the CTRL, ALT, and SHIFT keys when the last mouse down event occurred. Valid values are those enumerated by the seKey constants.

- LastMoveEventShift—Read-only Short that is the state of the CTRL, ALT, and SHIFT keys when the last mouse move event occurred. Valid values are those enumerated by the seKey constants.
- LastEventButton—Read-only Short that indicates which button the last mouse event occurred on. Valid values are those enumerated by the seButton constants.
- LastUpEventButton—Read-only Short that indicates which button the last mouse up event occurred on. Valid values are those enumerated by the seButton constants.
- LastDownEventButton—Read-only Short that indicates which button the last mouse down event occurred on. Valid values are those enumerated by the seButton constants.
- LastMoveEventButton—Read-only Short that indicates which button the last mouse move event occurred on. Valid values are those enumerated by the seButton constants.
- LastEventX—Read-only Double that is the X coordinate of the mouse when the last mouse event occurred.
- LastEventY—Read-only Double that is the Y coordinate of the mouse when the last mouse event occurred.
- LastEventZ—Read-only Double that is the Z coordinate of the mouse when the last mouse event occurred.
- LastUpEventX—Read-only Double that is the X coordinate of the mouse when the last mouse up event occurred.
- LastUpEventY—Read-only Double that is the Y coordinate of the mouse when the last mouse up event occurred.
- LastUpEventZ—Read-only Double that is the Z coordinate of the mouse when the last mouse up event occurred.
- LastDownEventX—Read-only Double that is the X coordinate of the mouse when the last mouse down event occurred.
- LastDownEventY—Read-only Double that is the Y coordinate of the mouse when the last mouse down event occurred.
- LastDownEventZ—Read-only Double that is the Z coordinate of the mouse when the last mouse down event occurred.
- LastMoveEventX—Read-only Double that is the X coordinate of the mouse when the last mouse move event occurred.
- LastMoveEventY—Read-only Double that is the Y coordinate of the mouse when the last mouse move event occurred.
- LastMoveEventZ—Read-only Double that is the Z coordinate of the mouse when the last mouse move event occurred.

- WindowTypes—Read-write Long which, if set to 0, implies that mouse events emanate from all windows. If set to 1, WindowTypes implies that mouse events emanate only from graphic windows.
- LastEventType—Read-only Long that returns the last mouse event type. Valid values are those enumerated by the seMouseAction constants.
- EnabledDrag—Read-write VARIANT_BOOL that if set to VARIANT_TRUE causes drag events to be fired.
- LocateMode—Read-write Long that indicates how to locate: 0 implies SmartMouse locate, 1 implies simple click locate (no multi-select dialog), 2 implies quick pick locate (multi-select dialog where applicable) and 3 implies no locate (used to receive mouse events without performing any locate). For simple and quick pick locate modes, users will be able to mouse down, drag and mouse up for fence locate. This property is applicable when a mouse service object is registered.
- DynamicsMode—Read-write Long that specifies which shape to draw in dynamics: 0 implies off, 1 implies line, 2 implies circle, 3 implies rectangle.
- PauseLocate—Read-write Long that specifies how long in milliseconds to wait before a locate occurs. Use this property when you don't want to locate during mouse moves but do want to locate as the mouse pauses or hesitates.

The methods of this interface are as follows:

- ClearLocateFilter()—Clears the locate filter. If the locate mode is not seLocateOff, clearing the filter enables all filters.
- AddToLocateFilter—Restricts locates to the graphic types specified. Valid values are those enumerated by the seLocateFilterConstants constants.

ISEMouseEvents/DISEMouseEvents

The command's Mouse property also supports these event sets. Add-in commands that are interested in mouse events, including locate capability, will normally provide one of these event sinks and connect the sink to the Mouse. The Mouse implements IConnectionPointContainer so the command can connect the sink using the standard COM connection point interfaces (be sure to Advise/Unadvise). When a command is in a state where it does not want to process mouse events or perform locates, it can Unadvise this sink until such time it requires those events, in which case it can call Advise again. Note that there is no limit to the number of connections attached to this connection point. However, be aware that there is only one set of properties that control the behavior of this sink. The member functions of this sink are as follows:

- MouseDown—This event is sent whenever the user presses a mouse button down. Button and Shift are identical to those used in the mouse property events.

```
 MouseDown( short sButton, short sShift, double dX, double
dY, double dZ,
   LPDISPATCH pWindowDispatch, long lKeyPointType,
    LPDISPATCH pGraphicDispatch )
```

- MouseUp—This is the same as the MouseDown event except that it is sent whenever a mouse up event occurs.
- MouseMove—This is the same as the MouseDown event except that it is sent whenever a mouse move event occurs.
- MouseClick—This is the same as the MouseDown event except that it is sent whenever a mouse click event occurs.
- MouseDblClick—This is the same as the MouseDown event except that it is sent whenever a mouse double click event occurs.
- MouseDrag—This event is sent whenever the user presses a mouse button down. Button and Shift are identical to those used in the mouse property events. See the seMouseDragStateConstants constants for values for the drag state.

```
MouseDrag( short sButton, short sShift, double dX, double
dY, double dZ,
   LPDISPATCH pWindowDispatch, , short DragState, long
lKeyPointType,
   LPDISPATCH pGraphicDispatch ) -
```

When a mouse click occurs in a Solid Edge window, the Down, Up, and then Click events are fired to the add-in command. Correspondingly, when a mouse double click occurs the Down, Up, Click, Double Click, and then Up events are fired. If enabled, drag events are fired when the mouse moves with a mouse button down. Let's take the case where the user clicks a button, moves the mouse with the button down, and then releases the button. In this case the following four possibilities exist:

	Enabled Drag	**Enabled Move**	**Events Fired on the Mouse control**
Case 1	False	False	Down, Up, Click
Case 2	False	True	Down, Move, ... Move, Up, Click
Case 3	True	False	Down, Drag (State = Enter), Drag (State = Leave)
Case 4	True	True	Down, Drag (State = Enter), Drag (State = In progress), Drag (State = In progress), Drag (State = Leave)

Effort is made to ensure that the mouse events are fired in sequence. To do this when a mouse down event occurs, the mouse input is locked to the window in which the down event occurred. When the corresponding mouse up event occurs, the lock is removed. This sequence ensures that the add-in command will always receive a mouse up event after receiving a mouse down event. Because the mouse up event will occur in the same window as the down event, this implies that it is not possible to drag across windows.

ISEWindowEvents/DISEWindowEvents

The command's CommandWindow property supports these event sets. Add-in commands that are interested in generic window events, including any registered private window messages (see the Window's API, RegisterWindowMessage), will want to provide this event sink and connect the sink to the CommandWindow property object. The object passed in implements IConnectionPointContainer so the command can connect the sink using the standard COM connection point interfaces (be sure to Advise/Unadvise). When a command is in a state where it does not want to process window events, it can Unadvise this sink until such time it requires those events, in which case it can call Advise again. Note that there is no limit to the number of connections attached to this connection point. The member functions of this sink are as follows:

```
WindowProc( IDispatch* pDoc, IDispatch pView, UINT nMsg,
   WPARAM wParam, LPARAM lParam, LRESULT *lResult )
```

Note that this function is analogous to the standard WindowProc function used by Window applications everywhere. The main difference is the dispatch pointers and the LRESULT passed into it. The reason the LRESULT exists is that this function is a member of a COM interface and hence must return an HRESULT. Since WindowProc functions normally return a value whose value and meaning is determined by the nMsg argument, this argument has been added and serves the same purpose.

An example of why an add-in may want to implement this sink is to take advantage of the WM_SETCURSOR message. For more information on that message, and what the value of LRESULT means to the caller of this event function, see the Window's documentation for WM_SETCURSOR.

ISolidEdgeBar

This interface can be used by an add-in to insert a page into the Solid Edge Edgebar tool. The Edgebar is available starting with Version 8. The pages that exist on the Edgebar tool are always document-specific. That means that a page added to the Edgebar for one document, will not appear on the Edgebar for any other document. In order to obtain the Edgebar interface, query the AddIn interface passed into the ISolidEdgeAddIn::OnConnection method using IID_ISolidEdgeBar. Once obtained, the interface can be used to add a page to the Edgebar, remove a previously added page from the Edgebar, and to set the active page of the Edgebar to one that has been added.

AddPage

AddPage is called to add a page to the Edgebar tool for the document passed in. The HWND passed back can be used in Windows APIs. For Visual C++ users this handle can also be used, for example, to create a CWnd object that can be used as the parent of a CDialog object that may be positioned inside the client area of the returned page.

```
HRESULT AddPage( IDispatch *theDocument, long nInstanceHandle, long
nBitmapID,
   BSTR strTooltip, long nOption, long *hWndPage)
```

- theDocument is the dispatch interface of the document for which the page is being added.
- nInstanceHandle is HINSTANCE of the add-in's resource DLL, cast to a long, in which the bitmap resides.
- nBitmapID is the resource identifier of the bitmap that will appear on the added page's tab. The bitmap dimensions should be 20 by 20.
- sStrTootip is a string which appears as a tooltip for the user when the cursor is passed over the page's bitmap.
- nOption indicates which options the page wants. The options available are enumerated by EdgeBarConstant located in the Solid Edge constants typelib. A value of zero is valid and indicates no option. When this document was written, the only available option indicates that resize events are not needed. Hence, zero indicates that resize events are needed.
- hWndPage is the HWND of the added page and is returned by Solid Edge to the caller. The handle can be used to, for example, to draw items on the page.

RemovePage

```
RemovePage( IDispatch *theDocument, long hWndPage, long nOptions )
```

- theDocument is the dispatch interface of the document for which the page is being removed.
- hWndPage is the HWND of the page being removed.
- nOption indicates which options are needed. This argument is not currently supported; set nOption to zero.

SetActivePage

```
SetActivePage(IDispatch *theDocument, long hWndPage, long nOptions
)
```

- theDocument is the dispatch interface of the document for which the page is being activated.
- hWndPage is the HWND of the page being activated.
- nOption indicates which options are needed. This argument is not currently supported; set nOption to zero.

Additional Solid Edge Objects, Interfaces and Events

Additional Solid Edge objects, interfaces and event sets are obtainable by means of the automation interface pointers (the IDispatch pointers passed into any of the add-in's event sinks or interfaces). These interfaces are not directly related to the add-in system. They are generic automation related interfaces and thus are documented by the Solid Edge SDK.

Registering an Add-in

For Solid Edge to know there is an add-in registered for use with it, the add-in needs to add the "Implemented Categories" subkey in the registry and add the GUID for CATID_SolidEdgeAddIn.

Solid Edge also defines a set of categories that are used to indicate which environment(s) an add-in is designed for. These categories are not directly supported by the Category Manager (the COM-supplied ICatInformation interface). Solid Edge, however, will search for an additional key, the "Environment Categories" key, which is much like the "Implemented/Required Categories" keys supported by the Category Manager. Add-ins will enumerate as sub-keys to that key, any Solid Edge environment for which the add-in plans to add commands and/or events. An add-in must implement at least one of these categories.

The following categories identify what environments an add-in is designed for:

4. CATID_SEApplication
5. CATID_SEAssembly
6. CATID_SEPart
7. CATID_SEProfile
8. CATID_SESheetMetal

In addition to registering the COM object that represents the add-in, which includes not only the normal registry entries that the Microsoft Component Object Model specifies, but also the component categories the add-in implements, there are a few other registry entries that Solid Edge requires the add-ins to register. These entries will be the values or subkeys of the classid key of the add-in classid registered in the HKEY_CLASSES_ROOT\CLSID registry entry. Currently, the entries and their meanings are as follows:

- Automatic connection indicator—This value name should be "AutoConnect" and the type should be a DWORD. Set the value to 1 for now.
- LocaleID—The value name should be a Microsoft defined locale id and the type should be a string. The string should contain a locale specific description of the add-in. Example: "409" (which identifies the locale as U.S. English) and "My company's Solid Edge Add-in". The id should be stored in hexadecimal format. For more information on locale ids (also known as LCIDs), see the Microsoft documentation concerning locales.

In addition to the previously described registry entries, Solid Edge Versions 8 and greater look for additional registry entries. These are used by the Add-In Manager (as is the original LocaleID value string already registered).

- Summary—The key name should be "Summary," and it should contain the following:

    - LocaleID—The value name should be a Microsoft defined locale id (for example, 409), and the type should be a string. The string should contain a locale-specific summary of the add-in. The summary string will be presented to the user upon invocation of the Solid Edge Add-In Manager. This entry is analogous to the add-in's description mentioned above. The id should be stored in hexadecimal format.

- Help—The key name should be "Help" and it should contain the following:

    - LocaleID—This named value is a string. The string is the name of the localized help filename. The help file can be invoked by the user via the Solid Edge Add-In Manager's GUI.

Note Failure to register the proper locale id will result in blank or invalid entries on the top-level menu for an add-in's commands. A blank summary of the add-in on the Add-In Manager's dialog box or the inability of the user to invoke the add-in's help file from the Add-In Manger's dialog box.

Although some of the locale-specific registry entries can also be set during connection, it is important for an add-in to add these entries to the registry. The reason for doing so is to enable Solid Edge to present the user with information about an add-in without loading the add-in.

Sample Registry File

```
REGEDIT4
 ;Copyright (C) 1999 Unigraphics Solutions. All rights reserved.
 ;Changes for REGISTRY format Change
 ; 10/27/99 JsBielat
 REGEDIT4

 ; Sample script for VB-based AddIns.
 ; VB automatically puts out most of the basic reg entries. Just
add the Solid Edge specific entries.


 [HKEY_CLASSES_ROOT\CLSID\{C7CF857B-7FE0-11D2-BE8E-080036B4D502}]
 @="My Solid Edge Add-in CLSID"
 "AutoConnect"=dword:00000001
 "409"="This is my localized (US English) addin registry string"

 [HKEY_CLASSES_ROOT\CLSID\{C7CF857B-7FE0-11D2-BE8E-
080036B4D502}\Environment Categories]

 [HKEY_CLASSES_ROOT\CLSID\{C7CF857B-7FE0-11D2-BE8E-
080036B4D502}\Environment Categories\{26618396-09D6-11d1-BA07-
080036230602}]
 @="Solid Edge Part Environment"

 [HKEY_CLASSES_ROOT\CLSID\{C7CF857B-7FE0-11D2-BE8E-
080036B4D502}\Implemented Categories]

 [HKEY_CLASSES_ROOT\CLSID\{C7CF857B-7FE0-11D2-BE8E-
080036B4D502}\Implemented Categories\{26B1D2D1-2B03-11d2-B589-
080036E8B802}]
 @="My Solid Edge AddIn"
 ; Register the SolidEdge Addins CATID in case it is not already
registered


 [HKEY_CLASSES_ROOT\Component Categories\{26B1D2D1-2B03-11d2-B589-
080036E8B802}]
 @="Solid Edge AddIn CATID"
 [HKEY_CLASSES_ROOT\Component Categories\{26B1D2D1-2B03-11d2-B589-
080036E8B802}\409]
 @="This is the CATID for SolidEdge AddIn"
```

C H A P T E R 17

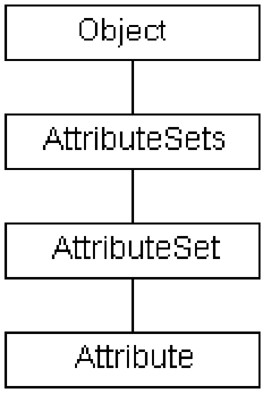
Working with Dynamic Attributes (Storing Private Data)

This chapter contains information on how to attach user-defined attributes to an existing object. This is very useful for users who want to include their own data with Solid Edge objects in a persistent manner (that is, data that is saved in the Solid Edge document).

Working with Dynamic Attributes—Overview

Dynamic attributes allow you to define new properties for most objects in Solid Edge. This feature is available only through automation.

Dynamic attributes are contained in attribute sets. An attribute set is a group of attributes associated with a host object. An object can contain many uniquely named attribute sets, so that attributes added by one user don't interfere with another user's attributes on the *same* object. Most objects in Solid Edge can be host objects; an object is a host if it supports the AttributeSets property.



Note While not an enforced rule, to avoid confusion, attribute names should be unique. A common practice is to use a company or organization prefix.

Defining Attribute Sets

You create Attribute Sets at runtime using the Add method of the AttributeSets collection. The syntax for creating a new attribute set and attaching it to an object is as follows:

```
<Object>.AttributeSets.Add ("<AttrSetName>")
```

The Add method returns the newly created attribute set object, so the following syntax is also valid:

```
Dim objAttrSet as Object
 objAttrSet = <Object>.AttributeSets.Add ("<AttrSetName>")
```

An Attribute Set is a collection; you use the Add method to add attributes to the collection as follows:

```
objAttrSet.Add "<AttributeName>", <type>
```

The Add method returns the newly created attribute. This can be used to set the value of the attribute, as described in *Manipulating Attribute Sets.*

Constants for attribute types are defined in the SolidEdgeConstants type library as AttributeTypeConstants. The following is a list of the available types and the constant values that correspond to them:

Constant Value	**Attribute Type**
seBoolean	Boolean
seByte	Byte
seCurrency	Currency
seDate	Date
seDouble	Double-precision floating point number
seInteger	Integer (2 byte)
seLong	Long integer (4 byte)
seSingle	Single-precision floating point number
seStringANSI	ANSI string
seStringUnicode	Unicode string

To remove a property from an attribute set, use the following syntax:

```
objAttrSet.Remove "<AttributeName>"
```

Manipulating Attribute Sets

To access a user-defined attribute set, use either of the following statements. Because Item is the default property of the AttributeSets collection, these statements are equivalent:

```
<Object>.AttributeSets.Item( "<AttributeSetName>" )
```

```
<Object>.AttributeSets( "<AttributeSetName>" )
```

You can access the value of a property in a user-defined attribute set in any of the following ways:

```
<AttributeSet>.Item("<AttributeName>")
```

```
<AttributeSet>("<AttributeName>")
```

You can combine the various ways to access Attribute and Attribute Sets as needed. For example, the following statements are equivalent:

```
<Object>.AttributeSets("<AttributeSetName>")("<AttributeName>")

<Object>.AttributeSets("<AttributeSetName>").Item("<AttributeName>"
)
```

The following syntax (with the equal sign on the right side of the property name) sets the value of a property in a user-defined attribute set:

```
<Object>.AttributeSets("<AttributeSetName>").Item("<AttributeName>
") = "<user-defined string>"
```

To access an attribute in a user-defined attribute set, the equal sign is placed on the left side of the property name. Each attribute is an object that supports three properties: Name, Type, and Value.

You can modify only the Value property, which gives the current value of the attribute. Value is the default property of the Attribute object. In the following example, the property value is stored in the strData variable:

```
 Dim objAttribute As Object
 Dim strData As String
 objAttribute =
<Object>.AttributeSets("<AttrSetName>").Item("<AttributeName>")
 If objAttribute.Type = seStringUnicode Then
 strData = objAttribute.Value
 End If
```

The following syntax allows you to determine if a named attribute set is present on an object:

```
<Object>.IsAttributeSetPresent "<AttributeSetName>"
```

Sample Program—Creating Attribute Sets

The following program defines a new attribute set and attribute for a cylinder. First, the program draws a line in Solid Edge and defines a new attribute for the line. Using the Add method of the AttributeSets collection, a new attribute set named MachineInfo is created and attached to a feature in the selection set. A new string attribute, FeatureType, is added to the attribute set using the Add method of the Attribute Set object. The new attribute is assigned the character string, "a string value."

```
'Declare the program variables.
Dim objApp As Object
Dim objFeature As Object
Dim objSelectSet As Object
Dim objAttrSet As Object

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Access an object from the selection set.
Set objSelectSet = objApp.ActiveDocument.SelectSet
If objSelectSet.Count <> 1 Then
  MsgBox "You must select a single feature."
  End
End If

Set objFeature = objSelectSet(1)

'Create a new attribute set and attach it to the object.
Set objAttrSet = objFeature.AttributeSets.Add("MachineInfo")

'Define a new string attribute for the line.
objAttrSet.Add "FeatureType", seStringUnicode

'Define the string value of the new attribute.
objAttrSet.FeatureType.Value = "a string value"
```

Note: The technique of accessing the attribute value through the attribute set is specific to Visual Basic. When programming with Visual C++, it is better to iterate through the AttributeSet, access the attribute, and then change the specific value.

Sample Program—Enumerating Attribute Sets

You can use the AttributeSets collection to determine which attribute sets are connected to an object. Similarly, you can use the attribute set collection to extract the attributes that it contains.

The following sample code extracts the names of the attribute sets connected to objects in a selection set:

```
'Declare the program variables.
Dim objApp As Object
Dim objAttributeSets As Object
Dim objAttributeSet As Object
Dim objAttribute As Object
Dim strSetName As String
Dim strAttributeName As String
Dim objGeometry As Object
Dim objSelectSet As Object
Dim strMsgText As String

'Connect to a running instance of Solid Edge.
Set objApp = GetObject(, "SolidEdge.Application")

'Access the SelectSet object.
Set objSelectSet = objApp.ActiveDocument.SelectSet

'If objects exist in a selection set
If objSelectSet.Count <> 0 Then
  'For each object
  For Each objGeometry In objSelectSet
    'Access the AttributeSets collection object.
    Set objAttributeSets = objGeometry.AttributeSets

    'For each AttributeSet object in the AttributeSets collection
  For Each objAttributeSet In objAttributeSets
    'display attribute set Name to a message box.
    If objAttributeSet.SetName = "MachineInfo" Then
    strMsgText = "Attribute Set" & Chr(13)
    strMsgText = strMsgText & "Name: " & _
    objAttributeSet.SetName & _
    Chr(13)
    For Each objAttribute In objAttributeSet
    strMsgText = strMsgText & _
    objAttribute.Name & _
    ": " & objAttribute.Value
    Next
    MsgBox strMsgText
    End If
   Next objAttributeSet
  Next objGeometry
Else
  MsgBox "There are no objects selected.", vbExclamation
End If
```

Note Most, but not *all* objects support attribute sets.

APPENDIX A

Learning Visual Basic

This chapter contains a suggested self-paced learning plan for Microsoft Visual Basic.

Learning Visual Basic—Overview

This section outlines a plan for learning Microsoft Visual Basic. Several books and resources are available on Visual Basic. The suggested exercises guide you through the Visual Basic concepts you need to understand before you can work efficiently with the Solid Edge automation interface. The following topics are covered:

- Creating and using Controls
- Menus and dialog boxes
- Managing projects
- Programming fundamentals
- Variables, constants, and data types
- Objects and instances
- ActiveX Automation
- Debugging
- Handling runtime errors

Exercises

Day One

Objective: To create and save a Visual Basic project. To design a dialog box using the objects available in the default tool box.

1. Read and work through the exercises in the first four chapters of Teach Yourself Visual Basic in 21 Days. These chapters give an introduction to creating projects in Visual Basic, as well as information on programming fundamentals. The exercises also teach how to create dialog boxes with Visual Basic.

Day Two

Objective: To design dialog boxes using objects in the default toolbox. To add custom controls to the toolbox and menus to dialog boxes.

2. Read and work through the exercises in Chapters 5 and 6 of Teach Yourself Visual Basic in 21 Days. These chapters describe how to add menus and custom dialog boxes.
3. Read Chapters 1 through 4 of the Visual Basic Programmer’s Guide, preferably at the computer, to expand your understanding of the fundamentals of working with forms, modules, controls, and projects.

Day Three

Objective: To understand the data types used in Visual Basic. To implement the techniques described in these chapters to add code to the objects introduced in the exercises from Day One.

4. Read Chapters 5 through 7 of the Visual Basic Programmer’s Guide to learn the most frequently used programming techniques. This includes an explanation of control structures, arrays, and the various data types available in Visual Basic.
5. Work through the examples in Chapter 15 of Teach Yourself Visual Basic in 21 Days. This chapter explains the data types used in Visual Basic.

Day Four

Objective: To understand the basics of how ActiveX Automation is used with Visual Basic. To know how to use Visual Basic’s debugging functions to find and correct errors in your code.

6. Read the section on ActiveX (OLE) in Chapter 15 of Teach Yourself Visual Basic in 21 Days for a brief introduction to implementing ActiveX Automation in Visual Basic. For more detailed information about ActiveX Automation, read Chapters 7 through 9 of the Visual Basic Programmer’s Guide.

7. Read Chapters 20 and 21 of the Visual Basic Programmer’s Guide to learn how to use the debugging capabilities in Visual Basic.

APPENDIX B

Sample Programs

This appendix describes the sample programs delivered with Solid Edge and explains how to run them.

How to Run the Samples

You can run the Solid Edge sample programs by accessing them from the Solid Edge\custom directory. A readme.txt file, which explains how to run the associated sample, is included in each directory

Adding Geometric Constraints Automatically (AutoCnst)

This sample allows you to select a set of geometry in Profile, Layout, or Draft and automatically recognize and add geometric constraints.

This sample shows how to make a DLL that will run "in-process," resulting in faster startup and performance. The autocnst.vbp project must be compiled as a DLL file and then run from the Solid Edge Profile, Layout, or Draft environment.

Demonstrates:

- Recognizing geometric constraints automatically.
- Adding geometric constraints automatically.
- Creating DLLs.

Controlling Batch Processing (batch)

This sample uses a dialog box that allows the user to choose three types of batch processing:

1. Printing a directory of Draft files.
2. Converting a set of files into Solid Edge (for example, converting a set of DXF files into Solid Edge draft files).
3. Converting Solid Edge files into another format.

Run this program (batch.exe) from Explorer. Batch.exe starts Solid Edge if it is not already running.

Demonstrates:

- How to control batch processing.

Copying Dimension Styles (CopyDimStyle)

This sample copies the style from one dimension to another. The program is a DLL that you run from within Solid Edge using the Run Macro command.

Demonstrates:

- How to read the current properties of a dimension.
- How to set the properties of a dimension.

Creating a Bill of Materials (bom)

This sample takes a currently open assembly and produces one of several Bill of Material reports. The sample navigates through the assembly structure and reports on all of the subassemblies and parts that exist in the assembly. The resulting list is displayed on a dialog box.

You can run this program (bom.exe) from Explorer. The program expects Solid Edge to be running with an open Assembly file. The program honors the IncludeInBOM property on the part and ignores parts and assemblies that have this property set to false.

Demonstrates:

- Working with assemblies.

Creating Features Using the Stock Wizard (stock)

This sample displays a dialog box that shows standard stock shapes. From the dialog box, users select a shape and enter values to define dimensions. When the user has defined the shape and dimensions, the solid base feature is created.

You run this program (stock.exe) from Explorer. The program expects Solid Edge to be running with an open Part file.

Demonstrates:

- Working with reference planes.
- Working with profiles.
- Working with features.

Customizing the Sheet Metal Application (SheetMetal)

This sample demonstrates how you can use Visual Basic to customize the Solid Edge SheetMetal application to compute the flat pattern length of bend features using different standards.

Demonstrates:

- Working with Din Standard.

- Working with a table.
- Working with an ANSI table.
- Working with an ISO table.

Extracting the Edges of a Flattened Model (GandT)

This sample demonstrates how you can use Visual Basic to flatten a sheet metal part and then use Geometry and Topology (G&T) portions of the automation interface to extract the edges of the flattened model. The resulting 2-D edges of the flattened model are written out to the c:\temp\SEDump.txt file.

You run this program (gandt.exe) from Explorer. The program expects Solid Edge to be running with an open Sheet Metal file.

Demonstrates:

- Working with sheet metal parts.
- Working with geometry and topology.

Modifying Graphics from Excel Data (bearing)

This sample demonstrates how to link cells in a Microsoft Excel spreadsheet to variables in a Solid Edge part drawing. The variables are linked to the spreadsheet cells using the variable table in Solid Edge. Once the cells are linked to the variables in the part drawing, the part is automatically updated by Solid Edge whenever a cell in the spreadsheet changes.

Demonstrates:

- How to attach Solid Edge variables to an Excel spreadsheet.

Opening and Saving Part Files (OpenSave)

This sample opens, saves, and closes each of the part files in the specified directory. The sample uses the Visual Basic Drive, Directory, and File List Box controls

Using this sample, you can update documents created before version 5 to the new Parasolids-based file formats.

Demonstrates:

- Opening, saving, and closing files.

Placing a Hexagonal Profile (HexPro)

This sample demonstrates running a macro in the Profile environment of Solid Edge. It uses the Command and Mouse controls to get a coordinate from the user. It then uses this coordinate as the center point to place a hexagonal-shaped profile. This sample performs only the sketching step of placing a feature. Once the profile has been placed, users exit the program and then interactively edit the profile to complete the feature.

Demonstrates:

- How to automate portions of a workflow.
- Working with units of measure.
- Working with 2-D graphic objects.

Reading and Writing Dynamic Attributes (DynAttrib)

This sample demonstrates how to use the automation interface to read and write Solid Edge dynamic attributes. To run this sample, Solid Edge must be running and a document of any type that contains graphics must be open.

Demonstrates:

- Working with dynamic attributes.

Setting the Document CreationLocale Property (ChngLcl)

This sample changes the CreationLocale property of Solid Edge documents. This property prevents documents from having incompatible locale-specific data written to them. Solid Edge will not let users save data from one locale in a file that was created in another locale. This preserves users' data from accidental corruption, but can pose an obstacle to sharing Solid Edge documents. This utility provides a workaround.

Note Users can safely change the CreationLocale property originally created on English systems to any other language because all Windows operating systems recognize the English character data. This is not the case with some of the other languages, so you should not use this utility to change CreationLocale, for example, to allow Japanese and Polish data to be written to the same document. Most importantly, you should never use this utility to change the CreationLocale property from a double-byte language to a single-byte language.

Run this utility (ChngLcl.exe) from Explorer.

Demonstrates:

- How to open documents.
- How to change property values.
- How to close documents.
- How to display messages.

Using Mouse and Command Controls (mouse)

This sample demonstrates how to use the Solid Edge Mouse and Command controls to allow a Visual Basic application to receive and process mouse events from Solid Edge. The sample displays a dialog box that allows you to manipulate all of the Mouse control properties and view the results.

On the Tools menu, click Macro from within any of the Solid Edge environments to run the sample.

Demonstrates:

- Using Mouse and Command controls.

Working with Revision Manager (Astruct)

This sample demonstrates how to use the Revision Manager Automation Interface by navigating an assembly and displaying the assembly components in a dialog box.

Run this program (Astruct.exe) from Explorer. Solid Edge must be running with an Assembly document open.

Demonstrates:

- Working with Revision Manager.

A P P E N D I X C

Solid Edge Commands and Controls

This appendix contains information on working with Solid Edge commands and controls.

Visual C++ Program Syntax

```
 // system includes
 #include <objbase.h>
 #include <comdef.h>
 #include <iostream.h>


 // Import all the Solid Edge type libraries. This will create a .tli and
.tlh file
 // for each of these type libraries, that will wrap the code to call into
the
 // methods and properties of the objects in the type libraries.

 #import "constant.tlb"
 #import "framewrk.tlb"
 #import "fwksupp.tlb"
 #import "geometry.tlb"
 #import "part.tlb"
 #import "assembly.tlb"
 #import "draft.tlb"

 // Use the typelib namespaces.

 using namespace SolidEdgeConstants;
 using namespace SolidEdgeFramework;
 using namespace SolidEdgeFrameworkSupport;
 using namespace SolidEdgeGeometry;
 using namespace SolidEdgePart;
 using namespace SolidEdgeAssembly;
 using namespace SolidEdgeDraft;


 // Error handling macro. Every function that calls this macro needs to
 // have a label called "wrapup" declared, to jump to in case of error.
 // This is where any cleanup should be done or resources freed before
 // exiting the function.
 #define HandleError(hr, message) \
 \
 if FAILED(hr) \
 { \
   cerr << "Error: 0x" << hex << hr << dec << " Line: " << __LINE__ << "
Message: " << message << endl; \
   goto wrapup; \
 }\
 \




 // NOTES:
 // -----
 //
 // 1. "RunSEAutomation()" is a driver function that attaches to
 // an instance of Solid Edge and gets an interface pointer to the
 // Solid Edge application. The real client code that does anything
 // useful is encapsulated within this function. The "main()" function
 // simply initializes COM, calls "RunSEAutomation()" and un-initializes
 // COM. See the "Warning" section below for more information on this.
 //
 // 2. We have chosen to use the easiet means of calling methods on the
 // Solid Edge automation objects. While there are MFC COleDispatchDriver
 // classes that can wrap IDispatch interfaces, it is significantly
 // easier to use the "#import" technique to import entire typelibs into
 // client code, which automatically creates all the wrapper classes and
 // their corresponding implementation of "IDispatch->Invoke()"
 // using smart pointers.
 //
 // 3. Some familiarity with COM smart pointers is assumed. For more
 // information/help:
 // a) See topics "Compiler COM Support Classes" & "Compiler COM
 // Support: Overview" in MSDN.
 // b) See Visual Studio include file "comdef.h.".
 //
 // When you #import a typelib, VC++ automatically creates .tli and
 // .tlh files. The former define smart pointers for each and every
interface
 // defined in the typelib (both vtable and dispinterface), using
 // the "_COM_SMARTPTR_TYPEDEF" macro. If there is an interface of type
 // "IFoo" in the typelib, the smart pointer associated with that is
 // named "IFooPtr" (by adding "Ptr" to the interface name). The
 // smart-pointer implementation basically encapsulates the real COM
 // interface pointers and eliminates the need to call the AddRef, Release,
 // QueryInterface methods that all interfaces support. In addition, they
 // hide the CoCreateInstance call for creating a new COM object. Since
 // these smart pointers are also know the UUID for the interface they
 // are wrapping, simply equating two smart pointers will call
 // QueryInterface on the interface on the right hand side for the
 // UUID of the interface supported by the smart pointer on the left
 // hand side (much like VB).
 //
 //
 // For example the following code (error checking omitted) :
 //
 // {
 // IUnknown *pUnk;
 // IAnyInterface *pAnyInterface;
 // [some code to get pAnyInterface]
 // pAnyInterface->QueryInterface(IID_IUnknown, (LPVOID**) &pUnk);
 // pUnk->Release();
 // pAnyInterface->Release();
 // }
 //
 // can be replaced by
 //
 // {
 // IUnknownPtr pUnk;
 // IAnyInterfacePtr pAnyInterface;
 // [some code to get pAnyInterface]
 // pUnk = pAnyInterface; // does the QI internally on pUnk
 // } // destructors on smart pointers "pAnyInterface" and "pUnk"
 // automatically call Release
 //
 // 4. Ensure that the following directories are in your include path:
 // a) The directory containing the Solid Edge typelibs
 // b) The directories containing the Visual Studio includes
 //
 // 5. Try to keep the "#import" statements in the standard pch header
 // so that all the .cpp files in the project automatically have access
 // to the smart pointers generated from the typelibs.
 //
 // 6. Smart pointers handle error returns by converting error
 // HRESULTs into "_com_error" exceptions. The "_com_error" class
 / encapsulates the HRESULT error code. Since these objects throw
 // exceptions, you will need a try-catch within your code, as shown
 // in function "RunSEAutomation()". However, if you use the "raw"
 // versions of the interface functions that are returned, you can
 // avoid exceptions, and deal with regular HRESULTs instead. For
 // more information, read the MSDN articles mentioned above.
 //
 // 7. The compiler support implementation converts properties into
 // Get/Put pairs. But the property is also usable directly, as in
 // VB. For example,the "Visible" property on the
 // Application object is usable in the following ways:
 //
 // ApplicationPtr pApp;
 // [get the app pointer]
 // pApp->Visible = VARIANT_TRUE; // this is VB like syntax
 // pApp->PutVisible(VARIANT_TRUE); // this is the equivalent C++ like
syntax
 //
 // However, methods are called as usual, such as "pApp->Activate()".
 //
 // 8. When Solid Edge creates typelibs, it tries to make each one
 // of them self-contained with respect to the constants that are
 // used by the objects within that typelib. This will allow users to
 // browse for constants used in a particular typelib within that same
 // typelib, without having to bring up another typelib in a typelib
 // browser. But a side effect of this when we are using compiler
 // support #import of typelibs is that we have to explicitly qualify
 // the constant as coming from a particular typelib (because more than
 // one has the same constant). In most such cases, we have to scope
 // the constant to the type library where the method/property/object
 // resides, because that is how the compiler will expect it to be
 // declared. If that does not work, scope it to SolidEdgeConstants. The
 // latter contains ALL the constants.
 //
 // 9. Unfortunately, parameters of type SAFEARRAY don't have compiler
support
 // classes, unlike VARIANT, whose corresponding compiler support class
 // is "_variant_t", or BSTR, whose corresponding class is "_bstr_t". So
 // SafeArrays have to be managed using the various SafeArray APIs that
Visual
 // C++ provides to manage the creation/manipulation/deletion of
SafeArrays.
 //
 //
 // WARNING:
 // -------
 //
 // 1. You will find interfaces of type "_<SomeInterface>" defined in the
 // typelib. These are vtable interfaces that support the corresponding
 // dispatch versions. Although these do show up in the typelib and
 // "#import" generates smart pointers for these, clients MUST NOT use
 // these in their code, for two reasons:
 // a) These interfaces are intentionally not marshalled so that any
 // out-of-proc client (exe) cannot use these interfaces.
 // b) These are private to Solid Edge and subject to change from version
 // to version, so client code can behave unpredictably when these are
 // used, including causing crashes.
 //
 // The vtable interfaces that COULD be used however, don't have
 // an "underbar" prefix, and can be used.
 //
 // For example:
 //
 // - Don't use "_IApplicationAutoPtr", but use "ApplicationPtr"
 // - Don't use "_IPartDocumentAutoPtr", but use "PartDocumentPtr"
 // - Don't use "_IDMDBodyPtr", but use "BodyPtr"
 // - Can use "ISEDocumentEventsPtr"
 // - Can use "ISEMousePtr"
 //
 // 2. The function "main()" only does initialization and uninitialization
 // of the COM runtime. The function it calls (i.e. "RunSEAutomation()")
 // does all the work. The reason is that we do NOT want to use any smart
 // pointers within "main()". What happens is this: smart pointers are
 // created on the stack and therefore destructed at the end of the
 // function scope. If there are any smart pointers declared in "main()",
 // then they will be destroyed AFTER the COM runtime has been
uninitialized.
 // This is a no-no, because un-initializing the COM runtime has to be
 // the VERY last thing we do. If we use any COM objects/interfaces
 // after un-initializing COM, it can cause runtime crashes. The other
 // alternative is to use a local scope in "main()" so that all
 // smartpointers are declared within it and are therefore destroyed
 // before CoUninitialize is called.
 //
 // 3. Be careful when you mix smart pointers and non-smart pointers (i.e.
 // straight COM interface pointers). In this case, you have to be aware
 // of the AddRefs and Releases going on in the background and
 // may have to manually insert code to do some AddRefs and Releases to
 // be COM-compliant.
 //



 void RunSEAutomation();
 HRESULT CreateAssemblyUsingPartFile(ApplicationPtr pSEApp);
 HRESULT CreateDrawingUsingAssemblyFile(ApplicationPtr pSEApp);


 // The entry-point function

 void main()
 {
   bool initialized = false;
   HRESULT hr = NOERROR;

   // Very first thing to do is to initialize the Component Object
   // Model (COM) runtime.
   hr = CoInitialize(NULL);
   HandleError(hr, "Failed to initialize COM runtime");

   // Very important. CoInitialize and CoUninitialize have to be called in
pairs.
   initialized = true;

   // Now get down to business
   RunSEAutomation();


 wrapup:
   // Make sure to un-initialize on the way out
   if (initialized)
   {
   // If we have initialized the COM runtime, we now have to uninitialize
it.
   CoUninitialize();
   }

   return;
 }

 // This function shows you how to connect to Solid Edge. After connecting
 // to Solid Edge, it runs some useful client automation code, by calling
 // "CreateAssemblyUsingPartFile()" and "CreateDrawingUsingAssemblyFile()".

 void RunSEAutomation()
 {
   HRESULT hr = NOERROR;

   ApplicationPtr pSEApp; // Smart pointer for
SolidEdgeFramework::Application


   // Since the compiler support classes throw C++ exceptions when servers
   // return an error HRESULT, we have to have this try-catch block here.

   try
   {
   // Try to get a running instance of Solid Edge from the
   // running object table.
   if (FAILED(pSEApp.GetActiveObject("SolidEdge.Application")))
   {
   // Dont have Solid Edge running. Create a new instance.
   hr = pSEApp.CreateInstance("SolidEdge.Application");
   HandleError(hr, "Failed to create an instance of Solid Edge");
   }

   // Now that we are sure Solid Edge is running, create new assembly
   // and drawing files. Note that any exceptions thrown from within
   // "CreateAssemblyUsingPartFile" and "CreateDrawingUsingAssemblyFile"
   // will be caught in this function.

   // First make the application visible.
   pSEApp->Visible = VARIANT_TRUE;

   hr = CreateAssemblyUsingPartFile(pSEApp);
   HandleError(hr, "Failed in CreateAssemblyUsingPartFile");

   hr = CreateDrawingUsingAssemblyFile(pSEApp);
   HandleError(hr, "Failed in CreateDrawingUsingAssemblyFile");

   }
   catch (_com_error &comerr)
   {
   cerr << "_com_error: " << comerr.Error() /* HRESULT */ << " : "
   << comerr.ErrorMessage() /* Error string */ << endl;

   }
   catch (...)
   {
   cerr << "Unexpected exception" << endl;
   }
 wrapup:

   return;
 }



 HRESULT CreateAssemblyUsingPartFile(ApplicationPtr pSEApp)
 {

   HRESULT hr = NOERROR;

   AssemblyDocumentPtr pAsmDoc;

   // First create a new Assembly document (default parameters
   // that are not specified are handled just as in VB)
   pAsmDoc = pSEApp->GetDocuments()->Add(L"SolidEdge.AssemblyDocument");

   // Add a new occurrence using a Part file. Creates a grounded
occurrence.
   if (pAsmDoc)
   {
   pAsmDoc->GetOccurrences()->AddByFilename("c:\\block.par");


   // Finally, save the assembly file.
   pAsmDoc->SaveAs("c:\\block.asm");
   }
   else
   {
   hr = E_FAIL;
   }

   return hr;

 }

 HRESULT CreateDrawingUsingAssemblyFile(ApplicationPtr pSEApp)
 {

   HRESULT hr = NOERROR;

   DraftDocumentPtr pDftDoc;
   ModelLinkPtr pLink;

   // First create a new Draft document (default parameters that are
   // not specified are handled just as in VB)
   pDftDoc = pSEApp->GetDocuments()->Add(L"SolidEdge.DraftDocument");

   if (pDftDoc)
   {
   // Link the newly created assembly file to this draft document
   pLink = pDftDoc->GetModelLinks()->Add("c:\\block.asm");

   if (pLink)
   {
   // Now create a drawing view using the model link above
   pDftDoc->GetActiveSheet()->GetDrawingViews()->Add(pLink,
   SolidEdgeDraft::igTopBackRightView, 1.0, 0.1, 0.1);
   }
   else
```

```
    {
    hr = E_FAIL;
    }

    // Finally, save the drawing file.
    pDftDoc->SaveAs("c:\\block.dft");

    }
    else
    {
    hr = E_FAIL;
    }
    return hr;

}
```

Glossary

ActiveX automation

A technology that allows any object to expose a set of commands and functions that some other piece of code can invoke. Automation is intended to allow applications to create system macro programming tools.

API

The acronym for Application Programming Interface. An API is a collection of formal definitions that describe the funciton calls and other protocols a program uses to communicate with the outside world.

argument

A value passed to a procedure. The value can be a constant, variable, or expression.

assembly

A group of parts specified as a single component. The parts in an assembly are usually individually specified elsewhere, combined according to requirements, and physically connected. When the assembly is a unit to be used in higher level assemblies, it is called a subassembly; when the assembly is a product, it is a top-level assembly.

associative

A condition in which an element is related to another element.

available

A condition in which a document can be accessed by a user for review or revision.

base feature

A feature that defines the basic part shape. Solid models are constructed by adding material to and removing material from a base feature.

chamfer

A corner that has been cut at an angle.

class

The definition of a data structure and the functions that manipulate that structure. C++ classes are generally defined in include files.

collection

A special type of object whose purpose is to provide methods of creating objects and also providing a way of accessing all the objects of a specific type.

COM

The acronym for the Component Object Model. This model specifies a binary standard for object implementation that is independent of the programming language you decide to use. This binary standard lets two applications communicate through object-oriented interfaces without requiring either to know anything about the other's implementation. It is supported at the operating system level and is the basis of all Microsoft products.

command stacking

The process of temporarily deactivating a command without terminating it.

compound document

A document that contains files with various formats. For example, a Word document that has a Solid Edge file embedded in it.

container

A document that contains documents created with other applications. Through ActiveX, you can access the application that created the document and link and embed a document created by another application.

cutout

A feature created by removing material from a part by extrusion, revolution, sweeping, or lofting. A profile defines the feature's shape.

dimension

A control that assigns and maintains a dimensional value to an individual element or establishes a dimensional relationship between multiple elements. Dimensions are represented graphically by a label consisting of text, lines, and arrows.

dimension axis

An axis for dimension orientation that you define by selecting a line. You can place linear dimensions that run parallel or perpendicular to the axis. By default, dimensions are placed horizontally or vertically.

dimension group

A series of dimensions. You can place a chained dimension group, a coordinate dimension group, or a stacked dimension group.

driven dimension

A dimension whose value depends on the value of other dimensions or elements.

driving dimension

A dimension whose value controls the size, orientation, or location of an element.

element

A single, selectable unit. You can select geometric elements, dimensions, annotations, objects placed in the drawing through ActiveX, and so forth. The type of element that can be selected is determined by command context.

embed

A method for inserting information from a source document into the active document. Once embedded, the information becomes part of the active document; if changes are made to the source document, the updates are not reflected in the active document.

enumerator

An object that iterates through a sequence of items.

feature

A characteristic of a part that is usually created by adding material to or removing material from the basic part shape. Features include holes, cutouts, protrusions, and so forth.

host object

Any object in Solid Edge to which you have attached an attribute set.

iterator

A statement that sequences through a set of objects for a specified number of times or until a certain condition is met. The three iteration statements in C++ are while, do, and for. In Visual Basic, two iteration statements are for, next and do, while. See also enumerator.

key point

A recognizable point on an element. Key points include vertices, mid points, center points, and so forth.

link

A method for inserting information stored in a source document into the active document. The two documents are connected, and changes made in the source document are reflected in the active document.

macro

A sequence of actions or commands that can be named and stored. When you run the macro, the software performs the actions or runs the commands.

method

Any operation that can be performed on an object.

native data

All the information that an application requires to edit an object.

object

Information that can be linked or embedded into an ActiveX-compliant product.

object browser

A feature of Microsoft Visual Basic that allows you to examine the contents of a specified object library to access information about the objects in that library.

object hierarchy (object model)

A diagram that shows the relationships among objects.

object persistance

A method for saving a complex network of objects in a permanent binary form, usually disk storage, that persists after those objects are deleted from memory.

parent feature

A feature upon which another feature is based. For example, a circular pattern feature is created by copying other features in a circular arrangement. The copied feature is the parent of the circular pattern feature.

profile

A 2-D set of variational elements used to construct a feature.

property

A unique characteristic of an element or object in a file. The characteristics can include the name, parent, application, and so forth.

reference element

An element that is not included in the part model. Reference elements, such as profile planes and centerline axes, are used for construction only.

reference plane

A flat, rectangular, infinite surface whose position and orientation provide a frame of reference for creating and working with profiles in 3-D space.

routing slip

A list that indicates the users that should receive electronic mail containing an attached document.

selection set

A single selected object or a group of selected objects.

server

The application that created objects that are linked or embedded in a compound document (container).

share embed

A document copied directly into a drawing. When you embed the same document more than once in the same document, the document elements are copied each time. When you share embed the same document more than once in the same document, the documents are not copied each time. Instead, the other documents reference the initial placement of the document.

SmartFrame

A placeholder for an ActiveX object. The contained ActiveX object can be 2-D or 3-D, but will be mapped to 2-D space. SmartFrames are generally rectangles on a sheet that enclose embedded or linked object(s) and have some intelligence about how to deal with the data in that frame.

symbol

A document placed in a drawing. You can override and edit the properties and style of the symbol. A document can be linked, embedded, shared embedded, or inserted as elements.

toolbar

Toolbars are menus that allow you to quickly access commands. You can define custom toolbars or use the toolbars delivered with the software.

type library (object library)

A standalone file or a component within another file (typically a DLL) containing type information that is used by automation controllers such as Visual Basic to find and invoke the properties and methods on an object.

window

An area defined by a standard border and buttons that is used to display information in an application.

Index