Solid Edge – Siemens PLM Software

.NET Programmer's Guide

Solid Edge with Synchronous Technology API

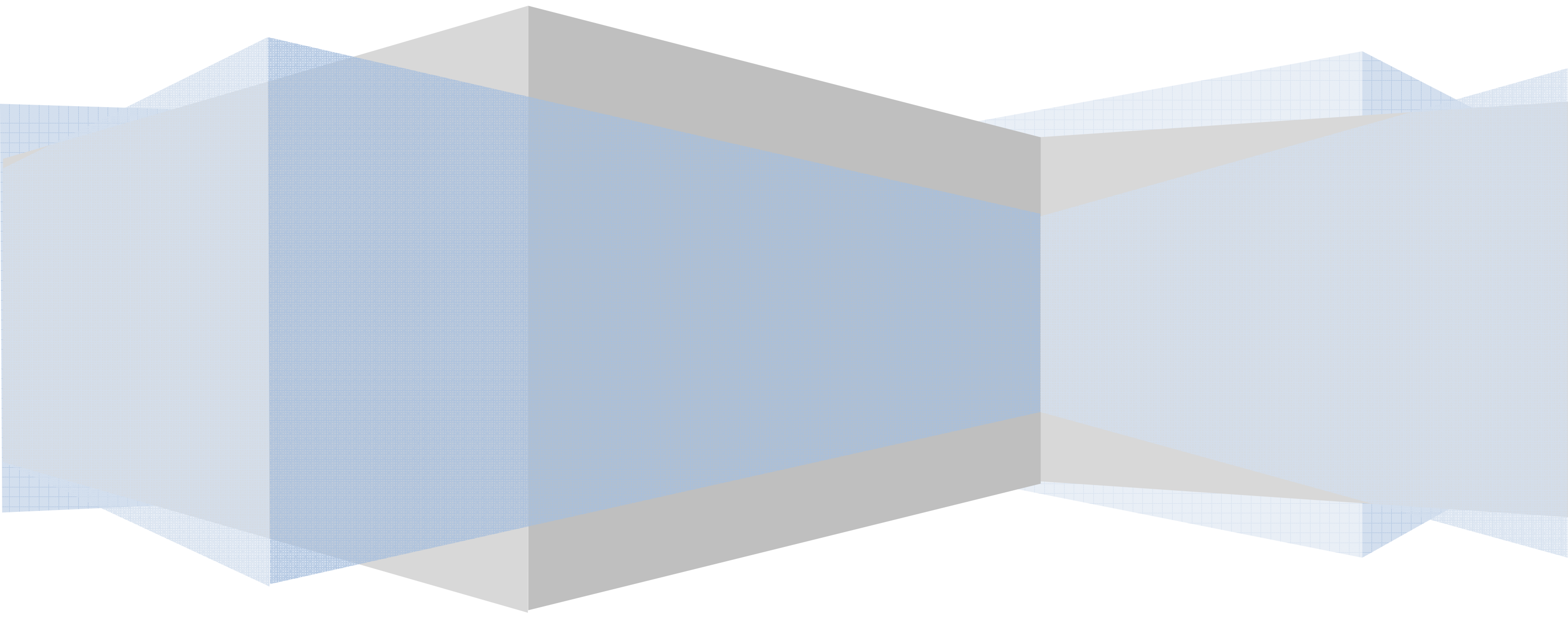


Table of Contents

Chapter 1 - Introduction

Welcome to the .NET Programmer's Guide for Solid Edge. This book was written in an effort to enable .NET developers to quickly get up to speed with automating Solid Edge. Learning the Solid Edge API can be a monumental task in itself. It takes most people several years before they feel comfortable with the API. The most important thing to remember is that it takes time and effort to fully understand and appreciate all of the techniques that this book will cover.

While there are many .NET programming \ scripting languages that can be used, I will focus primarily on Visual Basic .NET and C#. Other than syntax, there are relatively few differences between Visual Basic.NET and C#. A good example of this is the Microsoft Certified Application Developer (MCAD) exam. There is not a Visual Basic.NET version and C# version of the exam. There is only one version that asks questions for both languages. The reason is, from a .NET framework point of view, if you know how to use the framework in Visual Basic .NET, then you also know how to do it in C#. For this reason, focus of this book will be in the Visual Basic .NET environment with C# specific examples where necessary.

Who Should Read This Book

This book is for anyone wanting to learn how to automate Solid Edge using .NET programming languages. While prior programming experience will indeed help, it is not necessary. The goal of this book is to enable developers new to Solid Edge programming to quickly get started. There will be an abundance of source code examples to learn from.

Visual Basic 6.0 Users

If you are a Visual Basic 6.0 programmer and new to Visual Basic .NET, there are new concepts that you'll need to learn. While Visual Basic 6.0 was written with COM programming in mind, Visual Basic .NET was written with .NET in mind. COM and .NET are two completely different architectures that don't natively understand each other. They must interact via an "Interop" layer. This interop layer can be a source of much pain and frustration if not understood properly. One of the goals of this book is to save you a significant amount of time and frustration learning these new concepts.

It is normal for programmers new to Visual Basic .NET to expect the ability to upgrade their legacy code. While Microsoft does provide a Visual Basic Upgrade Wizard, it is often the case that the code does not work as expected after the conversion. There are well documented reasons on MSDN as to why this happens. While this can be very frustrating, you might consider this an opportunity to write your code from scratch, learning the new concepts along the way.

For a complete list of language changes, please refer to the following MSDN article:
"Language Changes for Visual Basic 6.0 Users "
http://msdn2.microsoft.com/en-us/library/skw8dhdd(VS.80).aspx.

Software Requirements

This book targets Solid Edge with Synchronous Technology and the Microsoft .NET Framework version 2.0. The .NET 2.0 Framework SDK is freely downloadable at http://msdn2.microsoft.com/downloads.

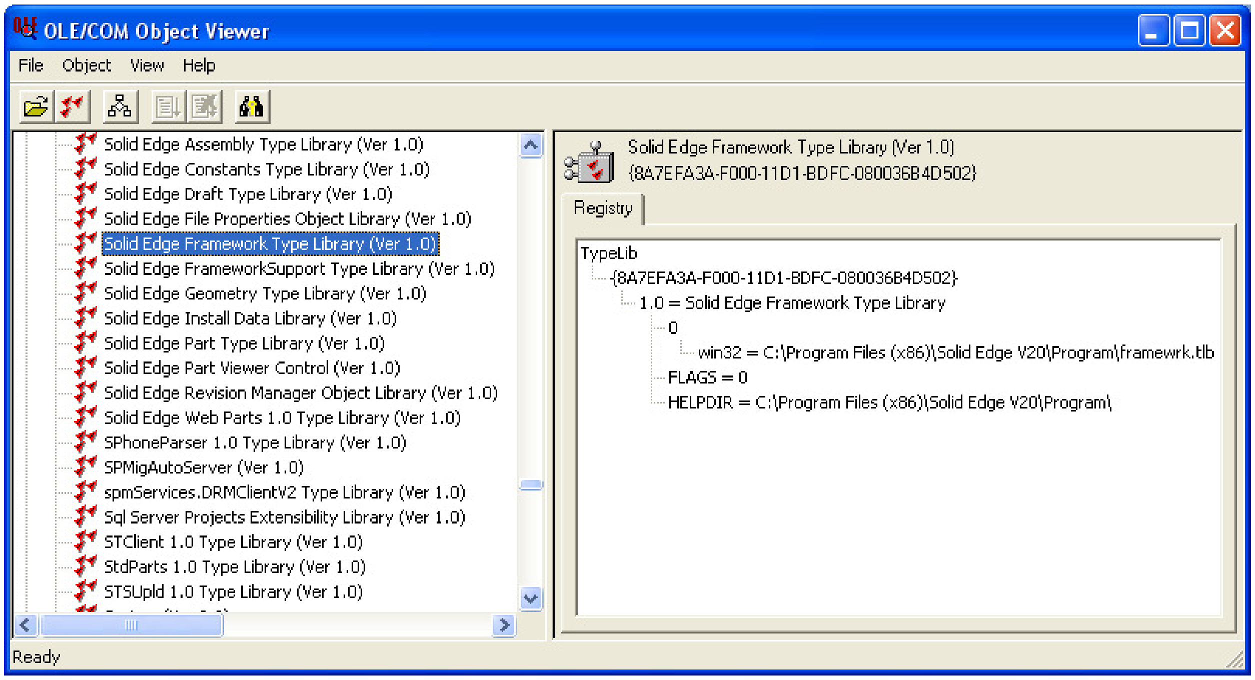
The examples and screen shots from this book will be from Microsoft Visual Studio 2005 Professional Edition. You may use any edition of Microsoft Visual Studio 2005 that meets your specific needs. While not covered in this book, you can also use alternative IDE's like SharpDevelop http://www.icsharpcode.net/OpenSource/SD/Default.aspx.

Chapter 2 - API Overview

This chapter introduces you to the Solid Edge COM API. While this book is focused on .NET technologies, you cannot ignore the fact that the API that you're working with is COM based. The majority of this chapter will discuss very little .NET. This chapter focuses on the aspects of COM technology that you have to understand in order to use it with .NET.

Solid Edge has a large and robust COM API. You can view these APIs with Microsoft's OLE/COM Object Viewer (oleview.exe). Oleview.exe can typically be found in the following path: "%VS80COMNTOOLS%bin\oleview.exe". It will tell you everything you need to know about what APIs are available and where they are located.

The information you see in the OLE/COM Object View is the rawest view of the Solid Edge type libraries that you can get. As you'll see later, .NET tends to obscure some of the API information that you need to do your job. When in doubt about API definitions, you should always refer to the type library information.



```
Solid Edge Assembly Type Library (Ver 1.0)
Solid Edge Constants Type Library (Ver 1.0)
Solid Edge Draft Type Library (Ver 1.0)
Solid Edge File Properties Object Library (Ver 1.0)
Solid Edge Framework Type Library (Ver 1.0)
Solid Edge FrameworkSupport Type Library (Ver 1.0)
Solid Edge Geometry Type Library (Ver 1.0)
Solid Edge Install Data Library (Ver 1.0)
Solid Edge Part Type Library (Ver 1.0)
Solid Edge Part Viewer Control (Ver 1.0)
Solid Edge Revision Manager Object Library (Ver 1.0)
Solid Edge Web Parts 1.0 Type Library (Ver 1.0)
SPhoneParser 1.0 Type Library (Ver 1.0)
SPMigAutoServer (Ver 1.0)
spmServices.DRMClientV2 Type Library (Ver 1.0)
Sql Server Projects Extensibility Library (Ver 1.0)
STClient 1.0 Type Library (Ver 1.0)
StdParts 1.0 Type Library (Ver 1.0)
STSUpld 1.0 Type Library (Ver 1.0)
```

```
Solid Edge Framework Type Library (Ver 1.0)
{8A7EFA3A-F000-11D1-BDFC-080036B4D502}

TypeLib
  {8A7EFA3A-F000-11D1-BDFC-080036B4D502}
    1.0 = Solid Edge Framework Type Library
      0
        win32 = C:\Program Files (x86)\Solid Edge V20\Program\framewrk.tlb
      FLAGS = 0
      HELPDIR = C:\Program Files (x86)\Solid Edge V20\Program\
```

Figure 2-1 - OLE/COM Object Viewer

Solid Edge Core Type Libraries

The Solid Edge core COM type libraries are the APIs that are available to automate the Solid Edge application. These APIs can be used by any programming or scripting language that supports COM.

You can find these type libraries in the installation path of Solid Edge under the “Program” folder. For example: “%PROGRAMFILES%\Solid Edge VXX\Program”.

Table of core APIs

Name	Description	Type Library
SolidEdgeFramework	Solid Edge Framework Type Library	framewrk.tlb
SolidEdgeFrameworkSupport	Solid Edge FrameworkSupport Type Library	fwksupp.tlb
SolidEdgePart	Solid Edge Part Type Library	part.tlb
SolidEdgeGeometry	Solid Edge Geometry Type Library	geometry.tlb
SolidEdgeAssembly	Solid Edge Assembly Type Library	assembly.tlb
SolidEdgeDraft	Solid Edge Draft Type Library	draft.tlb
SolidEdgeConstants	Solid Edge Constants Type Library	constant.tlb

SolidEdgeFramework Type Library (framewrk.tlb)

The SolidEdgeFramework type library is the root type library. It has no dependencies on any other type library. At a minimum, you must reference it to interact with Solid Edge with strongly typed objects. Most other core APIs have a dependency on this type library.

One of the most important interfaces that you will work with is the SolidEdgeFramework.Application interface. You must get a reference to this object before you can do any automation in Solid Edge. Typically, this comes in some form of GetObject() or CreateObject(). You can use the ProgID, *"SolidEdge.Application"* to get a reference to this object.

Another important interface is the SolidEdgeFramework.SolidEdgeDocument interface. You can use this interface to interrogate any Solid Edge document type. You will learn more about methods of obtaining these objects and how to use them in later chapters.

SolidEdgeFrameworkSupport Type Library (fwksupp.tlb)

The SolidEdgeFrameworkSupport type library contains interfaces that span all Solid Edge environments and are not environment specific.

SolidEdgePart Type Library (part.tlb)

The SolidEdgePart type library contains interfaces related to the following 3-D environments: Part, SheetMetal, and Weldment.

The most important interfaces in this type library are the SolidEdgePart.PartDocument, SolidEdgePart.SheetMetalDocument and SolidEdgePart.WeldmentDocument. These interfaces allow you to work with a specific document type in Solid Edge.

SolidEdgeGeometry Type Library (geometry.tlb)

The SolidEdgeGeometry type library contains interfaces that relate to geometry.

SolidEdgeAssembly Type Library (assembly.tlb)

The SolidEdgeAssembly type library contains interfaces related to the 3-D Assembly environment.

The most important interface in this type library is the SolidEdgeAssembly.AssemblyDocument interface. This interface allows you to work directly with assembly document types in Solid Edge.

SolidEdgeDraft Type Library (draft.tlb)

The SolidEdgeDraft type library contains interfaces related to the 2-D Draft environment.

The most important interface in this type library is the SolidEdgeAssembly.DraftDocument interface. This interface allows you to work directly with draft document types in Solid Edge.

SolidEdgeConstants Type Library (constant.tlb)

This type library is exclusively for global enumerations. Oleview.exe is a great way to view these enumerations and their values. You should be aware that some of the enumerations defined in this type library are duplicated in other type libraries.

Solid Edge Utility APIs

The Solid Edge Utility APIs allow you to work quickly with Solid Edge data without automating Solid Edge. Each of these lightweight APIs is focused for a specific area of tasks.

Table of utility APIs

Name	**Description**	**Type Library**
SEInstallDataLib	Solid Edge Install Data Library	SEInstallData.dll
SolidEdgeFileProperties	Solid Edge File Properties Object Library	PropAuto.dll
RevisionManager	Solid Edge Revision Manager Object Library	RevMgr.tlb

SEInstallDataLib (SEInstallData.dll)

The SEInstallDataLib API is used to safely harvest installation information about Solid Edge from the Windows registry. It contains only one class, the SEInstallData class. This class has straight forward named methods to get installation information.

You can use the ProgID “*SolidEdge.InstallData*” to create an instance of the SEInstallData class.

SolidEdgeFileProperties (PropAuto.dll)

The SolidEdgeFileProperties API is used to read and write Solid Edge file properties outside of Solid Edge. This can be extremely useful when you need to quickly work with a large dataset of files. Solid Edge uses standard Microsoft Structured Storage API to store properties.

You can learn more about Microsoft’s Structured Storage API on MSDN at:
http://msdn2.microsoft.com/en-us/library/aa380369(VS.80).aspx

RevisionManager (RevMgr.tlb)

The RevisionManager API can be used for several purposes.

- Manage file properties
- Manage link information
- Integrate with Insight

You can use the ProgID '*RevisionManager.Application*' to create an instance of the Application class.

Chapter 3 - .NET Overview

This chapter introduces the .NET framework and discusses the more important parts of the framework in regards to Solid Edge.

The Microsoft .NET framework is a platform for building, deploying, and running Web Services and applications. It provides a highly productive, standards-based, multi-language environment for integrating existing investments with next-generation applications and services.

You can learn more about Microsoft's .NET Framework on MSDN at:
http://msdn2.microsoft.com/netframework

Terminology

Application Domain

Application Domains are isolated boundaries of memory allocated by the framework for applications to execute inside of. When your programs execute, the framework will create a “Default AppDomain” for your application.

Assembly

Assemblies are the building blocks of .NET framework applications. When you compile a .NET application, you are building an assembly.

COM Interop

The .NET framework is a natural progression from COM because the two models share many central themes, including component reuse and language neutrality. For backward compatibility, COM interop provides access to existing COM components without requiring that the original component be modified. You can incorporate COM components into a .NET Framework application by using COM interop tools to import the relevant COM types. Once imported, the COM types are ready to use.

When you add a reference to a Solid Edge type library, Visual Studio .NET automatically build a .NET assembly for use in your project. This assembly will contain all of the type library definitions in a format that can be consumed from any .NET programming language. These generated assemblies are referred to as “Interop Assemblies”.

Garbage Collection

The .NET framework employs a non-deterministic approach to freeing memory whereas languages like Visual Basic 6.0 or C++ use a deterministic approach. This means that when you free a reference to an object, the memory allocated for the object is not immediately freed but rather is flagged as being available for garbage collection. Garbage collection occurs when the framework identifies an opportune time to perform the actual collection. You have no control of when this occurs nor will you be notified when it happens.

In regards to COM interop, this can lead to trouble releasing COM objects in a timely manner.

Interop Assemblies

Interop Assemblies are generated by the Type Library Importer (Tlbimp.exe) tool. Visual Studio .NET automatically generates interop assemblies for you when you reference COM type libraries. Once create this reference, your application will depend upon the interop assembly for execution.

Marshal Class

The Marshal class is a utility class for interacting with unmanaged code. While the scope of the class is rather large, you will be primarily interested in its interop services. Specifically, the Marshal.ReleaseComObject() method is what you will use to free references to COM objects.

Runtime Callable Wrapper (RCW)

The .NET runtime exposes COM objects through a proxy called the runtime callable wrapper. Although the RCW appears to be an ordinary object to .NET clients, its primary function is to marshal calls between a .NET client and a COM object. The runtime creates exactly one RCW for each COM object within an application domain.

In Visual Basic 6.0, you could simply set an object equal to “Nothing” to release the COM reference. In .NET, you must use a Marshal.ReleaseComObject() call on each and every COM object that you have a reference to in order to decrement the reference count on the RCW. It is only when the RCW reference count reaches zero will the actual COM reference be released.

To apply this concept to Solid Edge programming terms, say you have a simple reference to a Solid Edge document. You use the object in a function, set the object equal to Nothing, and your function completes. You won’t see it, but the .NET framework created a RCW for the object and, because of garbage collection, the RCW is still holding a reference to the COM object. Because you didn’t call Marshal.ReleaseComObject(), Solid Edge thinks that the object is still in use. Depending on the scenario, Solid Edge may crash as it is attempting to delete the COM object in memory but you still have an outstanding reference to it via the RCW.

It is this fundamental concept that Solid Edge programmers using .NET need to fully understand.

Chapter 4 - Getting Started

In this chapter, you will begin writing actual .NET code to interact with Solid Edge. Along the way, you'll take a look at what's really going on under the hood of Visual Studio. I've touched on the fact that COM and .NET do not natively understand each other. How is it possible to write a .NET application that uses a COM API? All of this will become clear as you proceed forward.

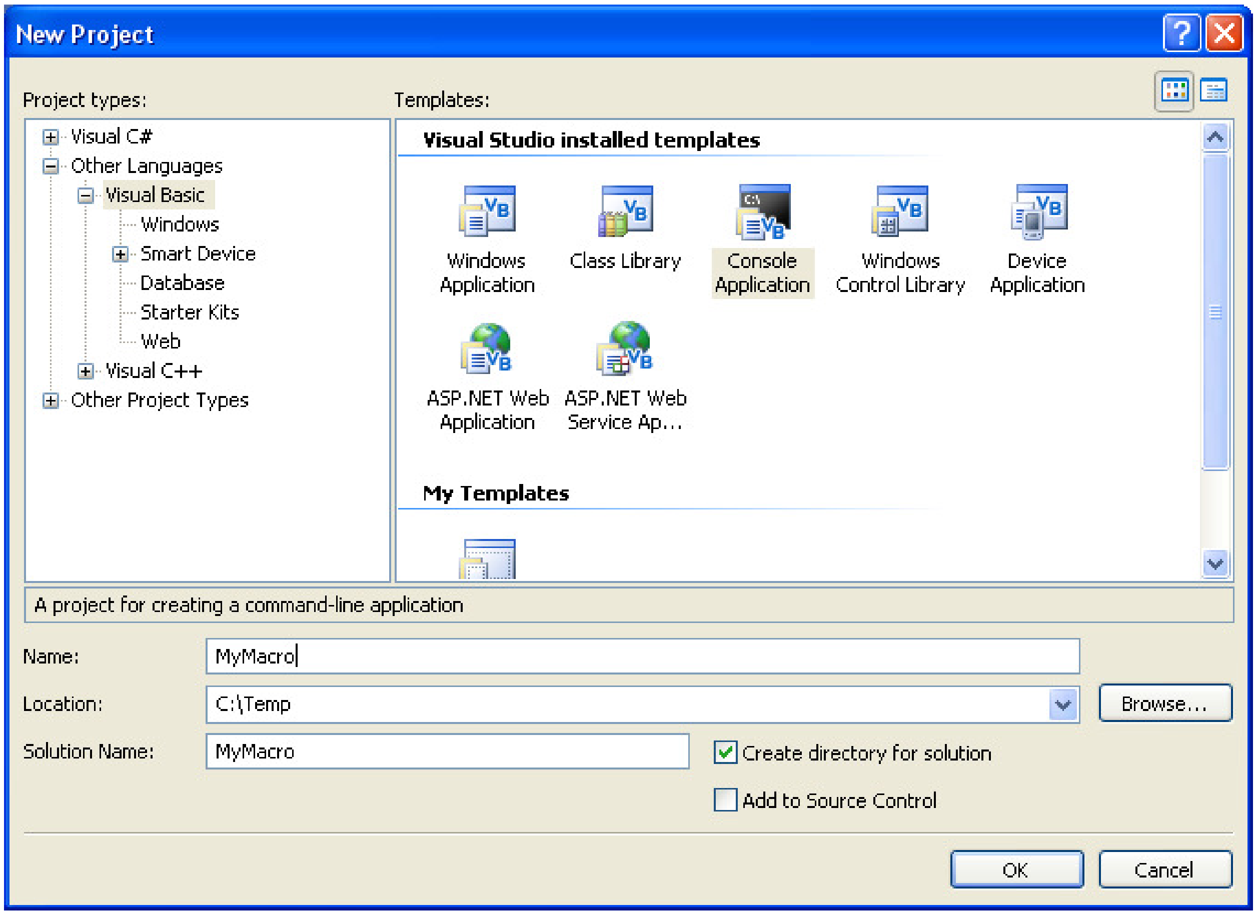
The most important concept to grasp at this time is the Interop concept. You need to understand that .NET objects do not communicate directly to COM objects and vice-versa. There will always be an Interop layer doing the communication and COM reference counting for us. For the most part, this Interop layer is discretely hidden from us but you still need to have a general understanding of how it works.

When working with COM based programming languages like Visual Basic 6.0 or Visual C++, it was enough to simply set a reference equal to Nothing or NULL. In .NET, we must also utilize the Marshal.ReleaseComObject() method to decrement the Runtime Callable Wrapper (RCW) reference count.

Your first macro

Create a new Visual Basic .NET project

Visual Studio .NET offers many different templates for different project types. Depending on your needs, the template that you choose will vary. If you simply want to automate Solid Edge without any user interaction, the Console Application template generally works well.



A project for creating a command-line application

Name: MyMacro

Location: C:\Temp

Solution Name: MyMacro

[✓] Create directory for solution

[ ] Add to Source Control

Figure 4-1 - New Project

This creates an empty console application project that you can use to begin your work.

Adding a reference to Solid Edge API

Before you can start working with the Solid Edge API, you'll need to add a reference to the type libraries that you'll be using. You can begin by selecting Project -> Add Reference from the main menu in Visual Studio .NET.

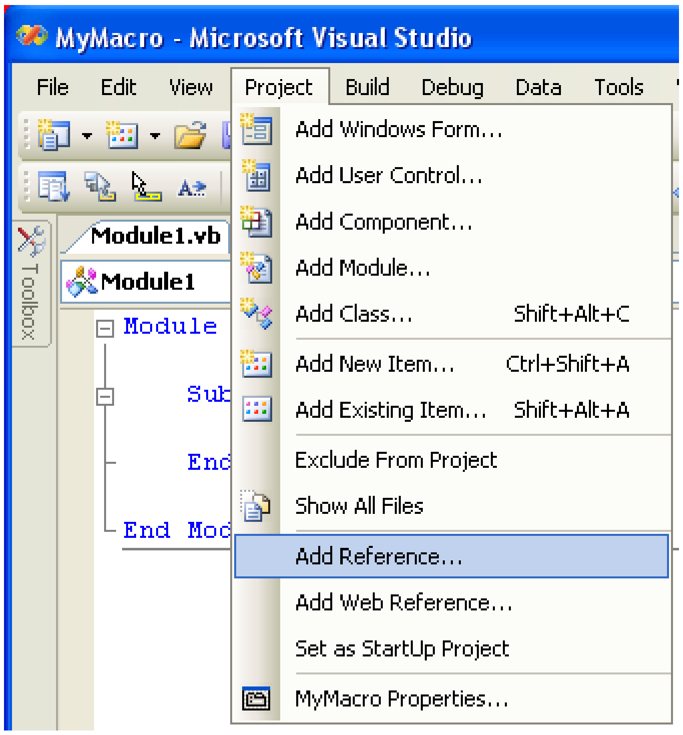
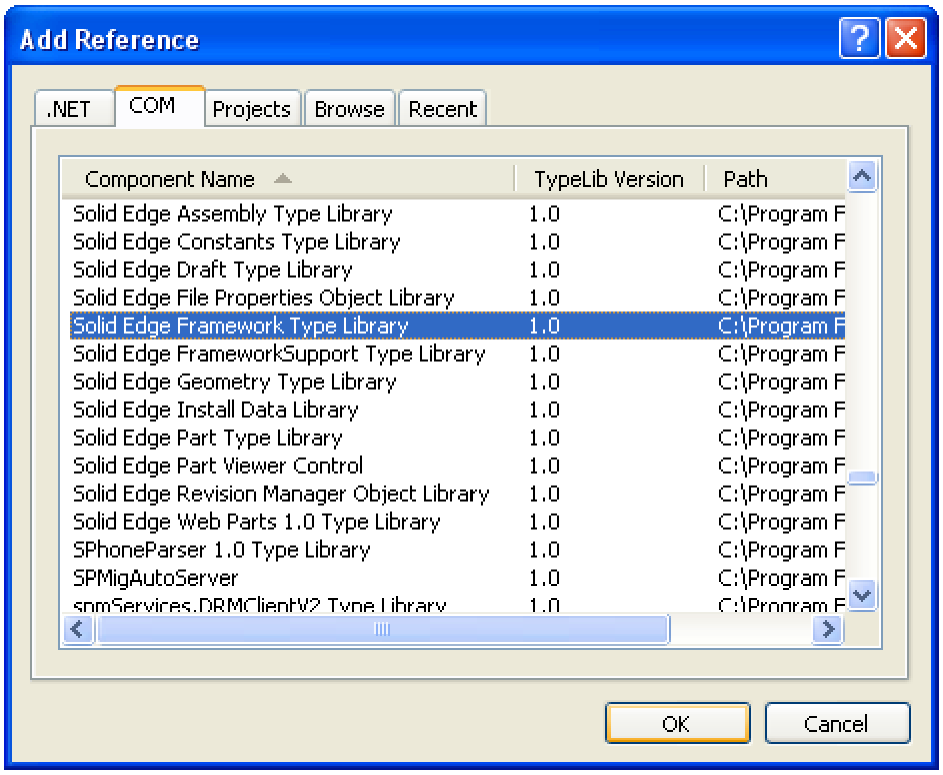


Figure 4-2 - Add Reference

The Add Reference dialog window will appear. You will first need to click the COM tab. Scroll down until you get to the Solid Edge Type Libraries. Select the Solid Edge Framework Type Library as shown below.



Component Name	TypeLib Version	Path
Solid Edge Assembly Type Library	1.0	C:\Program F
Solid Edge Constants Type Library	1.0	C:\Program F
Solid Edge Draft Type Library	1.0	C:\Program F
Solid Edge File Properties Object Library	1.0	C:\Program F
Solid Edge Framework Type Library	1.0	C:\Program F
Solid Edge FrameworkSupport Type Library	1.0	C:\Program F
Solid Edge Geometry Type Library	1.0	C:\Program F
Solid Edge Install Data Library	1.0	C:\Program F
Solid Edge Part Type Library	1.0	C:\Program F
Solid Edge Part Viewer Control	1.0	C:\Program F
Solid Edge Revision Manager Object Library	1.0	C:\Program F
Solid Edge Web Parts 1.0 Type Library	1.0	C:\Program F
SPhoneParser 1.0 Type Library	1.0	C:\Program F
SPMigAutoServer	1.0	C:\Program F
spmServices.DRMClientV2 Type Library	1.0	C:\Program F

Figure 4-3 - Add Reference

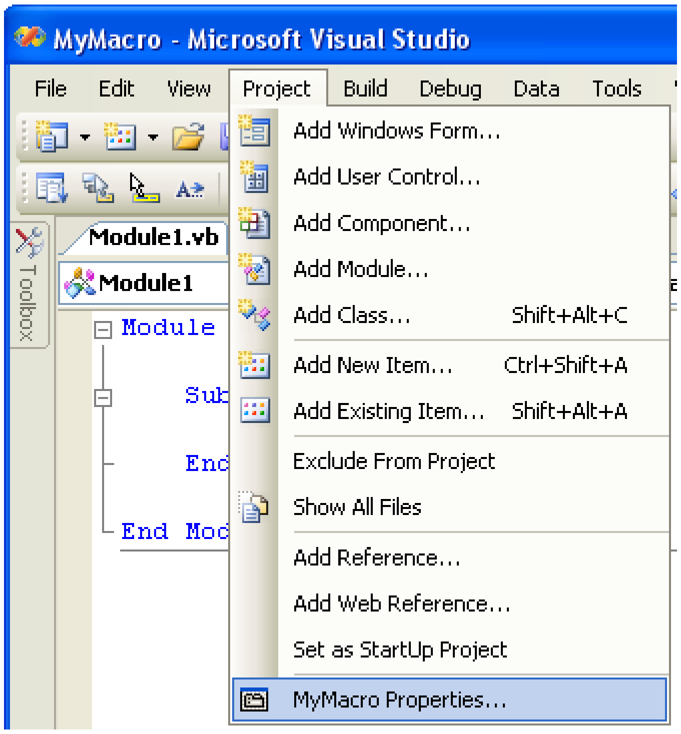
Visual Basic 6.0 users will immediately recognize this as a step they've had to do in the past. While the steps appear similar, the actual details of what's going on under the hood are quite different. Because Visual Basic 6.0 was written to be a COM programming language, it could reference existing COM Type Libraries directly.

When you add a COM reference in .NET, Visual Studio .NET actually calls a Type Library Importer program that generates an "Interop Assembly" for your project to use. This also means that your executable will forever and always dependent upon the generated interop assembly. If you plan on deploying your application to users, you'll need to be sure to include any interop assemblies that you've generated for your project.

If you wish to learn more about the Type Library Importer, please reference MSDN documentation: http://msdn2.microsoft.com/en-us/library/tt0cf3sx(VS.80).aspx

Viewing Interop Assembly References

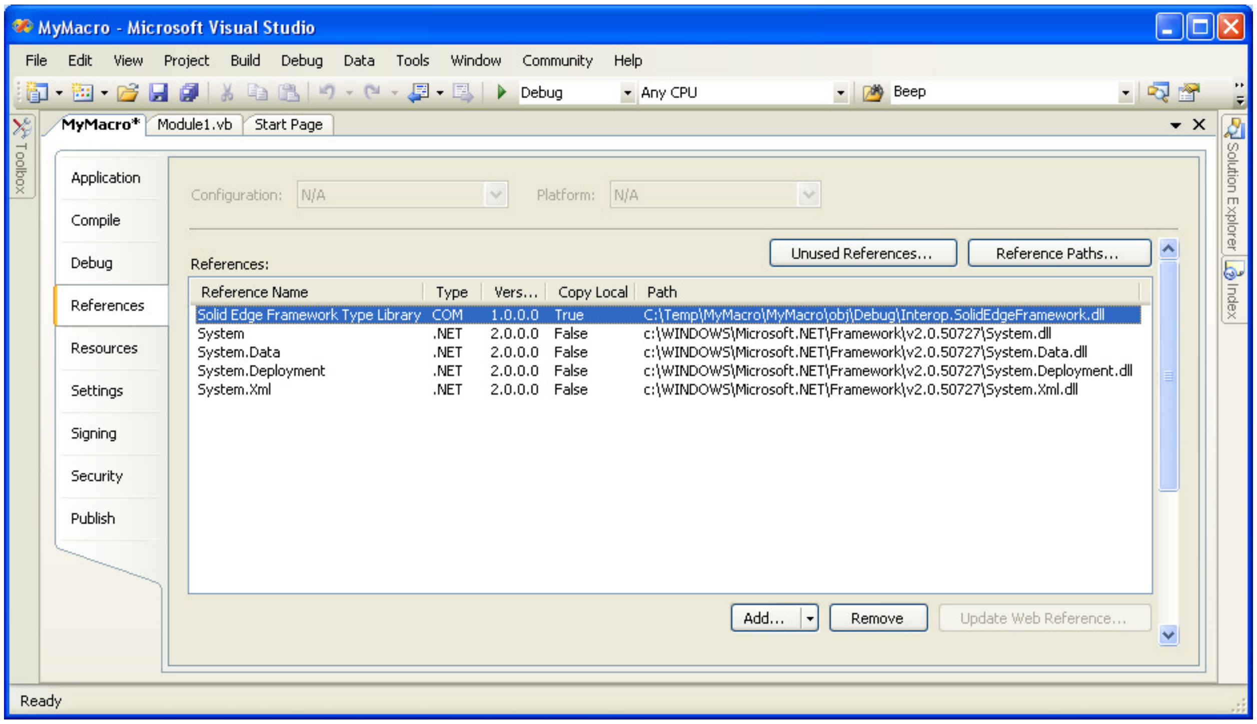
Now that you know how to add Solid Edge API support to your project, let see how you can view your project's dependencies, specifically interop assemblies. Select Project -> <Project Name> Properties from the main menu in Visual Studio .NET.



```
Module
    Sub
    End
End Mod
```

Figure 4-4 - Project Properties

Once the project properties appear, select the References tab. Here you can see the interop assembly, Interop.SolidEdgeFramework.dll, that the Type Library Importer created for your project. Your executable is dependent upon the references that you see in this window.



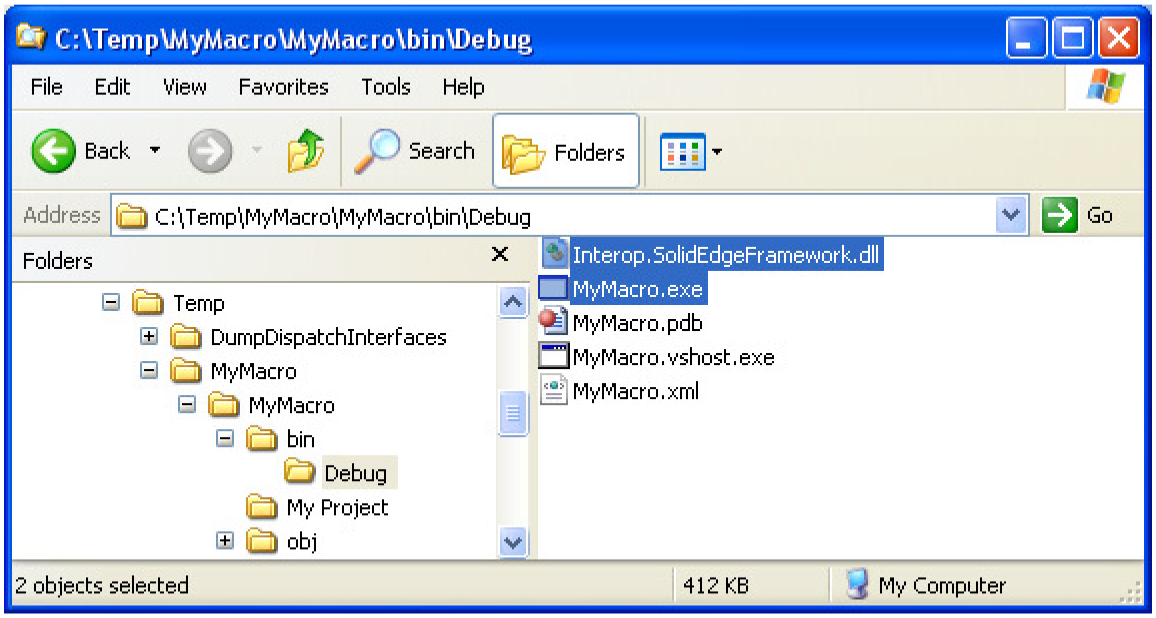
Configuration: N/A Platform: N/A

References:

Reference Name	Type	Vers...	Copy Local	Path
Solid Edge Framework Type Library	COM	1.0.0.0	True	C:\Temp\MyMacro\MyMacro\obj\Debug\Interop.SolidEdgeFramework.dll
System	.NET	2.0.0.0	False	c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.dll
System.Data	.NET	2.0.0.0	False	c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Data.dll
System.Deployment	.NET	2.0.0.0	False	c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Deployment.dll
System.Xml	.NET	2.0.0.0	False	c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Xml.dll

Figure 4-5 - Project References

For generated interop assemblies like the Interop.SolidEdgeFramework.dll, they will have the "Copy Local" flag set to True. This means that these .dll's will be placed in the projects output folder when you build the project.



C:\Temp\MyMacro\MyMacro\bin\Debug

Interop.SolidEdgeFramework.dll
MyMacro.exe
MyMacro.pdb
MyMacro.vshost.exe
MyMacro.xml

2 objects selected 412 KB

Figure 4-6 - Build Folder

Connecting to Solid Edge (Visual Basic.NET)

You may still use the familiar `GetObject()` method from the Visual Basic 6 days if you'd like. In Visual Basic .NET, `GetObject()` simply calls `Marshal.GetActiveObject()`. This is important to understand because GetObject() is a language specific feature of Visual Basic .NET. No other .NET language has a comparable feature. The other .NET languages must rely on Marshal.GetActiveObject().

```
Imports System.Runtime.InteropServices
Module Module1
  Sub Main()
    Dim objApp As SolidEdgeFramework.Application = Nothing
    Try
      'Old VB6 Syntax
      'objApp = GetObject(, "SolidEdge.Application")
      'New Visual Basic.NET Syntax

      ' Connect to a running instance of Solid Edge
      objApp = Marshal.GetActiveObject("SolidEdge.Application")
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objApp Is Nothing) Then
        Marshal.ReleaseComObject(objApp)
        objApp = Nothing
      End If
    End Try
  End Sub
End Module
```

Connecting to Solid Edge (C#)

```
using System;
using System.Runtime.InteropServices;
namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

Starting Solid Edge (Visual Basic .NET)

You may still use the familiar CreateObject() method from the Visual Basic 6 days if you'd like. In Visual Basic .NET, CreateObject() simply calls Activator.CreateInstance(). This is important to understand because CreateObject() is a language specific feature of Visual Basic .NET. No other .NET language has a comparable feature. The other .NET languages must rely on Activator.CreateInstance().

```
Imports System.Runtime.InteropServices
Module Module1
  Sub Main()
    Dim objApp As SolidEdgeFramework.Application
    Dim objType As Type
    Try
      'Old VB6 Syntax
      'objApp = CreateObject("SolidEdge.Application")
      'New Visual Basic.NET Syntax

      ' Get the type from the Solid Edge ProgID
      objType = Type.GetTypeFromProgID("SolidEdge.Application")

      ' Start Solid Edge
      objApp = Activator.CreateInstance(objType)

      ' Make Solid Edge visible
      objApp.Visible = True
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objApp Is Nothing) Then
        Marshal.ReleaseComObject(objApp)
        objApp = Nothing
      End If
    End Try
  End Sub
End Module
```

Starting Solid Edge (C#)

```
using System;
using System.Runtime.InteropServices;
namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      Type type = null;
      try
      {
        // Get the type from the Solid Edge ProgID
        type = Type.GetTypeFromProgID("SolidEdge.Application");

        // Start Solid Edge
        application = (SolidEdgeFramework.Application)
          Activator.CreateInstance(type);

        // Make Solid Edge visible
        application.Visible = true;
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

Working with Documents

The following lists the classes (also called ProgIDs) associated with Solid Edge and each type of document environment. These class names can be used in the Visual Basic function CreateObject and can also be used in the Add method of the Documents object.

All Solid Edge document objects implement the `SolidEdgeFramework.SolidEdgeDocument` interface. This generic interface allows document ambiguity when you don't know what type of document that you're working with.

Table of document ProgIds

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

Creating Documents Example (Visual Basic.NET)

```
Imports System.Runtime.InteropServices
Module Module1
  Sub Main()
    Dim objApp As SolidEdgeFramework.Application = Nothing
    Dim objDocuments As SolidEdgeFramework.Documents = Nothing
    Dim objAssembly As SolidEdgeAssembly.AssemblyDocument = Nothing
    Dim objDraft As SolidEdgeDraft.DraftDocument = Nothing
    Dim objPart As SolidEdgePart.PartDocument = Nothing
    Dim objSheetMetal As SolidEdgePart.SheetMetalDocument = Nothing
    Dim objWeldment As SolidEdgePart.WeldmentDocument = Nothing
    Dim objType As Type

    Try
      ' Get the type from the Solid Edge ProgID
      objType = Type.GetTypeFromProgID("SolidEdge.Application")

      ' Start Solid Edge
      objApp = Activator.CreateInstance(objType)

      ' Make Solid Edge visible
      objApp.Visible = True

      ' Turn off alerts.  Weldment environment will display a warning
      objApp.DisplayAlerts = False

      ' Get a reference to the Documents collection
      objDocuments = objApp.Documents

      ' Create an instance of each document environment
      objAssembly = objDocuments.Add("SolidEdge.AssemblyDocument")
      objDraft = objDocuments.Add("SolidEdge.DraftDocument")
      objPart = objDocuments.Add("SolidEdge.PartDocument")
      objSheetMetal = objDocuments.Add("SolidEdge.SheetMetalDocument")
      objWeldment = objDocuments.Add("SolidEdge.WeldmentDocument")

      ' Turn alerts back on
      objApp.DisplayAlerts = True
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objAssembly Is Nothing) Then
        Marshal.ReleaseComObject(objAssembly)
        objAssembly = Nothing
      End If
      If Not (objDraft Is Nothing) Then
        Marshal.ReleaseComObject(objDraft)
        objDraft = Nothing
      End If
      If Not (objPart Is Nothing) Then
        Marshal.ReleaseComObject(objPart)
        objPart = Nothing
      End If
      If Not (objSheetMetal Is Nothing) Then
        Marshal.ReleaseComObject(objSheetMetal)
        objSheetMetal = Nothing
      End If
      If Not (objWeldment Is Nothing) Then
        Marshal.ReleaseComObject(objWeldment)
        objWeldment = Nothing
      End If
      If Not (objDocuments Is Nothing) Then
        Marshal.ReleaseComObject(objDocuments)
        objDocuments = Nothing
      End If
      If Not (objApp Is Nothing) Then
        Marshal.ReleaseComObject(objApp)
        objApp = Nothing
      End If
    End Try
  End Sub
End Module
```

Creating Documents Example (C#)

```
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.Documents documents = null;
      SolidEdgeAssembly.AssemblyDocument assembly = null;
```

```
SolidEdgeDraft.DraftDocument draft = null;
SolidEdgePart.PartDocument part = null;
SolidEdgePart.SheetMetalDocument sheetmetal = null;
SolidEdgePart.WeldmentDocument weldment = null;
Type type = null;

try
{
  // Get the type from the Solid Edge ProgID
  type = Type.GetTypeFromProgID("SolidEdge.Application");

  // Start Solid Edge
  application = (SolidEdgeFramework.Application)
    Activator.CreateInstance(type);

  // Make Solid Edge visible
  application.Visible = true;

  // Turn off alerts.  Weldment environment will display a warning
  application.DisplayAlerts = false;

  // Get a reference to the Documents collection
  documents = application.Documents;

  // Create an instance of each document environment
  assembly = (SolidEdgeAssembly.AssemblyDocument)
    documents.Add("SolidEdge.AssemblyDocument", Missing.Value);
  draft = (SolidEdgeDraft.DraftDocument)
    documents.Add("SolidEdge.DraftDocument", Missing.Value);
  part = (SolidEdgePart.PartDocument)
    documents.Add("SolidEdge.PartDocument", Missing.Value);
  sheetmetal = (SolidEdgePart.SheetMetalDocument)
    documents.Add("SolidEdge.SheetMetalDocument", Missing.Value);
  weldment = (SolidEdgePart.WeldmentDocument)
    documents.Add("SolidEdge.WeldmentDocument", Missing.Value);

  // Turn alerts back on
  application.DisplayAlerts = true;
}
catch (System.Exception ex)
{
  Console.WriteLine(ex.Message);
}
finally
{
  if (assembly != null)
  {
    Marshal.ReleaseComObject(assembly);
    assembly = null;
  }
  if (draft != null)
  {
    Marshal.ReleaseComObject(draft);
    draft = null;
  }
  if (part != null)
  {
          Marshal.ReleaseComObject(part);
          part = null;
        }
        if (sheetmetal != null)
        {
          Marshal.ReleaseComObject(sheetmetal);
          sheetmetal = null;
        }
        if (weldment != null)
        {
          Marshal.ReleaseComObject(weldment);
          weldment = null;
        }
        if (documents != null)
        {
          Marshal.ReleaseComObject(documents);
          documents = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

Determining Document Type Example (Visual Basic.NET)

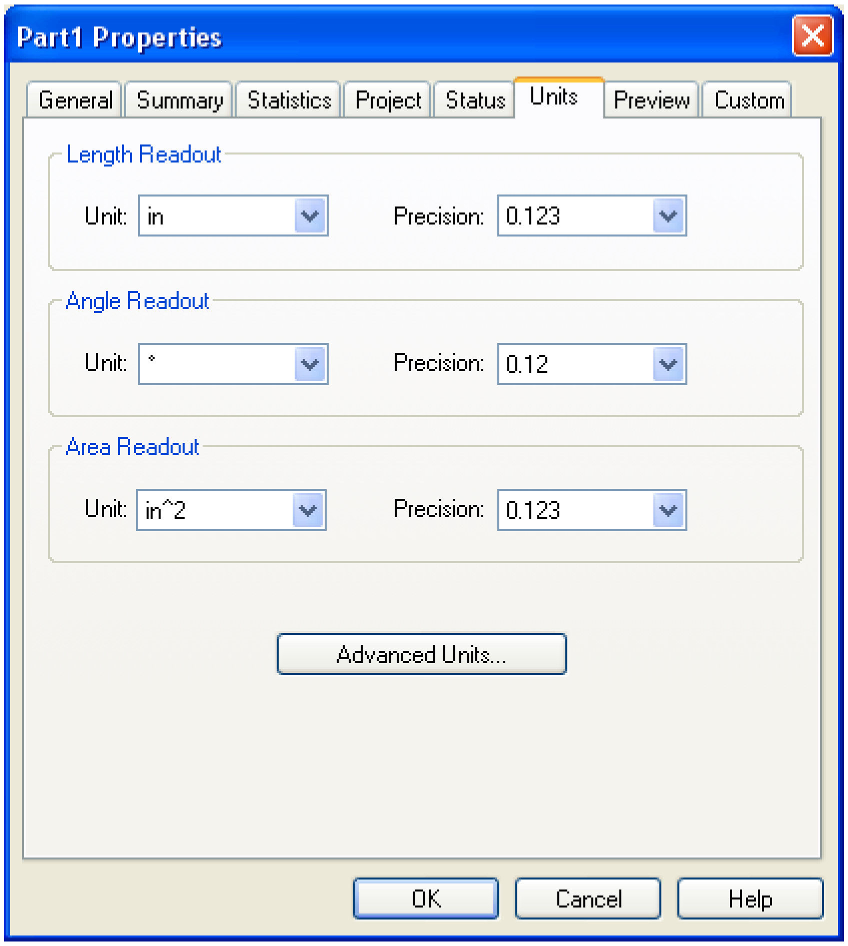
```
Imports System.Runtime.InteropServices
Imports SolidEdgeFramework

Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objDocument As SolidEdgeFramework.SolidEdgeDocument = Nothing
    Try
      ' Connect to a running instance of Solid Edge.
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the active document
      objDocument = objApplication.ActiveDocument

      ' Using Type property, determine document type
      Select Case objDocument.Type
        Case DocumentTypeConstants.igAssemblyDocument
          Console.WriteLine("Assembly Document")
        Case DocumentTypeConstants.igDraftDocument
          Console.WriteLine("Draft Document")
        Case DocumentTypeConstants.igPartDocument
          Console.WriteLine("Part Document")
        Case DocumentTypeConstants.igSheetMetalDocument
          Console.WriteLine("SheetMetal Document")
        Case DocumentTypeConstants.igUnknownDocument
          Console.WriteLine("Unknown Document")
        Case DocumentTypeConstants.igWeldmentAssemblyDocument
          Console.WriteLine("Weldment Assembly Document")
        Case DocumentTypeConstants.igWeldmentDocument
          Console.WriteLine("Weldment Document")
      End Select
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objDocument Is Nothing) Then
        Marshal.ReleaseComObject(objDocument)
        objDocument = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Determining Document Type Example (C#)

```
using SolidEdgeFramework;
using System;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.SolidEdgeDocument document = null;
      try
      {
        // Connect to a running instance of Solid Edge.
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the active document
        document = (SolidEdgeFramework.SolidEdgeDocument)
          application.ActiveDocument;

        // Using Type property, determine document type
        switch (document.Type)
        {
          case DocumentTypeConstants.igAssemblyDocument:
            Console.WriteLine("Assembly Document");
            break;
          case DocumentTypeConstants.igDraftDocument:
            Console.WriteLine("Draft Document");
            break;
          case DocumentTypeConstants.igPartDocument:
            Console.WriteLine("Part Document");
            break;
          case DocumentTypeConstants.igSheetMetalDocument:
            Console.WriteLine("SheetMetal Document");
            break;
          case DocumentTypeConstants.igUnknownDocument:
            Console.WriteLine("Unknown Document");
            break;
          case DocumentTypeConstants.igWeldmentAssemblyDocument:
            Console.WriteLine("Weldment Assembly Document");
            break;
          case DocumentTypeConstants.igWeldmentDocument:
            Console.WriteLine("Weldment Document");
            break;
        }
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (document != null)
        {
          Marshal.ReleaseComObject(document);
          document = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

Chapter 5 - Units of Measure

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: in
Precision: 0.123

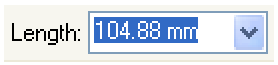
Angle Readout
Unit: °
Precision: 0.12

Area Readout
Unit: in^2
Precision: 0.123

5-1 - Units of Measure

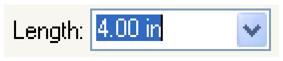
This is strictly a manipulation of the display of the precision; internally all measurements are stored at their full precision.

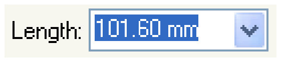
With a Length Readout precision of 0.12, the length of any linear measurement is displayed as follows:



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.



Converts to


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.

Internal Units

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.

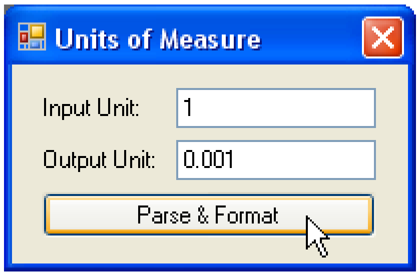
When automating Solid Edge, first convert user input to internal units. Calculations and geometric placements use the internal units. When displaying units, you must convert from internal units to default units. The UnitsOfMeasure object handles these conversions.

Working with Units of Measure

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 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.

The following programs uses both the ParseUnit and FormatUnit methods to duplicate the behavior of unit fields in Solid Edge. The code also checks whether the input is a valid unit key-in and outputs the 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.



5-2 - Units of Measure Example

Formatting and Displaying Units (Visual Basic.NET)

```
Imports SolidEdgeFramework
Imports System.Runtime.InteropServices

Public Class Form1
  Private m_application As SolidEdgeFramework.Application

  Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load

    Try
      ' Connect to a running instance of Solid Edge
      m_application = Marshal.GetActiveObject("SolidEdge.Application")
    Catch ex As Exception
      MessageBox.Show(ex.Message, "Error")
    End Try
  End Sub

  Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click
    Dim objDocument As SolidEdgeFramework.SolidEdgeDocument = Nothing
    Dim objUOM As SolidEdgeFramework.UnitsOfMeasure = Nothing
    Dim dHexSize As Double
    Try
      ' Get a reference to the active document
      objDocument = m_application.ActiveDocument

      ' Get a reference to the active document's unit of measure
      objUOM = objDocument.UnitsOfMeasure

      ' Attempt to parse the UOM input by user
      dHexSize = objUOM.ParseUnit( _
        UnitTypeConstants.igUnitDistance, TextBox1.Text)

      ' Update the 2nd textbox with the parsed UOM
      TextBox2.Text = dHexSize.ToString()
    Catch ex As Exception
      MessageBox.Show(ex.Message, "Invalid unit")
    Finally
      If Not (objUOM Is Nothing) Then
        Marshal.ReleaseComObject(objUOM)
        objUOM = Nothing
      End If
      If Not (objDocument Is Nothing) Then
        Marshal.ReleaseComObject(objDocument)
        objDocument = Nothing
      End If
    End Try
  End Sub

  Private Sub Form1_FormClosing(ByVal sender As System.Object, _
    ByVal e As System.Windows.Forms.FormClosingEventArgs) _
    Handles MyBase.FormClosing

    Try
      If Not (m_application Is Nothing) Then
        Marshal.ReleaseComObject(m_application)
        m_application = Nothing
      End If
    Catch ex As Exception
      MessageBox.Show(ex.Message, "Error")
    End Try
  End Sub
End Class
```

Formatting and Displaying Units (C#)

```
using SolidEdgeFramework;
using System;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace MyMacro
{
  public partial class Form1 : Form
  {
    private SolidEdgeFramework.Application m_application;

    public Form1()
    {
      InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
      try
      {
        // Connect to a running instance of Solid Edge
        m_application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");
      }
      catch (System.Exception ex)
      {
        MessageBox.Show(ex.Message, "Error");
      }
    }

    private void Button1_Click(object sender, EventArgs e)
    {
      SolidEdgeFramework.SolidEdgeDocument document = null;
      SolidEdgeFramework.UnitsOfMeasure uom = null;
      double dHexSize = 0;
      try
      {
        // Get a reference to the active document
        document = (SolidEdgeFramework.SolidEdgeDocument)
          m_application.ActiveDocument;

        // Get a reference to the active document's unit of measure
        uom = document.UnitsOfMeasure;

        // Attempt to parse the UOM input by user
        dHexSize = (double)uom.ParseUnit(
          (int)UnitTypeConstants.igUnitDistance, TextBox1.Text);

        // Update the 2nd textbox with the parsed UOM
        TextBox2.Text = dHexSize.ToString();
      }
      catch (System.Exception ex)
      {
        MessageBox.Show(ex.Message, "Invalid unit");
      }
      finally
      {
        if (uom != null)
        {
          Marshal.ReleaseComObject(uom);
          uom = null;
        }
        if (document != null)
        {
          Marshal.ReleaseComObject(document);
          document = null;
        }
      }
    }

    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
      try
      {
        if (m_application != null)
        {
          Marshal.ReleaseComObject(m_application);
          m_application = null;
        }
      }
      catch (System.Exception ex)
      {
        MessageBox.Show(ex.Message, "Error");
      }
    }
  }
}
```

Chapter 6 - Part and Sheet Metal Documents

Models Collection

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.

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.

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.

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.

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.

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 that 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 also 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.

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.

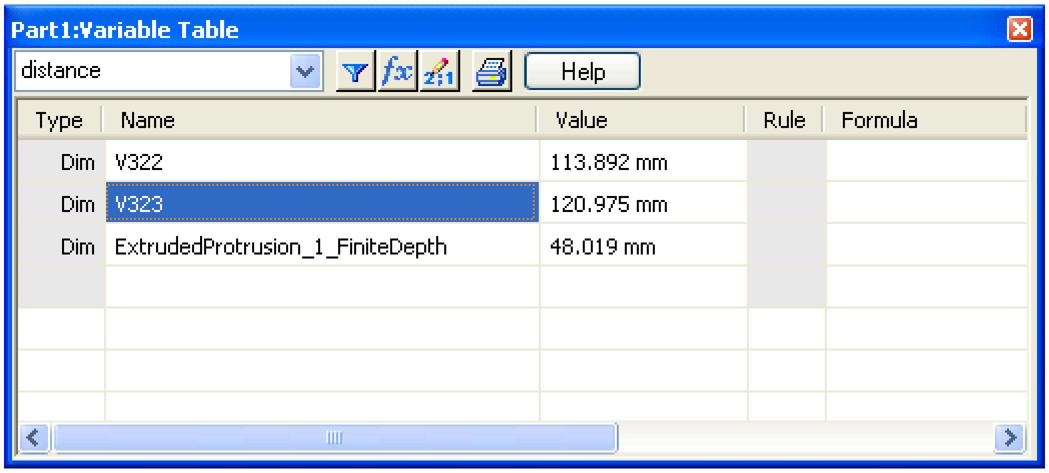
The Relations2d object provides the methods for placing relationships and for iterating through all the relationships that exist on the associated profile. To establish a relationship between elements, use an add method on the Relations2d collection on the profile.

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.

Variables

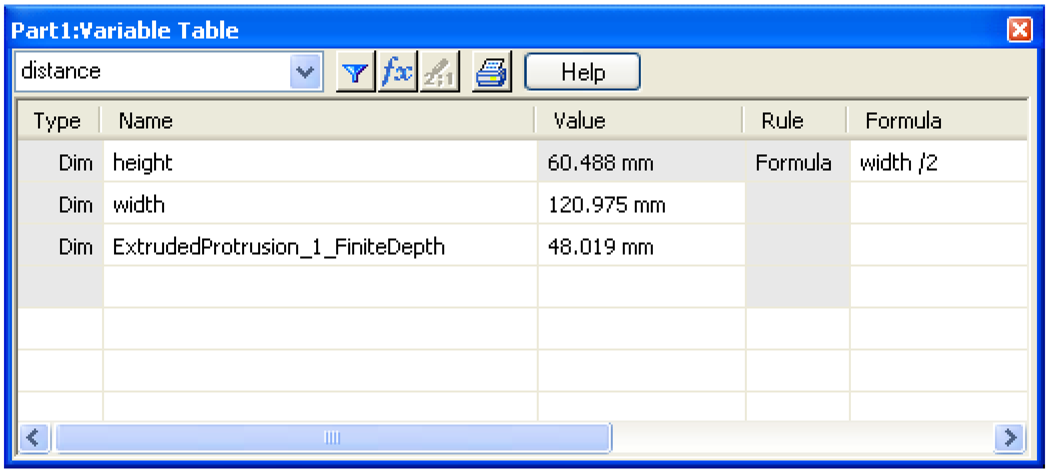
The variable system allows you to define relationships between variables and dimensions using equations, external functions, and spreadsheets. For example, you can construct a rectangular profile using four lines, and place two dimensions to control the height and width. The following illustration shows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



Type	Name	Value	Rule	Formula
Dim	V322	113.892 mm		
Dim	V323	120.975 mm		
Dim	ExtrudedProtrusion_1_FiniteDepth	48.019 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, V322 has been renamed to "height," and V323 has been renamed to "width."



Type	Name	Value	Rule	Formula
Dim	height	60.488 mm	Formula	width /2
Dim	width	120.975 mm		
Dim	ExtrudedProtrusion_1_FiniteDepth	48.019 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.

Part Modeling Examples

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.

Modeling a Part (Visual Basic .NET)

```
Imports SolidEdgeConstants
Imports System.Runtime.InteropServices

Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objDocuments As SolidEdgeFramework.Documents = Nothing
    Dim objPart As SolidEdgePart.PartDocument = Nothing
    Dim objProfileSets As SolidEdgePart.ProfileSets = Nothing
    Dim objProfileSet As SolidEdgePart.ProfileSet = Nothing
    Dim objProfiles As SolidEdgePart.Profiles = Nothing
    Dim objProfile As SolidEdgePart.Profile = Nothing
    Dim objRefplanes As SolidEdgePart.RefPlanes = Nothing
    Dim objRelations2d As SolidEdgeFrameworkSupport.Relations2d = Nothing
    Dim objRelation2d As SolidEdgeFrameworkSupport.Relation2d = Nothing
    Dim objLines2d As SolidEdgeFrameworkSupport.Lines2d = Nothing
    Dim objLine2d As SolidEdgeFrameworkSupport.Line2d = Nothing
    Dim objModels As SolidEdgePart.Models = Nothing
    Dim objModel As SolidEdgePart.Model = Nothing
    Dim aProfiles As Array

    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the documents collection
      objDocuments = objApplication.Documents

      ' Create a new part document
      objPart = objApplication.Add("SolidEdge.PartDocument")

      ' Get a reference to the profile sets collection
      objProfileSets = objPart.ProfileSets

      ' Add a new profile set
      objProfileSet = objProfileSets.Add()

      ' Get a reference to the profiles collection
      objProfiles = objProfileSet.Profiles

      ' Get a reference to the ref planes collection
      objRefplanes = objPart.RefPlanes

      ' Add a new profile
      objProfile = objProfiles.Add(objRefplanes.Item(3))
      ' Get a reference to the lines2d collection
      objLines2d = objProfile.Lines2d

      ' Draw the Base Profile
      objLine2d = objLines2d.AddBy2Points(0, 0, 0.08, 0)
      objLine2d = objLines2d.AddBy2Points(0.08, 0, 0.08, 0.06)
      objLine2d = objLines2d.AddBy2Points(0.08, 0.06, 0.064, 0.06)
      objLine2d = objLines2d.AddBy2Points(0.064, 0.06, 0.064, 0.02)
      objLine2d = objLines2d.AddBy2Points(0.064, 0.02, 0.048, 0.02)
      objLine2d = objLines2d.AddBy2Points(0.048, 0.02, 0.048, 0.06)
      objLine2d = objLines2d.AddBy2Points(0.048, 0.06, 0.032, 0.06)
      objLine2d = objLines2d.AddBy2Points(0.032, 0.06, 0.032, 0.02)
      objLine2d = objLines2d.AddBy2Points(0.032, 0.02, 0.016, 0.02)
      objLine2d = objLines2d.AddBy2Points(0.016, 0.02, 0.016, 0.06)
      objLine2d = objLines2d.AddBy2Points(0.016, 0.06, 0, 0.06)
      objLine2d = objLines2d.AddBy2Points(0, 0.06, 0, 0)

      ' Define Relations among the Line objects to make the Profile closed
      objRelations2d = objProfile.Relations2d
      objRelation2d = objRelations2d.AddKeypoint( _
        objLines2d.Item(1), _
        KeypointIndexConstants.igLineEnd, _
        objLines2d.Item(2), _
        KeypointIndexConstants.igLineStart)
      objRelation2d = objRelations2d.AddKeypoint( _
        objLines2d.Item(2), _
        KeypointIndexConstants.igLineEnd, _
        objLines2d.Item(3), _
        KeypointIndexConstants.igLineStart)
      objRelation2d = objRelations2d.AddKeypoint( _
        objLines2d.Item(3), _
        KeypointIndexConstants.igLineEnd, _
        objLines2d.Item(4), _
        KeypointIndexConstants.igLineStart)
      objRelation2d = objRelations2d.AddKeypoint( _
        objLines2d.Item(4), _
        KeypointIndexConstants.igLineEnd, _
        objLines2d.Item(5), _
        KeypointIndexConstants.igLineStart)
      objRelation2d = objRelations2d.AddKeypoint( _
        objLines2d.Item(5), _
        KeypointIndexConstants.igLineEnd, _
        objLines2d.Item(6), _
        KeypointIndexConstants.igLineStart)
      objRelation2d = objRelations2d.AddKeypoint( _
        objLines2d.Item(6), _
        KeypointIndexConstants.igLineEnd, _
        objLines2d.Item(7), _
        KeypointIndexConstants.igLineStart)
      objRelation2d = objRelations2d.AddKeypoint( _
        objLines2d.Item(7), _
        KeypointIndexConstants.igLineEnd, _
        objLines2d.Item(8), _
        KeypointIndexConstants.igLineStart)
      objRelation2d = objRelations2d.AddKeypoint( _
        objLines2d.Item(8), _
```

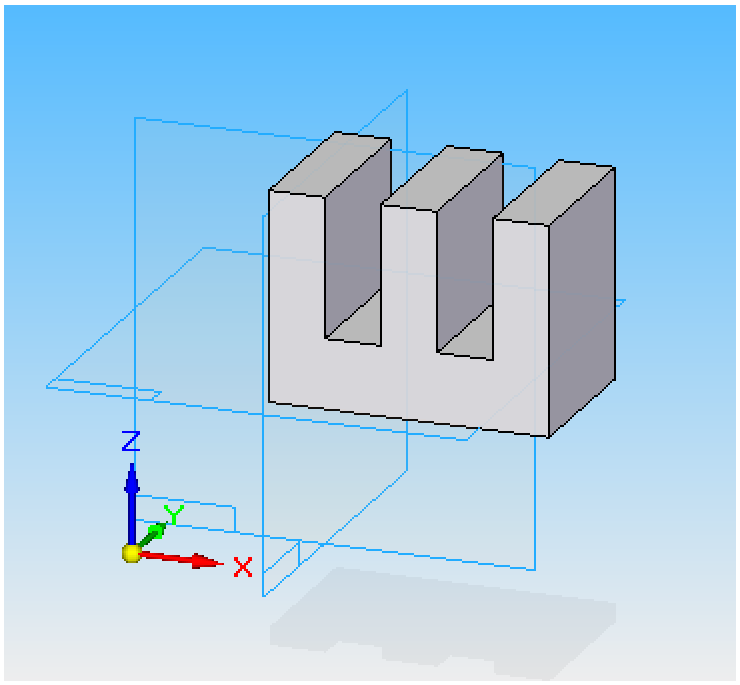
```
        KeypointIndexConstants.igLineEnd, _
        objLines2d.Item(9), _
        KeypointIndexConstants.igLineStart)
      objRelation2d = objRelations2d.AddKeypoint( _
        objLines2d.Item(9), _
        KeypointIndexConstants.igLineEnd, _
        objLines2d.Item(10), _
        KeypointIndexConstants.igLineStart)
      objRelation2d = objRelations2d.AddKeypoint( _
        objLines2d.Item(10), _
        KeypointIndexConstants.igLineEnd, _
        objLines2d.Item(11), _
        KeypointIndexConstants.igLineStart)
      objRelation2d = objRelations2d.AddKeypoint( _
        objLines2d.Item(11), _
        KeypointIndexConstants.igLineEnd, _
        objLines2d.Item(12), _
        KeypointIndexConstants.igLineStart)
      objRelation2d = objRelations2d.AddKeypoint( _
        objLines2d.Item(12), _
        KeypointIndexConstants.igLineEnd, _
        objLines2d.Item(1), _
        KeypointIndexConstants.igLineStart)

      ' Close the profile
      objProfile.End( _
        SolidEdgePart.ProfileValidationType.igProfileClosed)

      ' Hide the profile
      objProfile.Visible = False

      ' Create a new array of profile objects
      aProfiles = Array.CreateInstance(GetType(SolidEdgePart.Profile), 1)
      aProfiles.SetValue(objProfile, 0)

      ' Get a reference to the models collection
      objModels = objPart.Models

      ' Create the extended protrusion.
      objModel = objModels.AddFiniteExtrudedProtrusion( _
        aProfiles.Length, _
        aProfiles, _
        SolidEdgePart.FeaturePropertyConstants.igRight, _
        0.05)
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objModel Is Nothing) Then
        Marshal.ReleaseComObject(objModel)
        objModel = Nothing
      End If
      If Not (objModels Is Nothing) Then
        Marshal.ReleaseComObject(objModels)
        objModels = Nothing
      End If
      If Not (objRelations2d Is Nothing) Then
        Marshal.ReleaseComObject(objRelations2d)
        objRelations2d = Nothing
      End If
      If Not (objLine2d Is Nothing) Then
        Marshal.ReleaseComObject(objLine2d)
        objLine2d = Nothing
      End If
      If Not (objLines2d Is Nothing) Then
        Marshal.ReleaseComObject(objLines2d)
        objLines2d = Nothing
      End If
      If Not (objRelation2d Is Nothing) Then
        Marshal.ReleaseComObject(objRelation2d)
        objRelation2d = Nothing
      End If
      If Not (objRefplanes Is Nothing) Then
        Marshal.ReleaseComObject(objRefplanes)
        objRefplanes = Nothing
      End If
      If Not (objProfile Is Nothing) Then
        Marshal.ReleaseComObject(objProfile)
        objProfile = Nothing
      End If
      If Not (objProfiles Is Nothing) Then
        Marshal.ReleaseComObject(objProfiles)
        objProfiles = Nothing
      End If
      If Not (objProfileSet Is Nothing) Then
        Marshal.ReleaseComObject(objProfileSet)
        objProfileSet = Nothing
      End If
      If Not (objProfileSets Is Nothing) Then
        Marshal.ReleaseComObject(objProfileSets)
        objProfileSets = Nothing
      End If
      If Not (objPart Is Nothing) Then
        Marshal.ReleaseComObject(objPart)
        objPart = Nothing
      End If
      If Not (objDocuments Is Nothing) Then
        Marshal.ReleaseComObject(objDocuments)
        objDocuments = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Modeling a Part (C#)

```
using SolidEdgeConstants;
using System;
using System.Reflection;
using System.Runtime.InteropServices;
```

```
namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.Documents documents = null;
      SolidEdgePart.PartDocument part = null;
      SolidEdgePart.ProfileSets profileSets = null;
      SolidEdgePart.ProfileSet profileSet = null;
      SolidEdgePart.Profiles profiles = null;
      SolidEdgePart.Profile profile = null;
      SolidEdgePart.RefPlanes refplanes = null;
      SolidEdgeFrameworkSupport.Relations2d relations2d = null;
      SolidEdgeFrameworkSupport.Relation2d relation2d = null;
      SolidEdgeFrameworkSupport.Lines2d lines2d = null;
      SolidEdgeFrameworkSupport.Line2d line2d = null;
      SolidEdgePart.Models models = null;
      SolidEdgePart.Model model = null;
      System.Array aProfiles = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the documents collection
        documents = application.Documents;

        // Create a new part document
        part = (SolidEdgePart.PartDocument)
          documents.Add("SolidEdge.PartDocument", Missing.Value);

        // Get a reference to the profile sets collection
        profileSets = part.ProfileSets;

        // Add a new profile set
        profileSet = profileSets.Add();

        // Get a reference to the profiles collection
        profiles = profileSet.Profiles;

        // Get a reference to the ref planes collection
        refplanes = part.RefPlanes;

        // Add a new profile
        profile = profiles.Add(refplanes.Item(3));

        // Get a reference to the lines2d collection
        lines2d = profile.Lines2d;

        // Draw the Base Profile
        lines2d.AddBy2Points(0, 0, 0.08, 0);
        lines2d.AddBy2Points(0.08, 0, 0.08, 0.06);
        lines2d.AddBy2Points(0.08, 0.06, 0.064, 0.06);
        lines2d.AddBy2Points(0.064, 0.06, 0.064, 0.02);
        lines2d.AddBy2Points(0.064, 0.02, 0.048, 0.02);
        lines2d.AddBy2Points(0.048, 0.02, 0.048, 0.06);
        lines2d.AddBy2Points(0.048, 0.06, 0.032, 0.06);
        lines2d.AddBy2Points(0.032, 0.06, 0.032, 0.02);
        lines2d.AddBy2Points(0.032, 0.02, 0.016, 0.02);
        lines2d.AddBy2Points(0.016, 0.02, 0.016, 0.06);
        lines2d.AddBy2Points(0.016, 0.06, 0, 0.06);
        lines2d.AddBy2Points(0, 0.06, 0, 0);

        // Define Relations among the Line objects to make the Profile closed
        relations2d = (SolidEdgeFrameworkSupport.Relations2d)
          profile.Relations2d;
        relation2d = relations2d.AddKeypoint(
          lines2d.Item(1),
          (int)KeypointIndexConstants.igLineEnd,
          lines2d.Item(2),
          (int)KeypointIndexConstants.igLineStart,
          true);
        relation2d = relations2d.AddKeypoint(
          lines2d.Item(2),
          (int)KeypointIndexConstants.igLineEnd,
          lines2d.Item(3),
          (int)KeypointIndexConstants.igLineStart,
          true);
        relation2d = relations2d.AddKeypoint(
          lines2d.Item(3),
          (int)KeypointIndexConstants.igLineEnd,
          lines2d.Item(4),
          (int)KeypointIndexConstants.igLineStart,
          true);
        relation2d = relations2d.AddKeypoint(
          lines2d.Item(4),
          (int)KeypointIndexConstants.igLineEnd,
          lines2d.Item(5),
          (int)KeypointIndexConstants.igLineStart,
          true);
        relation2d = relations2d.AddKeypoint(
          lines2d.Item(5),
          (int)KeypointIndexConstants.igLineEnd,
          lines2d.Item(6),
          (int)KeypointIndexConstants.igLineStart,
          true);
        relation2d = relations2d.AddKeypoint(
          lines2d.Item(6),
          (int)KeypointIndexConstants.igLineEnd,
          lines2d.Item(7),
          (int)KeypointIndexConstants.igLineStart,
          true);
        relation2d = relations2d.AddKeypoint(
          lines2d.Item(7),
          (int)KeypointIndexConstants.igLineEnd,
          lines2d.Item(8),
          (int)KeypointIndexConstants.igLineStart,
          true);
        relation2d = relations2d.AddKeypoint(
          lines2d.Item(8),
          (int)KeypointIndexConstants.igLineEnd,
          lines2d.Item(9),
          (int)KeypointIndexConstants.igLineStart,
          true);
        relation2d = relations2d.AddKeypoint(
          lines2d.Item(10),
          (int)KeypointIndexConstants.igLineEnd,
          lines2d.Item(11),
          (int)KeypointIndexConstants.igLineStart,
          true);
        relation2d = relations2d.AddKeypoint(
          lines2d.Item(11),
          (int)KeypointIndexConstants.igLineEnd,
          lines2d.Item(12),
          (int)KeypointIndexConstants.igLineStart,
          true);
        relation2d = relations2d.AddKeypoint(
          lines2d.Item(12),
          (int)KeypointIndexConstants.igLineEnd,
          lines2d.Item(1),
          (int)KeypointIndexConstants.igLineStart,
          true);

        // Close the profile
        profile.End(
          SolidEdgePart.ProfileValidationType.igProfileClosed);

        // Hide the profile
        profile.Visible = false;

        // Create a new array of profile objects
        aProfiles = Array.CreateInstance(typeof(SolidEdgePart.Profile), 1);
        aProfiles.SetValue(profile, 0);

        // Get a reference to the models collection
        models = part.Models;

        // Create the extended protrusion.
        model = models.AddFiniteExtrudedProtrusion(
          aProfiles.Length,
          ref aProfiles,
          SolidEdgePart.FeaturePropertyConstants.igRight,
          0.05,
          Missing.Value,
          Missing.Value,
          Missing.Value,
          Missing.Value);
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (model != null)
        {
          Marshal.ReleaseComObject(model);
          model = null;
        }
        if (models != null)
        {
          Marshal.ReleaseComObject(models);
          models = null;
        }
        if (line2d != null)
        {
          Marshal.ReleaseComObject(line2d);
          line2d = null;
        }
        if (lines2d != null)
        {
          Marshal.ReleaseComObject(lines2d);
          lines2d = null;
        }
        if (relation2d != null)
        {
          Marshal.ReleaseComObject(relation2d);
          relation2d = null;
        }
        if (relations2d != null)
        {
          Marshal.ReleaseComObject(relations2d);
          relations2d = null;
        }
        if (refplanes != null)
        {
          Marshal.ReleaseComObject(refplanes);
          refplanes = null;
        }
        if (profile != null)
        {
          Marshal.ReleaseComObject(profile);
          profile = null;
        }
        if (profileSet != null)
        {
          Marshal.ReleaseComObject(profileSet);
          profileSet = null;
        }
        if (profileSets != null)
        {
          Marshal.ReleaseComObject(profileSets);
          profileSets = null;
        }
        if (part != null)
        {
          Marshal.ReleaseComObject(part);
          part = null;
        }
        if (documents != null)
        {
          Marshal.ReleaseComObject(documents);
          documents = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

The previous code examples produce the following model.



6-1 - Extended Protrusion

Variables Examples

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.

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.

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.

Variable Table Example(Visual Basic .NET)

```
Imports System.Runtime.InteropServices
Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objPart As SolidEdgePart.PartDocument = Nothing
    Dim objVariables As SolidEdgeFramework.Variables = Nothing
    Dim objVariable As SolidEdgeFramework.variable = Nothing
    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the active document
      objPart = objApplication.ActiveDocument

      ' Get a reference to the variables collection
      objVariables = objPart.Variables

      'Add a variable
      objVariable = objVariables.Add("NewVar", "1.5")

      'Change the formula of the variable to a function
      objVariable.Formula = Math.Sin(0.1).ToString()

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

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

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

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

      'Delete the variable
      objVariable.Delete()
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objVariable Is Nothing) Then
        Marshal.ReleaseComObject(objVariable)
        objVariable = Nothing
      End If
      If Not (objVariables Is Nothing) Then
        Marshal.ReleaseComObject(objVariables)
        objVariables = Nothing
      End If
      If Not (objPart Is Nothing) Then
        Marshal.ReleaseComObject(objPart)
        objPart = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Variable Table Example(C#)

```
using SolidEdgeFramework;
using System;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgePart.PartDocument part = null;
      SolidEdgeFramework.Variables variables = null;
      SolidEdgeFramework.variable variable = null;
      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the active document
        part = (SolidEdgePart.PartDocument)
          application.ActiveDocument;

        // Get a reference to the variables collection
        variables = (SolidEdgeFramework.Variables)part.Variables;

        // Add a variable
        variable = (SolidEdgeFramework.variable)
          variables.Add("NewVar", "1.5", UnitTypeConstants.igUnitDistance);

        // Change the formula of the variable to a function
        variable.Formula = Math.Sin(0.1).ToString();

        // Change the name of the variable
        variable.Name = "NewName";

        // Change the value of the variable. This will not change
        // the value of the variable
        variable.Value = 123;

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

        // Change the value of the variable. It works now
        variable.Value = 789;

        //Delete the variable
        variable.Delete();
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (variable != null)
        {
          Marshal.ReleaseComObject(variable);
          variable = null;
        }
        if (variables != null)
        {
          Marshal.ReleaseComObject(variables);
          variables = null;
        }
        if (part != null)
        {
          Marshal.ReleaseComObject(part);
          part = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

Working with Dimensions—Overview

Solid Edge allows you to place and edit dimensions on elements. 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 Profile environment, the dimensions control the geometry; if the dimension changes, the geometry updates.

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

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

In the 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.

Placing Dimensions Example (Visual Basic .NET)

```
Imports SolidEdgeConstants
Imports System.Runtime.InteropServices

Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objDocuments As SolidEdgeFramework.Documents = Nothing
    Dim objPart As SolidEdgePart.PartDocument = Nothing
    Dim objProfileSets As SolidEdgePart.ProfileSets = Nothing
    Dim objProfileSet As SolidEdgePart.ProfileSet = Nothing
    Dim objProfiles As SolidEdgePart.Profiles = Nothing
    Dim objProfile As SolidEdgePart.Profile = Nothing
    Dim objRefPlanes As SolidEdgePart.RefPlanes = Nothing
    Dim objRefPlane As SolidEdgePart.RefPlane = Nothing
    Dim objLines2d As SolidEdgeFrameworkSupport.Lines2d = Nothing
    Dim aLine2d(0 To 3) As SolidEdgeFrameworkSupport.Line2d
    Dim objRelations2d As SolidEdgeFrameworkSupport.Relations2d = Nothing
    Dim objDimensions As SolidEdgeFrameworkSupport.Dimensions = Nothing
    Dim objDimension As SolidEdgeFrameworkSupport.Dimension = Nothing
    Dim objDimStyle As SolidEdgeFrameworkSupport.DimStyle = Nothing
    Dim i As Integer

    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the documents collection
      objDocuments = objApplication.Documents
```

```
' Add a Part document
objPart = objDocuments.Add("SolidEdge.PartDocument")

' Get a reference to the ref planes collection
objRefPlanes = objPart.RefPlanes

' Get a reference to the first reference plane
objRefPlane = objRefPlanes.Item(1)

' Get a reference to the profile sets collection
objProfileSets = objPart.ProfileSets

' Add a new profile set
objProfileSet = objProfileSets.Add()

' Get a reference to the profiles collection
objProfiles = objProfileSet.Profiles

' Add a new profile on the selected reference plane
objProfile = objProfiles.Add(objRefPlane)

' Get a reference to the lines 2d collection
objLines2d = objProfile.Lines2d

' Get a reference to the relations 2d collection
objRelations2d = objProfile.Relations2d

' Get a reference to the dimensions collection
objDimensions = objProfile.Dimensions

' Draw the geometry.
aLine2d(0) = objLines2d.AddBy2Points(0, 0, 0.1, 0)
aLine2d(1) = objLines2d.AddBy2Points(0.1, 0, 0.1, 0.1)
aLine2d(2) = objLines2d.AddBy2Points(0.1, 0.1, 0, 0.05)
aLine2d(3) = objLines2d.AddBy2Points(0, 0.05, 0, 0)

' Add endpoint relationships between the lines
objRelations2d.AddKeypoint( _
  aLine2d(0), _
  KeypointIndexConstants.igLineEnd, _
  aLine2d(1), _
  KeypointIndexConstants.igLineStart)

objRelations2d.AddKeypoint( _
  aLine2d(1), _
  KeypointIndexConstants.igLineEnd, _
  aLine2d(2), _
  KeypointIndexConstants.igLineStart)

objRelations2d.AddKeypoint( _
  aLine2d(2), _
  KeypointIndexConstants.igLineEnd, _
  aLine2d(3), _
  KeypointIndexConstants.igLineStart)

objRelations2d.AddKeypoint( _
        aLine2d(3), _
        KeypointIndexConstants.igLineEnd, _
        aLine2d(0), _
        KeypointIndexConstants.igLineStart)

      ' Add dimensions, and change the dimension units to meters
      objDimension = objDimensions.AddLength(aLine2d(1))
      objDimension.Constraint = True
      objDimStyle = objDimension.Style
      objDimStyle.PrimaryUnits = _
        DimLinearUnitConstants.igDimStyleLinearMeters

      objDimension = objDimensions.AddLength(aLine2d(3))
      objDimension.Constraint = True
      objDimStyle = objDimension.Style
      objDimStyle.PrimaryUnits = _
        DimLinearUnitConstants.igDimStyleLinearMeters

      objDimension = objDimensions.AddDistanceBetweenObjects( _
        aLine2d(1), 0.1, 0.1, 0, False, _
        aLine2d(2), 0, 0.05, 0, False)
      objDimension.Constraint = True
      objDimStyle = objDimension.Style
      objDimStyle.PrimaryUnits = _
        DimLinearUnitConstants.igDimStyleLinearMeters

    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      For i = 0 To aLine2d.Length - 1
        If Not (aLine2d(i) Is Nothing) Then
          Marshal.ReleaseComObject(aLine2d(i))
          aLine2d(i) = Nothing
        End If
      Next
      If Not (objDimStyle Is Nothing) Then
        Marshal.ReleaseComObject(objDimStyle)
        objDimStyle = Nothing
      End If
      If Not (objDimension Is Nothing) Then
        Marshal.ReleaseComObject(objDimension)
        objDimension = Nothing
      End If
      If Not (objDimensions Is Nothing) Then
        Marshal.ReleaseComObject(objDimensions)
        objDimensions = Nothing
      End If
      If Not (objRelations2d Is Nothing) Then
        Marshal.ReleaseComObject(objRelations2d)
        objRelations2d = Nothing
      End If
      If Not (objLines2d Is Nothing) Then
        Marshal.ReleaseComObject(objLines2d)
        objLines2d = Nothing
      End If
      If Not (objRefPlane Is Nothing) Then
        Marshal.ReleaseComObject(objRefPlane)
        objRefPlane = Nothing
      End If
      If Not (objRefPlanes Is Nothing) Then
        Marshal.ReleaseComObject(objRefPlanes)
        objRefPlanes = Nothing
      End If
      If Not (objProfile Is Nothing) Then
        Marshal.ReleaseComObject(objProfile)
        objProfile = Nothing
      End If
      If Not (objProfiles Is Nothing) Then
        Marshal.ReleaseComObject(objProfiles)
        objProfiles = Nothing
      End If
      If Not (objProfileSet Is Nothing) Then
        Marshal.ReleaseComObject(objProfileSet)
        objProfileSet = Nothing
      End If
      If Not (objProfileSets Is Nothing) Then
        Marshal.ReleaseComObject(objProfileSets)
        objProfileSets = Nothing
      End If
      If Not (objPart Is Nothing) Then
        Marshal.ReleaseComObject(objPart)
        objPart = Nothing
      End If
      If Not (objDocuments Is Nothing) Then
        Marshal.ReleaseComObject(objDocuments)
        objDocuments = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Placing Dimensions Example (C#)

```
using SolidEdgeConstants;
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.Documents documents = null;
      SolidEdgePart.PartDocument part = null;
      SolidEdgePart.ProfileSets profileSets = null;
      SolidEdgePart.ProfileSet profileSet = null;
      SolidEdgePart.Profiles profiles = null;
      SolidEdgePart.Profile profile = null;
      SolidEdgePart.RefPlanes refPlanes = null;
      SolidEdgePart.RefPlane refPlane = null;
      SolidEdgeFrameworkSupport.Lines2d lines2d = null;
      SolidEdgeFrameworkSupport.Line2d[] aLine2d =
        new SolidEdgeFrameworkSupport.Line2d[4];
      SolidEdgeFrameworkSupport.Relations2d relations2d = null;
      SolidEdgeFrameworkSupport.Dimensions dimensions = null;
      SolidEdgeFrameworkSupport.Dimension dimension = null;
      SolidEdgeFrameworkSupport.DimStyle dimStyle = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the documents collection
        documents = application.Documents;

        // Add a Part document
        part = (SolidEdgePart.PartDocument)
          documents.Add("SolidEdge.PartDocument", Missing.Value);

        // Get a reference to the ref planes collection
        refPlanes = part.RefPlanes;

        // Get a reference to the first reference plane
        refPlane = refPlanes.Item(1);

        // Get a reference to the profile sets collection
        profileSets = part.ProfileSets;

        // Add a new profile set
        profileSet = profileSets.Add();

        // Get a reference to the profiles collection
        profiles = profileSet.Profiles;

        // Add a new profile on the selected reference plane
        profile = profiles.Add(refPlane);

        // Get a reference to the lines 2d collection
        lines2d = profile.Lines2d;

        // Get a reference to the relations 2d collection
        relations2d = (SolidEdgeFrameworkSupport.Relations2d)
          profile.Relations2d;

        // Get a reference to the dimensions collection
        dimensions = (SolidEdgeFrameworkSupport.Dimensions)
          profile.Dimensions;

        // Draw the geometry.
        aLine2d[0] = lines2d.AddBy2Points(0, 0, 0.1, 0);
        aLine2d[1] = lines2d.AddBy2Points(0.1, 0, 0.1, 0.1);
        aLine2d[2] = lines2d.AddBy2Points(0.1, 0.1, 0, 0.05);
        aLine2d[3] = lines2d.AddBy2Points(0, 0.05, 0, 0);

        // Add endpoint relationships between the lines
        relations2d.AddKeypoint(
          aLine2d[0],
          (int)KeypointIndexConstants.igLineEnd,
          aLine2d[1],
          (int)KeypointIndexConstants.igLineStart,
          Missing.Value);

        relations2d.AddKeypoint(
          aLine2d[1],
          (int)KeypointIndexConstants.igLineEnd,
          aLine2d[2],
          (int)KeypointIndexConstants.igLineStart,
          Missing.Value);

        relations2d.AddKeypoint(
          aLine2d[2],
          (int)KeypointIndexConstants.igLineEnd,
          aLine2d[3],
          (int)KeypointIndexConstants.igLineStart,
          Missing.Value);

        relations2d.AddKeypoint(
          aLine2d[3],
          (int)KeypointIndexConstants.igLineEnd,
          aLine2d[0],
          (int)KeypointIndexConstants.igLineStart,
          Missing.Value);

        // Add dimensions, and change the dimension units to meters
        dimension = dimensions.AddLength(aLine2d[1]);
        dimension.Constraint = true;
        dimStyle = dimension.Style;
        dimStyle.PrimaryUnits =
SolidEdgeFrameworkSupport.DimLinearUnitConstants.igDimStyleLinearMeters;

        dimension = dimensions.AddLength(aLine2d[3]);
        dimension.Constraint = true;
        dimStyle = dimension.Style;
        dimStyle.PrimaryUnits =
SolidEdgeFrameworkSupport.DimLinearUnitConstants.igDimStyleLinearMeters;

        dimension = dimensions.AddDistanceBetweenObjects(
          aLine2d[1], 0.1, 0.1, 0, false, aLine2d[2], 0, 0.05, 0, false);
        dimension.Constraint = true;
        dimStyle = dimension.Style;
        dimStyle.PrimaryUnits =
SolidEdgeFrameworkSupport.DimLinearUnitConstants.igDimStyleLinearMeters;
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        for (int i = 0; i < aLine2d.Length; i++)
        {
          if (aLine2d[i] != null)
          {
            Marshal.ReleaseComObject(aLine2d[i]);
            aLine2d[i] = null;
          }
        }
        if (dimStyle != null)
        {
          Marshal.ReleaseComObject(dimStyle);
          dimStyle = null;
        }
        if (dimension != null)
        {
          Marshal.ReleaseComObject(dimension);
          dimension = null;
        }
        if (dimensions != null)
        {
          Marshal.ReleaseComObject(dimensions);
          dimensions = null;
        }
        if (relations2d != null)
        {
          Marshal.ReleaseComObject(relations2d);
          relations2d = null;
        }
        if (lines2d != null)
        {
          Marshal.ReleaseComObject(lines2d);
          lines2d = null;
        }
        if (refPlane != null)
        {
          Marshal.ReleaseComObject(refPlane);
          refPlane = null;
        }
        if (refPlanes != null)
        {
          Marshal.ReleaseComObject(refPlanes);
          refPlanes = null;
        }
        if (profile != null)
        {
          Marshal.ReleaseComObject(profile);
          profile = null;
        }
        if (profiles != null)
        {
          Marshal.ReleaseComObject(profiles);
          profiles = null;
        }
        if (profileSet != null)
        {
          Marshal.ReleaseComObject(profileSet);
          profileSet = null;
        }
        if (profileSets != null)
        {
          Marshal.ReleaseComObject(profileSets);
          profileSets = null;
        }
        if (part != null)
        {
          Marshal.ReleaseComObject(part);
          part = null;
        }
        if (documents != null)
        {
          Marshal.ReleaseComObject(documents);
          documents = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

Dimensions via Variables

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 following programs assume that Solid Edge is running and in the Profile environment.

Accessing Dimensions through the Variable Table (Visual Basic .NET)

```
Imports System.Runtime.InteropServices
Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objPart As SolidEdgePart.PartDocument = Nothing
    Dim objProfileSets As SolidEdgePart.ProfileSets = Nothing
    Dim objProfileSet As SolidEdgePart.ProfileSet = Nothing
    Dim objProfiles As SolidEdgePart.Profiles = Nothing
    Dim objProfile As SolidEdgePart.Profile = Nothing
    Dim objVariables As SolidEdgeFramework.Variables = Nothing
    Dim objVariable As Object = Nothing
    Dim objLines2d As SolidEdgeFrameworkSupport.Lines2d = Nothing
    Dim objLine2d As SolidEdgeFrameworkSupport.Line2d = Nothing
    Dim objDimensions As SolidEdgeFrameworkSupport.Dimensions = Nothing
    Dim objDimension1 As SolidEdgeFrameworkSupport.Dimension = Nothing
    Dim objDimension2 As SolidEdgeFrameworkSupport.Dimension = Nothing
    Dim objVariableList As SolidEdgeFramework.VariableList = Nothing
    Dim sName As String
    Dim sFormula As String
    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the active document
      objPart = objApplication.ActiveDocument

      ' Get a reference to the active document
      objProfileSets = objPart.ProfileSets

      ' Get a reference to the profile set
      objProfileSet = objProfileSets.Item(objProfileSets.Count)

      ' Get a reference to the profiles collection
      objProfiles = objProfileSet.Profiles

      ' Get a reference to the profile
      objProfile = objProfiles.Item(1)

      ' Get a reference to the variables collection
      objVariables = objPart.Variables

      ' Get a reference to the lines2d collection
      objLines2d = objProfile.Lines2d

      ' Get a reference to the dimensions collection
      objDimensions = objProfile.Dimensions

      ' Create a new line
      objLine2d = objLines2d.AddBy2Points(0, 0, 0.1, 0.1)

      ' Place a dimension on the line to control its length
      objDimension1 = objDimensions.AddLength(objLine2d)

      ' Make the dimension a driving dimension
      objDimension1.Constraint = True

      ' Create a second line
      objLine2d = objLines2d.AddBy2Points(0, 0.1, 0.1, 0.2)

      ' Place a dimension on the line to control its length
      objDimension2 = objDimensions.AddLength(objLine2d)

      ' Make the dimension a driving dimension
      objDimension2.Constraint = True

      ' Assign a name to the dimension placed on the first line
      objVariables.PutName(objDimension1, "Dimension1")

      ' Retrieve the system name of the second dimension, and display
      ' it in the debug window

      sName = objVariables.GetName(objDimension2)
      Console.WriteLine(String.Format("Dimension = {0}", sName))

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

      ' Retrieve the formula from the dimension, and print it to the
      ' debug window to verify
      sFormula = objVariables.GetFormula(sName)
      Console.WriteLine(String.Format("Formula = {0}", sFormula))

      ' This demonstrates the ability to reference a dimension object
      ' by its name
      objDimension1 = objVariables.Translate("Dimension1")

      ' To verify a dimension object was returned, change its
      ' TrackDistance property to cause the dimension to change
      objDimension1.TrackDistance = objDimension1.TrackDistance * 2

      ' Use the Query method to list all all user-defined
      ' variables and user-named Dimension objects and
      ' display in the debug window
      objVariableList = objVariables.Query("*")
      For Each objVariable In objVariableList
        Console.WriteLine(objVariables.GetName(objVariable))
      Next
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objVariableList Is Nothing) Then
        Marshal.ReleaseComObject(objVariableList)
        objVariableList = Nothing
      End If
      If Not (objDimension2 Is Nothing) Then
        Marshal.ReleaseComObject(objDimension2)
        objDimension2 = Nothing
      End If
      If Not (objDimension1 Is Nothing) Then
        Marshal.ReleaseComObject(objDimension1)
        objDimension1 = Nothing
      End If
      If Not (objDimensions Is Nothing) Then
        Marshal.ReleaseComObject(objDimensions)
        objDimensions = Nothing
      End If
      If Not (objLine2d Is Nothing) Then
        Marshal.ReleaseComObject(objLine2d)
        objLine2d = Nothing
      End If
      If Not (objLines2d Is Nothing) Then
        Marshal.ReleaseComObject(objLines2d)
        objLines2d = Nothing
      End If
      If Not (objVariable Is Nothing) Then
```

```
        Marshal.ReleaseComObject(objVariable)
        objVariable = Nothing
      End If
      If Not (objVariables Is Nothing) Then
        Marshal.ReleaseComObject(objVariables)
        objVariables = Nothing
      End If
      If Not (objProfile Is Nothing) Then
        Marshal.ReleaseComObject(objProfile)
        objProfile = Nothing
      End If
      If Not (objProfiles Is Nothing) Then
        Marshal.ReleaseComObject(objProfiles)
        objProfiles = Nothing
      End If
      If Not (objProfileSet Is Nothing) Then
        Marshal.ReleaseComObject(objProfileSet)
        objProfileSet = Nothing
      End If
      If Not (objProfileSets Is Nothing) Then
        Marshal.ReleaseComObject(objProfileSets)
        objProfileSets = Nothing
      End If
      If Not (objPart Is Nothing) Then
        Marshal.ReleaseComObject(objPart)
        objPart = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Accessing Dimensions through the Variable Table (C#)

```
using SolidEdgeFramework;
using System;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgePart.PartDocument part = null;
      SolidEdgePart.ProfileSets profileSets = null;
      SolidEdgePart.ProfileSet profileSet = null;
      SolidEdgePart.Profiles profiles = null;
      SolidEdgePart.Profile profile = null;
      SolidEdgeFramework.Variables variables = null;
      Object oVariable = null;
      SolidEdgeFrameworkSupport.Lines2d lines2d = null;
      SolidEdgeFrameworkSupport.Line2d line2d = null;
      SolidEdgeFrameworkSupport.Dimensions dimensions = null;
      SolidEdgeFrameworkSupport.Dimension dimension1 = null;
      SolidEdgeFrameworkSupport.Dimension dimension2 = null;
      SolidEdgeFramework.VariableList variableList = null;
      String sName;
      String sFormula;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the active document
        part = (SolidEdgePart.PartDocument)
          application.ActiveDocument;


        // Get a reference to the profile sets collection
        profileSets = part.ProfileSets;

        // Get a reference to the profile set
        profileSet = profileSets.Item(profileSets.Count);

        // Get a reference to the profiles collection
        profiles = profileSet.Profiles;

        // Get a reference to the profile
        profile = profiles.Item(1);

        // Get a reference to the variables collection
        variables = (SolidEdgeFramework.Variables)part.Variables;

        // Get a reference to the lines2d collection
        lines2d = profile.Lines2d;

        // Get a reference to the dimensions collection
        dimensions = (SolidEdgeFrameworkSupport.Dimensions)
          profile.Dimensions;

        // Create a line
        line2d = lines2d.AddBy2Points(0, 0, 0.1, 0.1);

        // Place a dimension on the line to control its length
        dimension1 = dimensions.AddLength(line2d);

        // Make the dimension a driving dimension
        dimension1.Constraint = true;

        // Create a second line
        line2d = lines2d.AddBy2Points(0, 0.1, 0.1, 0.2);

        // Place a dimension on the line to control its length
        dimension2 = dimensions.AddLength(line2d);

        // Make the dimension a driving dimension
        dimension2.Constraint = true;
        // Assign a name to the dimension placed on the first line
        variables.PutName(dimension1, "Dimension1");

        // Retrieve the system name of the second dimension, and display
        // it in the debug window
        sName = variables.GetName(dimension2);
        Console.WriteLine(String.Format("Dimension = {0}", sName));

        // Edit the formula of the second dimension to be half
        // the value of the first dimension
        variables.Edit(sName, "Dimension1/2.0");

        // Retrieve the formula from the dimension, and print it to the
        // debug window to verify
        sFormula = variables.GetFormula(sName);
        Console.WriteLine(String.Format("Formula = {0}", sFormula));

        // This demonstrates the ability to reference a dimension object
        // by its name
        dimension1 = (SolidEdgeFrameworkSupport.Dimension)
          variables.Translate("Dimension1");

        // To verify a dimension object was returned, change its
        // TrackDistance property to cause the dimension to change
        dimension1.TrackDistance = dimension1.TrackDistance * 2;

        // Use the Query method to list all all user-defined
        // variables and user-named Dimension objects and
        // display in the debug window
        variableList = (SolidEdgeFramework.VariableList)
          variables.Query("*", null, null, false);

        for (int i = 1; i <= variableList.Count; i++)
        {
          oVariable = variableList.Item(i);
          Console.WriteLine(variables.GetName(oVariable));
        }
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (variableList != null)
        {
          Marshal.ReleaseComObject(variableList);
          variableList = null;
        }
        if (dimension2 != null)
        {
          Marshal.ReleaseComObject(dimension2);
          dimension2 = null;
        }
        if (dimension1 != null)
        {
          Marshal.ReleaseComObject(dimension1);
          dimension1 = null;
        }
        if (dimensions != null)
        {
          Marshal.ReleaseComObject(dimensions);
          dimensions = null;
        }
        if (line2d != null)
        {
          Marshal.ReleaseComObject(line2d);
          line2d = null;
        }
        if (lines2d != null)
        {
          Marshal.ReleaseComObject(lines2d);
          lines2d = null;
        }
        if (oVariable != null)
        {
          Marshal.ReleaseComObject(oVariable);
          oVariable = null;
        }
        if (variables != null)
        {
          Marshal.ReleaseComObject(variables);
          variables = null;
        }
        if (profile != null)
        {
          Marshal.ReleaseComObject(profile);
          profile = null;
        }
        if (profiles != null)
        {
          Marshal.ReleaseComObject(profiles);
          profiles = null;
        }
        if (profileSet != null)
        {
          Marshal.ReleaseComObject(profileSet);
          profileSet = null;
        }
        if (profileSets != null)
        {
          Marshal.ReleaseComObject(profileSets);
          profileSets = null;
        }
        if (part != null)
        {
          Marshal.ReleaseComObject(part);
          part = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
```

```
                }
            }
        }
    }
}
```

Chapter 7 - Assemblies Documents

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.

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.

The following programs connect 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.

Creating Reference Elements (Visual Basic .NET)

```
Imports SolidEdgePart
Imports System.Runtime.InteropServices
Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objDocuments As SolidEdgeFramework.Documents = Nothing
    Dim objAssembly As SolidEdgeAssembly.AssemblyDocument = Nothing
    Dim objAsmRefPlanes As SolidEdgeAssembly.AsmRefPlanes = Nothing
    Dim objAsmRefPlane As SolidEdgeAssembly.AsmRefPlane = Nothing
    Dim objPPlane As SolidEdgeAssembly.AsmRefPlane = Nothing
    Dim objPart As SolidEdgePart.PartDocument = Nothing
    Dim objRefPlanes As SolidEdgePart.RefPlanes = Nothing
    Dim objRefPlane As SolidEdgePart.RefPlane = Nothing

    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Access the Documents collection object
      objDocuments = objApplication.Documents

      ' Add an Assembly document
      objAssembly = objDocuments.Add("SolidEdge.AssemblyDocument")

      ' Access the AsmRefPlanes collection object
      objAsmRefPlanes = objAssembly.AsmRefPlanes

      ' Create a reference plane at an angle to a
      ' principal reference plane
      objPPlane = objAsmRefPlanes.Item(2)

      ' Add a new reference plane
      objAsmRefPlane = objAsmRefPlanes.AddAngularByAngle( _
        objPPlane, _
        (2 * Math.PI / 3), _
        objAsmRefPlanes.Item(1), _
        ReferenceElementConstants.igPivotEnd, _
        ReferenceElementConstants.igNormalSide, _
        True)

      ' Add a Part document
      objPart = objDocuments.Add("SolidEdge.PartDocument")

      ' Access the RefPlanes collection object
      objRefPlanes = objPart.RefPlanes

      ' Create a global reference plane parallel to the top reference plane
      objRefPlane = objRefPlanes.AddParallelByDistance( _
        objRefPlanes.Item(1), _
        0.1, _
        ReferenceElementConstants.igNormalSide, _
        False)

    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objRefPlane Is Nothing) Then
        Marshal.ReleaseComObject(objRefPlane)
        objRefPlane = Nothing
      End If
      If Not (objRefPlanes Is Nothing) Then
        Marshal.ReleaseComObject(objRefPlanes)
        objRefPlanes = Nothing
      End If
      If Not (objPart Is Nothing) Then
        Marshal.ReleaseComObject(objPart)
        objPart = Nothing
      End If
      If Not (objPPlane Is Nothing) Then
        Marshal.ReleaseComObject(objPPlane)
        objPPlane = Nothing
      End If
      If Not (objAsmRefPlane Is Nothing) Then
        Marshal.ReleaseComObject(objAsmRefPlane)
        objAsmRefPlane = Nothing
      End If
      If Not (objAsmRefPlanes Is Nothing) Then
        Marshal.ReleaseComObject(objAsmRefPlanes)
        objAsmRefPlanes = Nothing
      End If
      If Not (objAssembly Is Nothing) Then
        Marshal.ReleaseComObject(objAssembly)
        objAssembly = Nothing
      End If
      If Not (objDocuments Is Nothing) Then
        Marshal.ReleaseComObject(objDocuments)
        objDocuments = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Creating Reference Elements (C#)

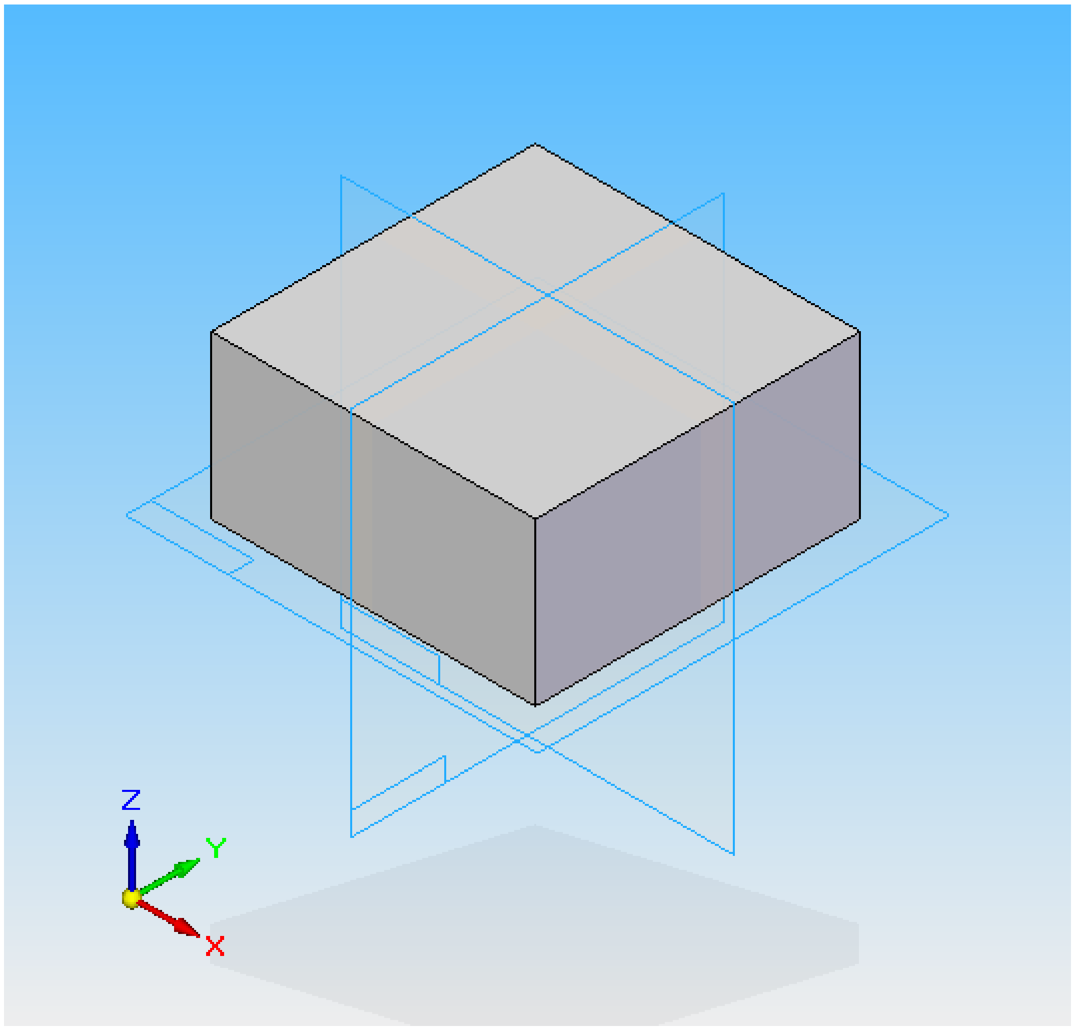
```
using SolidEdgeFramework;
using SolidEdgePart;
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.Documents documents = null;
      SolidEdgeAssembly.AssemblyDocument assembly = null;
      SolidEdgeAssembly.AsmRefPlanes asmRefPlanes = null;
      SolidEdgeAssembly.AsmRefPlane asmRefPlane = null;
      SolidEdgeAssembly.AsmRefPlane pPlane = null;
      SolidEdgePart.PartDocument part = null;
      SolidEdgePart.RefPlanes refPlanes = null;
      SolidEdgePart.RefPlane refPlane = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Access the Documents collection object
        documents = application.Documents;

        // Add an Assembly document
        assembly = (SolidEdgeAssembly.AssemblyDocument)
          documents.Add("SolidEdge.AssemblyDocument", Missing.Value);

        // Access the AsmRefPlanes collection object
        asmRefPlanes = assembly.AsmRefPlanes;

        // Create a reference plane at an angle to a principal
        // reference plane
        pPlane = asmRefPlanes.Item(2);

        // Add a new reference plane.Add a new reference plane
        asmRefPlane = asmRefPlanes.AddAngularByAngle(
          pPlane,
          (2 * Math.PI / 3),
          asmRefPlanes.Item(1),
          ReferenceElementConstants.igPivotEnd,
          ReferenceElementConstants.igNormalSide,
          true);
        // Add a Part document
        part = (SolidEdgePart.PartDocument)
          documents.Add("SolidEdge.PartDocument", Missing.Value);

        // Access the RefPlanes collection object
        refPlanes = part.RefPlanes;

        // Create a global reference plane parallel to the top
        // reference plane
        refPlane = refPlanes.AddParallelByDistance(
          refPlanes.Item(1),
          0.1,
          ReferenceElementConstants.igNormalSide,
          false,
          Missing.Value,
          Missing.Value,
          Missing.Value);
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (refPlane != null)
        {
          Marshal.ReleaseComObject(refPlane);
          refPlane = null;
        }
        if (refPlanes != null)
        {
          Marshal.ReleaseComObject(refPlanes);
          refPlanes = null;
        }
        if (part != null)
        {
          Marshal.ReleaseComObject(part);
          part = null;
        }
        if (pPlane != null)
        {
          Marshal.ReleaseComObject(pPlane);
          pPlane = null;
        }
        if (asmRefPlane != null)
        {
          Marshal.ReleaseComObject(asmRefPlane);
          asmRefPlane = null;
        }
        if (asmRefPlanes != null)
        {
          Marshal.ReleaseComObject(asmRefPlanes);
          asmRefPlanes = null;
        }
        if (assembly != null)
        {
          Marshal.ReleaseComObject(assembly);
          assembly = null;
        }
        if (documents != null)
        {
          Marshal.ReleaseComObject(documents);
          documents = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

Occurrences

The Occurrence object represents an instance of a part or subassembly within an assembly. It is an OLE link to a part or assembly document. An Occurrence object can only be added to an assembly through an Occurrences collection object.

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.

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.



7-1 - Occurrence Example

Adding a new Occurrence (Visual Basic .NET)

```
Imports SolidEdgeConstants
Imports System.Runtime.InteropServices

Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objDocuments As SolidEdgeFramework.Documents = Nothing
    Dim objAssembly As SolidEdgeAssembly.AssemblyDocument = Nothing
    Dim objOccurrences As SolidEdgeAssembly.Occurrences = Nothing
    Dim objOccurrence As SolidEdgeAssembly.Occurrence = Nothing

    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the documents collection
      objDocuments = objApplication.Documents
      ' Create a new assembly document
      objAssembly = objDocuments.Add("SolidEdge.AssemblyDocument")

      ' Get a reference to the occurrences collection
      objOccurrences = objAssembly.Occurrences

      ' Add a part document to the assembly
      objOccurrence = objOccurrences.AddByFilename("C:\Part1.par")
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objOccurrence Is Nothing) Then
        Marshal.ReleaseComObject(objOccurrence)
        objOccurrence = Nothing
      End If
      If Not (objOccurrences Is Nothing) Then
        Marshal.ReleaseComObject(objOccurrences)
        objOccurrences = Nothing
      End If
      If Not (objAssembly Is Nothing) Then
        Marshal.ReleaseComObject(objAssembly)
        objAssembly = Nothing
      End If
      If Not (objDocuments Is Nothing) Then
        Marshal.ReleaseComObject(objDocuments)
        objDocuments = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

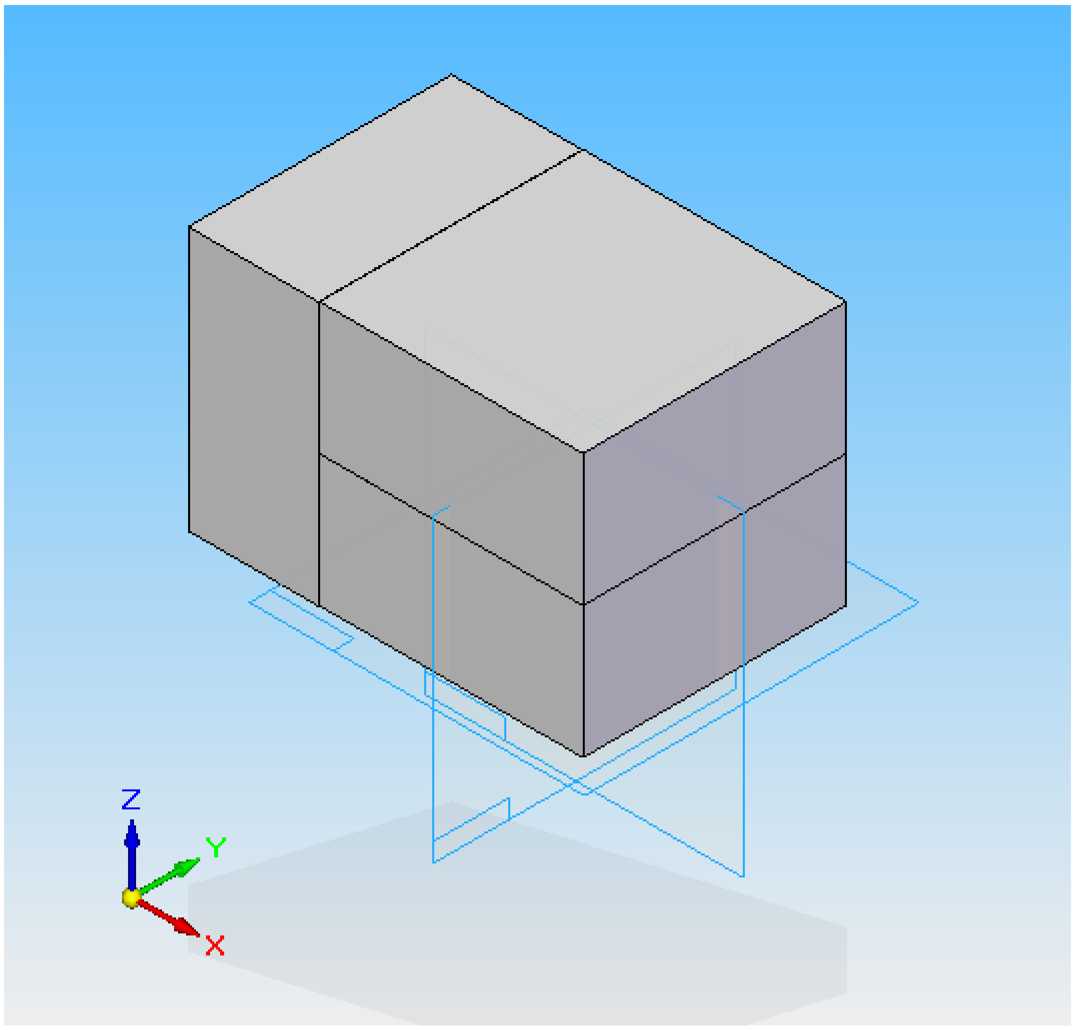
Adding a new Occurrence (C#)

```
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.Documents documents = null;
      SolidEdgeAssembly.AssemblyDocument assembly = null;
      SolidEdgeAssembly.Occurrences occurrences = null;
      SolidEdgeAssembly.Occurrence occurrence = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
```

```
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the documents collection
        documents = application.Documents;

        // Create a new assembly document
        assembly = (SolidEdgeAssembly.AssemblyDocument)
          documents.Add("SolidEdge.AssemblyDocument", Missing.Value);

        // Get a reference to the occurrences collection
        occurrences = assembly.Occurrences;

        // Add a part document to the assembly
        occurrence = occurrences.AddByFilename(
          @"C:\Part1.par", Missing.Value);
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (occurrence != null)
        {
          Marshal.ReleaseComObject(occurrence);
          occurrence = null;
        }
        if (occurrences != null)
        {
          Marshal.ReleaseComObject(occurrences);
          occurrences = null;
        }
        if (assembly != null)
        {
          Marshal.ReleaseComObject(assembly);
          assembly = null;
        }
        if (documents != null)
        {
          Marshal.ReleaseComObject(documents);
          documents = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

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. 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:



7-2 - Occurrences Example

Manipulating Occurrences (Visual Basic .NET)

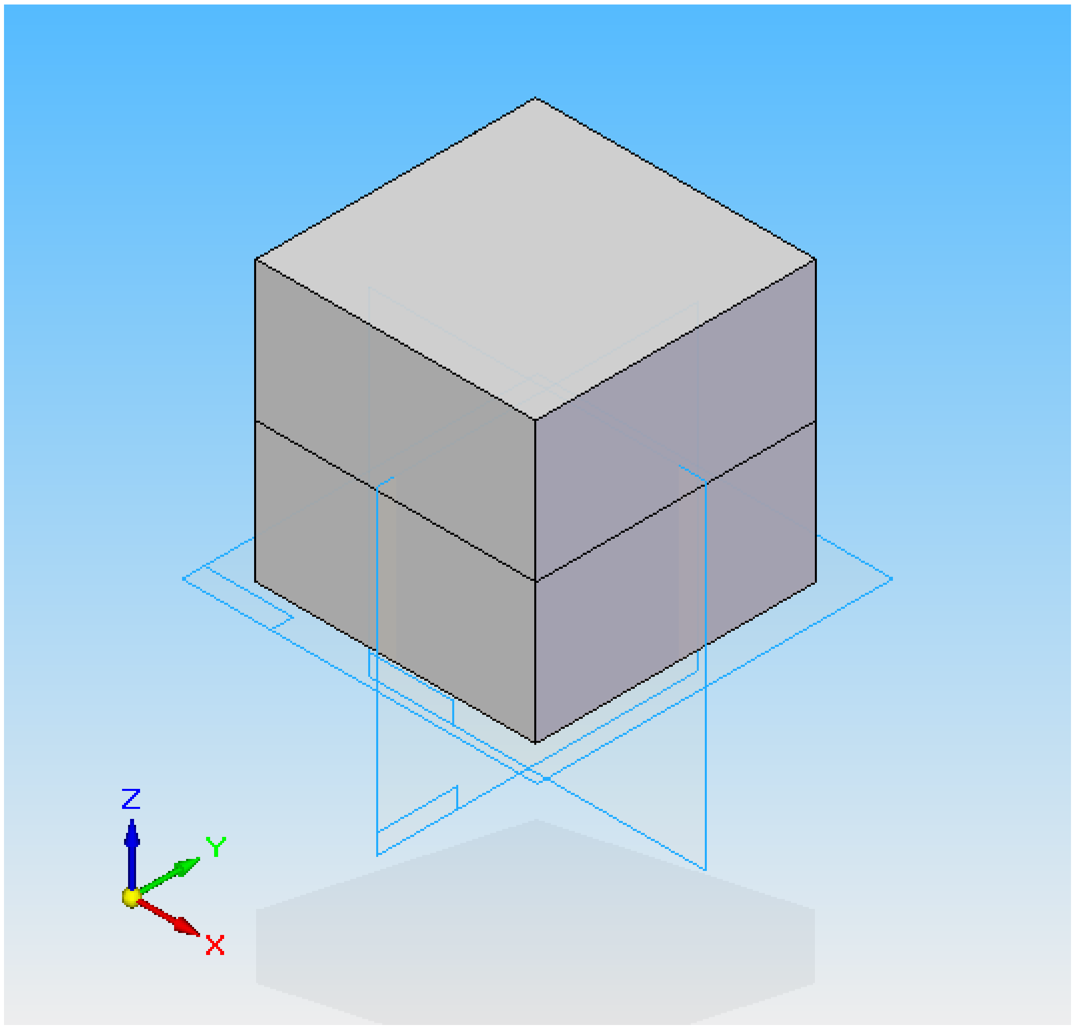
```
Imports SolidEdgeConstants
Imports System.Runtime.InteropServices

Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objDocuments As SolidEdgeFramework.Documents = Nothing
    Dim objAssembly As SolidEdgeAssembly.AssemblyDocument = Nothing
    Dim objOccurrences As SolidEdgeAssembly.Occurrences = Nothing
    Dim objOccurrence As SolidEdgeAssembly.Occurrence = Nothing
    Dim objOccurrence1 As SolidEdgeAssembly.Occurrence = Nothing
    Dim objOccurrence2 As SolidEdgeAssembly.Occurrence = Nothing
    Dim objOccurrence3 As SolidEdgeAssembly.Occurrence = Nothing

    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the documents collection
      objDocuments = objApplication.Documents

      ' Create a new assembly document
      objAssembly = objDocuments.Add("SolidEdge.AssemblyDocument")

      ' Get a reference to the occurrences collection
      objOccurrences = objAssembly.Occurrences

      ' Add the first block to the assembly
      objOccurrence1 = objOccurrences.AddByFilename("C:\Part1.par")

      ' Add the second block to the assembly
      objOccurrence2 = objOccurrences.AddByFilename("C:\Part1.par")

      ' It is currently in the same position and orientation as the first
      ' block, so reposition it
      objOccurrence2.Move(0, 0, 0.05)

      ' Add the third block to the assembly
      objOccurrence3 = objOccurrences.AddByFilename("C:\Part1.par")

      ' Rotate the third block to a vertical position.
      objOccurrence3.Rotate(0, 0, 0, 0, 1, 0, -Math.PI / 2)

      ' Reposition the third block.
      objOccurrence3.Move(-0.049, 0, 0.049)

    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objOccurrence3 Is Nothing) Then
        Marshal.ReleaseComObject(objOccurrence3)
        objOccurrence3 = Nothing
      End If
      If Not (objOccurrence2 Is Nothing) Then
        Marshal.ReleaseComObject(objOccurrence2)
        objOccurrence2 = Nothing
      End If
      If Not (objOccurrence1 Is Nothing) Then
        Marshal.ReleaseComObject(objOccurrence1)
        objOccurrence1 = Nothing
      End If
      If Not (objOccurrences Is Nothing) Then
        Marshal.ReleaseComObject(objOccurrences)
        objOccurrences = Nothing
      End If
      If Not (objAssembly Is Nothing) Then
        Marshal.ReleaseComObject(objAssembly)
        objAssembly = Nothing
      End If
      If Not (objDocuments Is Nothing) Then
        Marshal.ReleaseComObject(objDocuments)
        objDocuments = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Manipulating Occurrences (C#)

```
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.Documents documents = null;
      SolidEdgeAssembly.AssemblyDocument assembly = null;
      SolidEdgeAssembly.Occurrences occurrences = null;
      SolidEdgeAssembly.Occurrence occurrence1 = null;
      SolidEdgeAssembly.Occurrence occurrence2 = null;
      SolidEdgeAssembly.Occurrence occurrence3 = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the documents collection
        documents = application.Documents;

        // Create a new assembly document
        assembly = (SolidEdgeAssembly.AssemblyDocument)
          documents.Add("SolidEdge.AssemblyDocument", Missing.Value);

        // Get a reference to the occurrences collection
        occurrences = assembly.Occurrences;

        // Add the first block to the assembly
        occurrence1 = occurrences.AddByFilename(
          @"C:\Part1.par", Missing.Value);

        // Add the second block to the assembly
        occurrence2 = occurrences.AddByFilename(
          @"C:\Part1.par", Missing.Value);

        // It is currently in the same position and orientation as the first
        // block, so reposition it
        occurrence2.Move(0, 0, 0.05);

        // Add the third block to the assembly
        occurrence3 = occurrences.AddByFilename(
          @"C:\Part1.par", Missing.Value);

        // Rotate the third block to a vertical position.
        occurrence3.Rotate(0, 0, 0, 0, 1, 0, -Math.PI / 2);

        // Reposition the third block.
        occurrence3.Move(-0.049, 0, 0.049);
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (occurrence3 != null)
        {
          Marshal.ReleaseComObject(occurrence3);
          occurrence3 = null;
        }
        if (occurrence2 != null)
        {
          Marshal.ReleaseComObject(occurrence2);
          occurrence2 = null;
        }
        if (occurrence1 != null)
        {
          Marshal.ReleaseComObject(occurrence1);
          occurrence1 = null;
        }
        if (occurrences != null)
        {
          Marshal.ReleaseComObject(occurrences);
          occurrences = null;
        }
        if (assembly != null)
        {
          Marshal.ReleaseComObject(assembly);
          assembly = null;
        }
        if (documents != null)
        {
          Marshal.ReleaseComObject(documents);
          documents = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
```

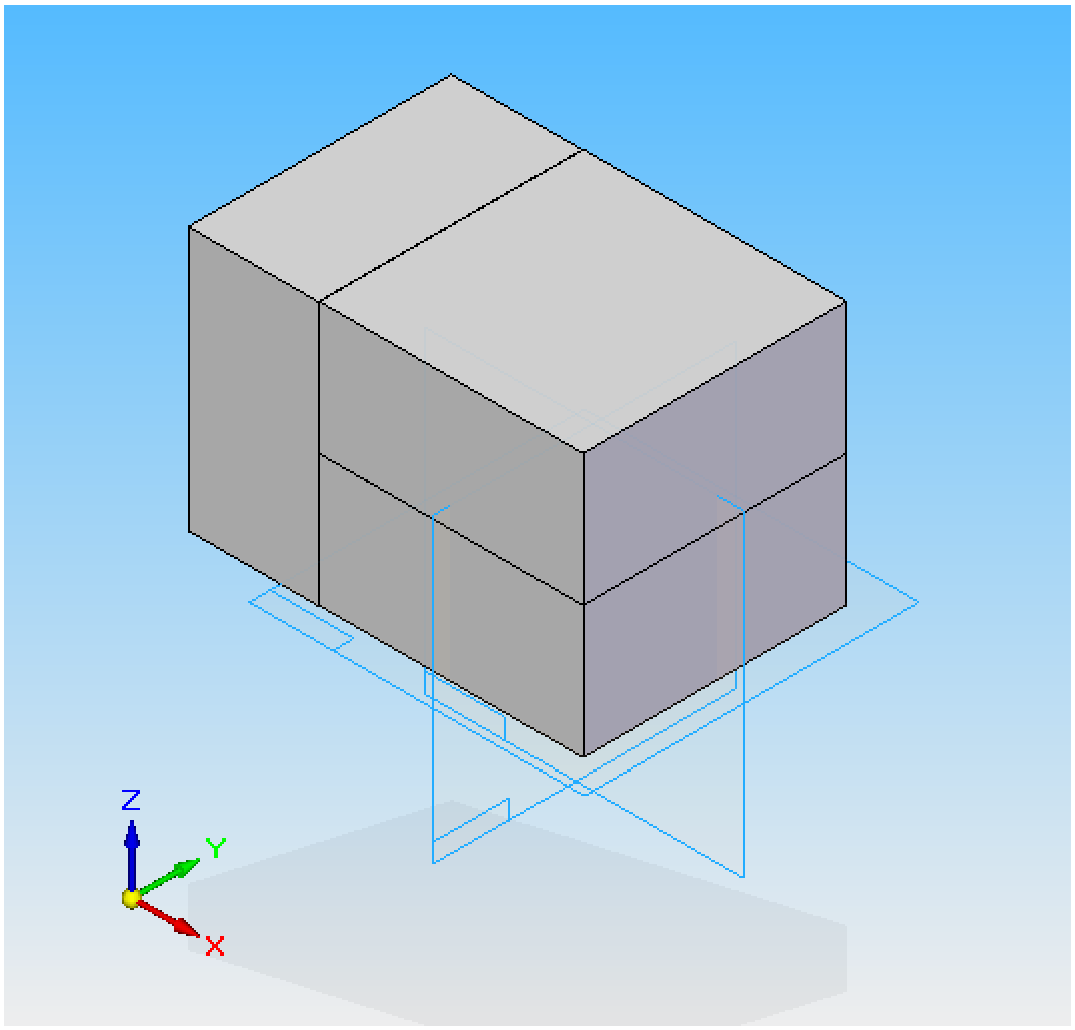
```
                }
            }
        }
    }
}
```

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



7-3 - Occurrences Example

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



7-4 - Occurrences Example

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.

References

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-placeactivate 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.

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.

Analyzing Existing Assembly Relationships (Visual Basic .NET)

```
Imports System.Runtime.InteropServices

Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objAssembly As SolidEdgeAssembly.AssemblyDocument = Nothing
    Dim objRelations3d As SolidEdgeAssembly.Relations3d = Nothing
    Dim objRelation3d As Object = Nothing
    Dim objAngularRelation3d As SolidEdgeAssembly.AngularRelation3d = Nothing
    Dim objAxialRelation3d As SolidEdgeAssembly.AxialRelation3d = Nothing
    Dim objGroundRelation3d As SolidEdgeAssembly.GroundRelation3d = Nothing
    Dim objPointRelation3d As SolidEdgeAssembly.PointRelation3d = Nothing
    Dim objPlanarRelation3d As SolidEdgeAssembly.PlanarRelation3d = Nothing

    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the active document
      objAssembly = objApplication.ActiveDocument

      ' Get a reference to the relations 3d collection
      objRelations3d = objAssembly.Relations3d

      ' Loop through the relations 3d objects
      For Each objRelation3d In objRelations3d
        ' Determine the relation type
        Select Case objRelation3d.Type
          Case SolidEdgeFramework.ObjectType.igAngularRelation3d
            objAngularRelation3d = objRelation3d
          Case SolidEdgeFramework.ObjectType.igAxialRelation3d
            objAxialRelation3d = objRelation3d
          Case SolidEdgeFramework.ObjectType.igGroundRelation3d
            objGroundRelation3d = objRelation3d
          Case SolidEdgeFramework.ObjectType.igPointRelation3d
            objPointRelation3d = objRelation3d
          Case SolidEdgeFramework.ObjectType.igPlanarRelation3d
            objPlanarRelation3d = objRelation3d
        End Select
      Next
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objPlanarRelation3d Is Nothing) Then
        Marshal.ReleaseComObject(objPlanarRelation3d)
        objPlanarRelation3d = Nothing
      End If
      If Not (objPointRelation3d Is Nothing) Then
        Marshal.ReleaseComObject(objPointRelation3d)
        objPointRelation3d = Nothing
      End If
      If Not (objGroundRelation3d Is Nothing) Then
```

```
        Marshal.ReleaseComObject(objGroundRelation3d)
        objGroundRelation3d = Nothing
      End If
      If Not (objAxialRelation3d Is Nothing) Then
        Marshal.ReleaseComObject(objAxialRelation3d)
        objAxialRelation3d = Nothing
      End If
      If Not (objAngularRelation3d Is Nothing) Then
        Marshal.ReleaseComObject(objAngularRelation3d)
        objAngularRelation3d = Nothing
      End If
      If Not (objRelation3d Is Nothing) Then
        Marshal.ReleaseComObject(objRelation3d)
        objRelation3d = Nothing
      End If
      If Not (objRelations3d Is Nothing) Then
        Marshal.ReleaseComObject(objRelations3d)
        objRelations3d = Nothing
      End If
      If Not (objAssembly Is Nothing) Then
        Marshal.ReleaseComObject(objAssembly)
        objAssembly = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Analyzing Existing Assembly Relationships (C#)

```
using SolidEdgeFramework;
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeAssembly.AssemblyDocument assembly = null;
      SolidEdgeAssembly.Relations3d relations3d = null;
      object relation3d = null;
      SolidEdgeAssembly.AngularRelation3d angularRelation3d = null;
      SolidEdgeAssembly.AxialRelation3d axialRelation3d = null;
      SolidEdgeAssembly.GroundRelation3d groundRelation3d = null;
      SolidEdgeAssembly.PointRelation3d pointRelation3d = null;
      SolidEdgeAssembly.PlanarRelation3d planarRelation3d = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the active document
        assembly = (SolidEdgeAssembly.AssemblyDocument)
          application.ActiveDocument;

        // Get a reference to the relations 3d collection
        relations3d = assembly.Relations3d;

        // Loop through the relations 3d objects
        for (int i = 1; i <= relations3d.Count; i++)
        {
          relation3d = relations3d.Item(i);
          Type type = relation3d.GetType();

          // Get the value of the Type proprety via Reflection
          SolidEdgeFramework.ObjectType objectType =
            (SolidEdgeFramework.ObjectType)type.InvokeMember(
              "Type",
              BindingFlags.GetProperty,
              null,
              relation3d,
              null);

          // Determine the relation type
          switch (objectType)
          {
            case SolidEdgeFramework.ObjectType.igAngularRelation3d:
              angularRelation3d = (SolidEdgeAssembly.AngularRelation3d)
                relation3d;
              break;
            case SolidEdgeFramework.ObjectType.igAxialRelation3d:
              axialRelation3d = (SolidEdgeAssembly.AxialRelation3d)
                relation3d;
              break;
            case SolidEdgeFramework.ObjectType.igGroundRelation3d:
              groundRelation3d = (SolidEdgeAssembly.GroundRelation3d)
                relation3d;
              break;
            case SolidEdgeFramework.ObjectType.igPointRelation3d:
              pointRelation3d = (SolidEdgeAssembly.PointRelation3d)
                relation3d;
              break;
            case SolidEdgeFramework.ObjectType.igPlanarRelation3d:
              planarRelation3d = (SolidEdgeAssembly.PlanarRelation3d)
                relation3d;
              break;
          }
        }
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (planarRelation3d != null)
        {
          Marshal.ReleaseComObject(planarRelation3d);
          planarRelation3d = null;
        }
        if (pointRelation3d != null)
        {
          Marshal.ReleaseComObject(pointRelation3d);
          pointRelation3d = null;
        }
        if (groundRelation3d != null)
        {
          Marshal.ReleaseComObject(groundRelation3d);
          groundRelation3d = null;
        }
        if (axialRelation3d != null)
        {
          Marshal.ReleaseComObject(axialRelation3d);
          axialRelation3d = null;
        }
        if (angularRelation3d != null)
        {
          Marshal.ReleaseComObject(angularRelation3d);
          angularRelation3d = null;
        }
        if (relation3d != null)
        {
          Marshal.ReleaseComObject(relation3d);
          relation3d = null;
        }
        if (relations3d != null)
        {
          Marshal.ReleaseComObject(relations3d);
          relations3d = null;
        }
        if (assembly != null)
        {
          Marshal.ReleaseComObject(assembly);
          assembly = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

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.

Adding New Assembly Relationships (Visual Basic .NET)

```
Imports SolidEdgeGeometry
Imports System.Runtime.InteropServices

Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objDocuments As SolidEdgeFramework.Documents = Nothing
    Dim objAssembly As SolidEdgeAssembly.AssemblyDocument = Nothing
    Dim objOccurrences As SolidEdgeAssembly.Occurrences = Nothing
    Dim objOccurrence1 As SolidEdgeAssembly.Occurrence = Nothing
    Dim objOccurrence2 As SolidEdgeAssembly.Occurrence = Nothing
    Dim objPart As SolidEdgePart.PartDocument = Nothing
    Dim objModels As SolidEdgePart.Models = Nothing
    Dim objModel As SolidEdgePart.Model = Nothing
    Dim objRevolvedProtrusions As SolidEdgePart.RevolvedProtrusions = Nothing
    Dim objRevolvedProtrusion As SolidEdgePart.RevolvedProtrusion = Nothing
    Dim objRevolvedCutouts As SolidEdgePart.RevolvedCutouts = Nothing
    Dim objRevolvedCutout As SolidEdgePart.RevolvedCutout = Nothing
    Dim objFaces As SolidEdgeGeometry.Faces = Nothing
    Dim objFace As SolidEdgeGeometry.Face = Nothing
    Dim objGeometry As Object = Nothing
    Dim objScrewConicalFace As SolidEdgeGeometry.Face = Nothing
    Dim objNutConicalFace As SolidEdgeGeometry.Face = Nothing
    Dim objRefToCylinderInScrew As SolidEdgeFramework.Reference = Nothing
    Dim objRefToConeInNut As SolidEdgeFramework.Reference = Nothing
    Dim objRelations3d As SolidEdgeAssembly.Relations3d = Nothing
    Dim objGroundRel As SolidEdgeAssembly.GroundRelation3d = Nothing
    Dim objRelNuttoScrew As SolidEdgeAssembly.AxialRelation3d = Nothing

    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the documents collection
      objDocuments = objApplication.Documents

      ' Create a new assembly document
      objAssembly = objDocuments.Add("SolidEdge.AssemblyDocument")

      ' Get a reference to the occurrences collection
      objOccurrences = objAssembly.Occurrences

      ' Add the first occurrence
      objOccurrence1 = objOccurrences.AddByFilename("C:\Screw.par")

      ' Get a reference to the occurrence document
      objPart = objOccurrence1.OccurrenceDocument

      ' Get a reference to the models collection
      objModels = objPart.Models

      ' Get a reference to the one and only model
```

```
objModel = objModels.Item(1)

' Get a reference to the revolved protrusions collection
objRevolvedProtrusions = objModel.RevolvedProtrusions

' Get a reference to the first revolved protrusion
objRevolvedProtrusion = objRevolvedProtrusions.Item(1)

' Get a reference to the side faces collection
objFaces = objRevolvedProtrusion.SideFaces

' Loop through the faces
For Each objFace In objFaces
  ' Get a reference to the geometry object
  objGeometry = objFace.Geometry
  If objGeometry.Type = GNTTypePropertyConstants.igCylinder Then
    objScrewConicalFace = objFace
    Exit For
  End If
Next

' Create the first reference
objRefToCylinderInScrew = objAssembly.CreateReference( _
  objOccurrence1, objScrewConicalFace)

' Add the second occurrence
objOccurrence2 = objOccurrences.AddByFilename("C:\Nut.par")

' Get a reference to the occurrence document
objPart = objOccurrence2.OccurrenceDocument

' Get a reference to the models collection
objModels = objPart.Models

' Get a reference to the one and only model
objModel = objModels.Item(1)

' Get a reference to the revolved cutouts collection
objRevolvedCutouts = objModel.RevolvedCutouts

' Get a reference to the first revolved cutout
objRevolvedCutout = objRevolvedCutouts.Item(1)

' Get a reference to the side faces collection
objFaces = objRevolvedCutout.SideFaces

' Loop through the faces
For Each objFace In objFaces
  objGeometry = objFace.Geometry
  If objGeometry.Type = GNTTypePropertyConstants.igCone Then
    objNutConicalFace = objFace
    Exit For
  End If
Next

' Create the second reference
objRefToConeInNut = objAssembly.CreateReference( _
      objOccurrence2, 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.
    objRelations3d = objAssembly.Relations3d
    objGroundRel = objRelations3d.Item(2)
    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.
    objRelNuttoScrew = objRelations3d.AddAxial( _
      objRefToConeInNut, objRefToCylinderInScrew, False)
  Catch ex As Exception
    Console.WriteLine(ex.Message)
  Finally
    If Not (objRelNuttoScrew Is Nothing) Then
      Marshal.ReleaseComObject(objRelNuttoScrew)
      objRelNuttoScrew = Nothing
    End If
    If Not (objGroundRel Is Nothing) Then
      Marshal.ReleaseComObject(objGroundRel)
      objGroundRel = Nothing
    End If
    If Not (objRelations3d Is Nothing) Then
      Marshal.ReleaseComObject(objRelations3d)
      objRelations3d = Nothing
    End If
    If Not (objRefToConeInNut Is Nothing) Then
      Marshal.ReleaseComObject(objRefToConeInNut)
      objRefToConeInNut = Nothing
    End If
    If Not (objRefToCylinderInScrew Is Nothing) Then
      Marshal.ReleaseComObject(objRefToCylinderInScrew)
      objRefToCylinderInScrew = Nothing
    End If
    If Not (objNutConicalFace Is Nothing) Then
      Marshal.ReleaseComObject(objNutConicalFace)
      objNutConicalFace = Nothing
    End If
    If Not (objScrewConicalFace Is Nothing) Then
      Marshal.ReleaseComObject(objScrewConicalFace)
      objScrewConicalFace = Nothing
    End If
    If Not (objGeometry Is Nothing) Then
      Marshal.ReleaseComObject(objGeometry)
      objGeometry = Nothing
    End If
    If Not (objFace Is Nothing) Then
      Marshal.ReleaseComObject(objFace)
      objFace = Nothing
    End If
    If Not (objFaces Is Nothing) Then
      Marshal.ReleaseComObject(objFaces)
      objFaces = Nothing
    End If
```

```
      If Not (objRevolvedCutout Is Nothing) Then
        Marshal.ReleaseComObject(objRevolvedCutout)
        objRevolvedCutout = Nothing
      End If
      If Not (objRevolvedCutouts Is Nothing) Then
        Marshal.ReleaseComObject(objRevolvedCutouts)
        objRevolvedCutouts = Nothing
      End If
      If Not (objRevolvedProtrusion Is Nothing) Then
        Marshal.ReleaseComObject(objRevolvedProtrusion)
        objRevolvedProtrusion = Nothing
      End If
      If Not (objRevolvedProtrusions Is Nothing) Then
        Marshal.ReleaseComObject(objRevolvedProtrusions)
        objRevolvedProtrusions = Nothing
      End If
      If Not (objModel Is Nothing) Then
        Marshal.ReleaseComObject(objModel)
        objModel = Nothing
      End If
      If Not (objModels Is Nothing) Then
        Marshal.ReleaseComObject(objModels)
        objModels = Nothing
      End If
      If Not (objPart Is Nothing) Then
        Marshal.ReleaseComObject(objPart)
        objPart = Nothing
      End If
      If Not (objOccurrence2 Is Nothing) Then
        Marshal.ReleaseComObject(objOccurrence2)
        objOccurrence2 = Nothing
      End If
      If Not (objOccurrence1 Is Nothing) Then
        Marshal.ReleaseComObject(objOccurrence1)
        objOccurrence1 = Nothing
      End If
      If Not (objOccurrences Is Nothing) Then
        Marshal.ReleaseComObject(objOccurrences)
        objOccurrences = Nothing
      End If
      If Not (objAssembly Is Nothing) Then
        Marshal.ReleaseComObject(objAssembly)
        objAssembly = Nothing
      End If
      If Not (objDocuments Is Nothing) Then
        Marshal.ReleaseComObject(objDocuments)
        objDocuments = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Adding New Assembly Relationships (C#)

```
using SolidEdgeFramework;
using SolidEdgeGeometry;
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.Documents documents = null;
      SolidEdgeAssembly.AssemblyDocument assembly = null;
      SolidEdgeAssembly.Occurrences occurrences = null;
      SolidEdgeAssembly.Occurrence occurrence1 = null;
      SolidEdgeAssembly.Occurrence occurrence2 = null;
      SolidEdgePart.PartDocument part = null;
      SolidEdgePart.Models models = null;
      SolidEdgePart.Model model = null;
      SolidEdgePart.RevolvedProtrusions revolvedProtrusions = null;
      SolidEdgePart.RevolvedProtrusion revolvedProtrusion = null;
      SolidEdgePart.RevolvedCutouts revolvedCutouts = null;
      SolidEdgePart.RevolvedCutout revolvedCutout = null;
      SolidEdgeGeometry.Faces faces = null;
      SolidEdgeGeometry.Face face = null;
      object geometry = null;
      SolidEdgeGeometry.Face screwConicalFace = null;
      SolidEdgeGeometry.Face nutConicalFace = null;
      SolidEdgeFramework.Reference refToCylinderInScrew = null;
      SolidEdgeFramework.Reference refToConeInNut = null;
      SolidEdgeAssembly.Relations3d relations3d = null;
      SolidEdgeAssembly.GroundRelation3d groundRelation3d = null;
      SolidEdgeAssembly.AxialRelation3d relNuttoScrew = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the documents collection
        documents = application.Documents;

        // Get a reference to the active document
        assembly = (SolidEdgeAssembly.AssemblyDocument)
          documents.Add("SolidEdge.AssemblyDocument", Missing.Value);

        // Get a reference to the occurrences collection
        occurrences = assembly.Occurrences;

        // Add the first occurrence
        occurrence1 = occurrences.AddByFilename(@"C:\Screw.par",
Missing.Value);
```

```
// Get a reference to the occurrence document
part = (SolidEdgePart.PartDocument)occurrence1.OccurrenceDocument;

// Get a reference to the models collection
models = part.Models;

// Get a reference to the one and only model
model = models.Item(1);

// Get a reference to the revolved protrusions collection
revolvedProtrusions = model.RevolvedProtrusions;

// Get a reference to the first revolved protrusion
revolvedProtrusion = revolvedProtrusions.Item(1);

// Get a reference to the side faces collection
faces = (SolidEdgeGeometry.Faces)revolvedProtrusion.SideFaces;

// Loop through the faces
for (int i = 1; i <= faces.Count; i++)
{
  // Get a reference to the face object
  face = (SolidEdgeGeometry.Face)faces.Item(i);

  // Get a reference to the geometry object
  geometry = face.Geometry;

  // Determine the face type
  GNTTypePropertyConstants typeProperty = (GNTTypePropertyConstants)
    geometry.GetType().InvokeMember(
      "Type",
      BindingFlags.GetProperty,
      null,
      geometry,
      null);

  if (typeProperty == GNTTypePropertyConstants.igCylinder)
  {
    screwConicalFace = face;
    break;
  }
}

// Create the first reference
refToCylinderInScrew = (SolidEdgeFramework.Reference)
  assembly.CreateReference(occurrence1, screwConicalFace);

// Add the second occurrence
occurrence2 = occurrences.AddByFilename(
  @"C:\Nut.par", Missing.Value);

// Get a reference to the occurrence document
part = (SolidEdgePart.PartDocument)occurrence2.OccurrenceDocument;

// Get a reference to the models collection
models = part.Models;
```

```
        // Get a reference to the one and only model
        model = models.Item(1);

        // Get a reference to the revolved cutouts collection
        revolvedCutouts = model.RevolvedCutouts;

        // Get a reference to the first revolved cutout
        revolvedCutout = revolvedCutouts.Item(1);

        // Get a reference to the side faces collection
        faces = (SolidEdgeGeometry.Faces)revolvedCutout.SideFaces;

        // Loop through the faces
        for (int i = 1; i <= faces.Count; i++)
        {
          // Get a reference to the face object
          face = (SolidEdgeGeometry.Face)faces.Item(i);

          // Get a reference to the geometry object
          geometry = face.Geometry;

          // Determine the face type
          GNTTypePropertyConstants typeProperty = (GNTTypePropertyConstants)
            geometry.GetType().InvokeMember(
              "Type",
              BindingFlags.GetProperty,
              null,
              geometry,
              null);

          if (typeProperty == GNTTypePropertyConstants.igCone)
          {
            nutConicalFace = face;
            break;
          }
        }

        // Create the second reference
        refToConeInNut = (SolidEdgeFramework.Reference)
          assembly.CreateReference(occurrence2, nutConicalFace);

        // All Occurrences placed through automation are placed "Grounded."
        // You must delete the ground constraint on the second Occurrence
        // before you can place other relationships.
        relations3d = assembly.Relations3d;
        groundRelation3d = (SolidEdgeAssembly.GroundRelation3d)
          relations3d.Item(2);
        groundRelation3d.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.
        relNuttoScrew = relations3d.AddAxial(
          refToConeInNut,
          refToCylinderInScrew,
          false);
```

```
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (relNuttoScrew != null)
        {
          Marshal.ReleaseComObject(relNuttoScrew);
          relNuttoScrew = null;
        }
        if (groundRelation3d != null)
        {
          Marshal.ReleaseComObject(groundRelation3d);
          groundRelation3d = null;
        }
        if (relations3d != null)
        {
          Marshal.ReleaseComObject(relations3d);
          relations3d = null;
        }
        if (refToConeInNut != null)
        {
          Marshal.ReleaseComObject(refToConeInNut);
          refToConeInNut = null;
        }
        if (refToCylinderInScrew != null)
        {
          Marshal.ReleaseComObject(refToCylinderInScrew);
          refToCylinderInScrew = null;
        }
        if (nutConicalFace != null)
        {
          Marshal.ReleaseComObject(nutConicalFace);
          nutConicalFace = null;
        }
        if (screwConicalFace != null)
        {
          Marshal.ReleaseComObject(screwConicalFace);
          screwConicalFace = null;
        }
        if (geometry != null)
        {
          Marshal.ReleaseComObject(geometry);
          geometry = null;
        }
        if (face != null)
        {
          Marshal.ReleaseComObject(face);
          face = null;
        }
        if (faces != null)
        {
          Marshal.ReleaseComObject(faces);
          faces = null;
        }
        if (revolvedCutout != null)
        {
          Marshal.ReleaseComObject(revolvedCutout);
          revolvedCutout = null;
        }
        if (revolvedCutouts != null)
        {
          Marshal.ReleaseComObject(revolvedCutouts);
          revolvedCutouts = null;
        }
        if (revolvedProtrusion != null)
        {
          Marshal.ReleaseComObject(revolvedProtrusion);
          revolvedProtrusion = null;
        }
        if (revolvedProtrusions != null)
        {
          Marshal.ReleaseComObject(revolvedProtrusions);
          revolvedProtrusions = null;
        }
        if (model != null)
        {
          Marshal.ReleaseComObject(model);
          model = null;
        }
        if (models != null)
        {
          Marshal.ReleaseComObject(models);
          models = null;
        }
        if (part != null)
        {
          Marshal.ReleaseComObject(part);
          part = null;
        }
        if (occurrence2 != null)
        {
          Marshal.ReleaseComObject(occurrence2);
          occurrence2 = null;
        }
        if (occurrence1 != null)
        {
          Marshal.ReleaseComObject(occurrence1);
          occurrence1 = null;
        }
        if (occurrences != null)
        {
          Marshal.ReleaseComObject(occurrences);
          occurrences = null;
        }
        if (assembly != null)
        {
          Marshal.ReleaseComObject(assembly);
          assembly = null;
        }
        if (documents != null)
        {
          Marshal.ReleaseComObject(documents);
          documents = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

Chapter 8 - Draft Documents

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

Sections and Sheets

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:

- **Working Section** - Contains Sheet objects (Working Sheets) on which normal 2-D drawing objects (Lines, Arcs, DrawingViews, and so forth) are placed.
- **Background Section** - Contains background sheets, which hold the sheet borders.
- **2D Model Section** - 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.

Sections and Sheets Example (Visual Basic .NET)

```
Imports SolidEdgeDraft
Imports System.Runtime.InteropServices

Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objDocuments As SolidEdgeFramework.Documents = Nothing
    Dim objDraft As SolidEdgeDraft.DraftDocument = Nothing
    Dim objSections As SolidEdgeDraft.Sections = Nothing
    Dim objSection As SolidEdgeDraft.Section = Nothing
    Dim objSectionSheets As SolidEdgeDraft.SectionSheets = Nothing
    Dim objSheets As SolidEdgeDraft.Sheets = Nothing
    Dim objSheet As SolidEdgeDraft.Sheet = Nothing
    Dim strFormat1 As String = "Section = {0}"
    Dim strFormat2 As String = "Sheet = {0}"

    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the documents collection
      objDocuments = objApplication.Documents

      ' Add a Draft document
      objDraft = objDocuments.Add("SolidEdge.DraftDocument")

      ' Get a reference to the sections collection
      objSections = objDraft.Sections

      ' Loop through the sections
      ' igWorkingSection, igBackgroundSection & ig2dModelSection
      For Each objSection In objSections
        ' Output the section type
        Console.WriteLine( _
          String.Format(strFormat1, objSection.Type.ToString()))

        ' Get a reference to the section sheets collection
        objSectionSheets = objSection.Sheets

        ' Loop through the sheets
        For Each objSheet In objSectionSheets
          ' Output the sheet name
          Console.WriteLine(String.Format(strFormat2, objSheet.Name))
        Next
      Next

      ' Access the igWorkingSection directly
      objSection = objSections.WorkingSection

      ' Access the igBackgroundSection directly
      objSection = objSections.BackgroundSection

      ' Get a reference to the sheets collection
      objSheets = objDraft.Sheets

      ' Add a new sheet
      objSheet = objSheets.AddSheet( _
        "Sheet2", SheetSectionTypeConstants.igWorkingSection)

      ' Make the newly added sheet active
      objSheet.Activate()

    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objSheet Is Nothing) Then
        Marshal.ReleaseComObject(objSheet)
        objSheet = Nothing
      End If
      If Not (objSheets Is Nothing) Then
        Marshal.ReleaseComObject(objSheets)
        objSheets = Nothing
      End If
      If Not (objSectionSheets Is Nothing) Then
        Marshal.ReleaseComObject(objSectionSheets)
        objSectionSheets = Nothing
      End If
      If Not (objSection Is Nothing) Then
        Marshal.ReleaseComObject(objSection)
        objSection = Nothing
      End If
      If Not (objSections Is Nothing) Then
        Marshal.ReleaseComObject(objSections)
        objSections = Nothing
      End If
      If Not (objDraft Is Nothing) Then
        Marshal.ReleaseComObject(objDraft)
        objDraft = Nothing
      End If
      If Not (objDocuments Is Nothing) Then
        Marshal.ReleaseComObject(objDocuments)
        objDocuments = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Sections and Sheets Example (C#)

```
using SolidEdgeDraft;
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.Documents documents = null;
      SolidEdgeDraft.DraftDocument draft = null;
      SolidEdgeDraft.Sections sections = null;
      SolidEdgeDraft.Section section = null;
      SolidEdgeDraft.SectionSheets sectionSheets = null;
      SolidEdgeDraft.Sheets sheets = null;
      SolidEdgeDraft.Sheet sheet = null;
      string format1 = "Section = {0}";
      string format2 = "Sheet = {0}";
      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the documents collection
        documents = application.Documents;

        // Add a Draft document
        draft = (SolidEdgeDraft.DraftDocument)
          documents.Add("SolidEdge.DraftDocument", Missing.Value);

        // Get a reference to the sections collection
        sections = draft.Sections;

        // Loop through the sections
        // igWorkingSection, igBackgroundSection & ig2dModelSection
        for (int i = 1; i <= sections.Count; i++)
        {
          section = sections.Item(i);

          // Get a reference to the section sheets collection
          sectionSheets = section.Sheets;

          // Output the section type
          Console.WriteLine(String.Format(format1, section.Type.ToString()));

          // Loop through the sheets
          for (int j = 1; j <= sectionSheets.Count; j++)
          {
            sheet = sectionSheets.Item(j);

            // Output the sheet name
            Console.WriteLine(String.Format(format2, sheet.Name));
          }

          // Access the igWorkingSection directly
          section = sections.WorkingSection;

          // Access the igBackgroundSection directly
          section = sections.BackgroundSection;

          // Get a reference to the sheets collection
          sheets = draft.Sheets;

          // Add a new sheet
          sheet = sheets.AddSheet(
            "Sheet2",
            SheetSectionTypeConstants.igWorkingSection,
            Missing.Value,
            Missing.Value);

          // Make the newly added sheet active
          sheet.Activate();
        }
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (sheet != null)
        {
          Marshal.ReleaseComObject(sheet);
          sheet = null;
        }
        if (sheets != null)
        {
          Marshal.ReleaseComObject(sheets);
          sheets = null;
        }
        if (sectionSheets != null)
        {
          Marshal.ReleaseComObject(sectionSheets);
          sectionSheets = null;
        }
        if (section != null)
        {
          Marshal.ReleaseComObject(section);
          section = null;
        }
        if (sections != null)
        {
          Marshal.ReleaseComObject(sections);
          sections = null;
        }
        if (draft != null)
        {
          Marshal.ReleaseComObject(draft);
          draft = null;
        }
        if (documents != null)
        {
          Marshal.ReleaseComObject(documents);
          documents = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

SmartFrames

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.

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.

Linking and Embedding Example (Visual Basic .NET)

```
Imports SolidEdgeDraft
Imports System.Runtime.InteropServices

Module Module1
```

```
Sub Main()
  Dim objApplication As SolidEdgeFramework.Application = Nothing
  Dim objDocuments As SolidEdgeFramework.Documents = Nothing
  Dim objDraft As SolidEdgeDraft.DraftDocument = Nothing
  Dim objSheet As SolidEdgeDraft.Sheet = Nothing
  Dim objSmartFrames2d As SolidEdgeFrameworkSupport.SmartFrames2d = Nothing
  Dim objSmartFrame2d As SolidEdgeFrameworkSupport.SmartFrame2d = Nothing

  Try
    ' Connect to a running instance of Solid Edge
    objApplication = Marshal.GetActiveObject("SolidEdge.Application")

    ' Get a reference to the documents collection
    objDocuments = objApplication.Documents

    ' Add a Draft document
    objDraft = objDocuments.Add("SolidEdge.DraftDocument")

    ' Get a reference to the active sheet
    objSheet = objDraft.ActiveSheet

    ' Get a reference to the smart frames 2d collection
    objSmartFrames2d = objSheet.SmartFrames2d

    ' Create a SmartFrame2d object by two points.
    objSmartFrame2d = objSmartFrames2d.AddBy2Points( _
      "", 0.05, 0.05, 0.1, 0.1)

    ' Add a description to the SmartFrame
    objSmartFrame2d.Description = "My SmartFrame2d"

    ' Embed document within the SmartFrame
    objSmartFrame2d.CreateEmbed("C:\MyFile.doc")

    ' or

    ' Link document within the SmartFrame
    'objSmartFrame2d.CreateLink("C:\MyFile.doc")
  Catch ex As Exception
    Console.WriteLine(ex.Message)
  Finally
    If Not (objSmartFrame2d Is Nothing) Then
      Marshal.ReleaseComObject(objSmartFrame2d)
      objSmartFrame2d = Nothing
    End If
    If Not (objSmartFrames2d Is Nothing) Then
      Marshal.ReleaseComObject(objSmartFrames2d)
      objSmartFrames2d = Nothing
    End If
    If Not (objSheet Is Nothing) Then
      Marshal.ReleaseComObject(objSheet)
      objSheet = Nothing
    End If
    If Not (objDraft Is Nothing) Then
      Marshal.ReleaseComObject(objDraft)
      objDraft = Nothing
    End If
```

```
      If Not (objDocuments Is Nothing) Then
        Marshal.ReleaseComObject(objDocuments)
        objDocuments = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Linking and Embedding Example (C#)

```
using SolidEdgeDraft;
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.Documents documents = null;
      SolidEdgeDraft.DraftDocument draft = null;
      SolidEdgeDraft.Sheet sheet = null;
      SolidEdgeFrameworkSupport.SmartFrames2d smartFrames2d = null;
      SolidEdgeFrameworkSupport.SmartFrame2d smartFrame2d = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the documents collection
        documents = application.Documents;

        // Add a Draft document
        draft = (SolidEdgeDraft.DraftDocument)
          documents.Add("SolidEdge.DraftDocument", Missing.Value);

        // Get a reference to the active sheet
        sheet = draft.ActiveSheet;

        // Get a reference to the smart frames 2d collection
        smartFrames2d = (SolidEdgeFrameworkSupport.SmartFrames2d)
          sheet.SmartFrames2d;

        // Create a SmartFrame2d object by two points.
        smartFrame2d = smartFrames2d.AddBy2Points(
          "",
          0.02,
          0.02,
          0.07,
          0.07);

        // Add a description to the SmartFrame
        smartFrame2d.Description = "My SmartFrame2d";

        // Embed document within the SmartFrame
        smartFrame2d.CreateEmbed(@"C:\MyFile.doc", Missing.Value);

        // or

        // Link document within the SmartFrame
        smartFrame2d.CreateLink(@"C:\MyFile.doc", Missing.Value);
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (smartFrame2d != null)
        {
          Marshal.ReleaseComObject(smartFrame2d);
          smartFrame2d = null;
        }
        if (smartFrames2d != null)
        {
          Marshal.ReleaseComObject(smartFrames2d);
          smartFrames2d = null;
        }
        if (sheet != null)
        {
          Marshal.ReleaseComObject(sheet);
          sheet = null;
        }
        if (draft != null)
        {
          Marshal.ReleaseComObject(draft);
          draft = null;
        }
        if (documents != null)
        {
          Marshal.ReleaseComObject(documents);
          documents = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

Symbols

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 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.

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.

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.

Symbols Example (Visual Basic .NET)

```
Imports SolidEdgeFramework
Imports System.Runtime.InteropServices
```

```
Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objDocuments As SolidEdgeFramework.Documents = Nothing
    Dim objDraft As SolidEdgeDraft.DraftDocument = Nothing
    Dim objSheet As SolidEdgeDraft.Sheet = Nothing
    Dim objSymbols As SolidEdgeFramework.Symbols = Nothing
    Dim objSymbol1 As SolidEdgeFramework.Symbol2d = Nothing
    Dim objSymbol2 As SolidEdgeFramework.Symbol2d = Nothing
    Dim objSymbol3 As SolidEdgeFramework.Symbol2d = Nothing
    Dim x As Double
    Dim y As Double
    Dim strSourceDoc As String
    Dim objWordDoc As Object = Nothing
    Dim objWordApp As Object = Nothing

    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the documents collection
      objDocuments = objApplication.Documents

      ' Add a Draft document
      objDraft = objDocuments.Add("SolidEdge.DraftDocument")

      ' Get a reference to the active sheet
      objSheet = objDraft.ActiveSheet

      ' Get a reference to the symbols collection
      objSymbols = objSheet.Symbols

      ' Create a linked symbol
      objSymbol1 = objSymbols.Add( _
        OLEInsertionTypeConstant.igOLELinked, _
        "C:\MyFile.doc", 0.1, 0.1)

      ' Create a embedded symbol
      objSymbol2 = objSymbols.Add( _
        OLEInsertionTypeConstant.igOLEEmbedded, _
        "C:\MyFile.doc", 0.1, 0.2)

      ' Create a shared embedded symbol
      objSymbol3 = objSymbols.Add( _
        OLEInsertionTypeConstant.igOLESharedEmbedded, _
        "C:\MyFile.doc", 0.1, 0.3)

      ' Retrieve the origin of the first symbol
      objSymbol1.GetOrigin(x, y)

      ' Modify the first symbol's origin
      objSymbol1.SetOrigin(x + 0.1, y + 0.1)

      ' Set the angle of rotation of the first symbol to 45 degrees
      ' (in radians)
      objSymbol1.Angle = 45 * (Math.PI / 180)
      ' Find the path to the linked document
      If objSymbol1.OLEType = OLEInsertionTypeConstant.igOLELinked Then
        strSourceDoc = objSymbol1.SourceDoc
      End If

      ' Open the source document to modify it
      objSymbol1.DoVerb( _
        SolidEdgeConstants.StandardOLEVerbConstants.igOLEOpen)

      ' In this case, we know that we're dealing with a word document
      ' Get a reference the source document dispatch interface
      ' At this point, you can use the Word API to manipulate the document
      objWordDoc = objSymbol1.Object

      ' Get a reference to the word application object
      objWordApp = objWordDoc.Application

      ' Save and close the word document
      objWordDoc.Save()
      objWordDoc.Close()

      ' Quit word
      objWordApp.Quit()
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objWordApp Is Nothing) Then
        Marshal.ReleaseComObject(objWordApp)
        objWordApp = Nothing
      End If
      If Not (objWordDoc Is Nothing) Then
        Marshal.ReleaseComObject(objWordDoc)
        objWordDoc = Nothing
      End If
      If Not (objSymbol3 Is Nothing) Then
        Marshal.ReleaseComObject(objSymbol3)
        objSymbol3 = Nothing
      End If
      If Not (objSymbol2 Is Nothing) Then
        Marshal.ReleaseComObject(objSymbol2)
        objSymbol2 = Nothing
      End If
      If Not (objSymbol1 Is Nothing) Then
        Marshal.ReleaseComObject(objSymbol1)
        objSymbol1 = Nothing
      End If
      If Not (objSymbols Is Nothing) Then
        Marshal.ReleaseComObject(objSymbols)
        objSymbols = Nothing
      End If
      If Not (objSheet Is Nothing) Then
        Marshal.ReleaseComObject(objSheet)
        objSheet = Nothing
      End If
      If Not (objDraft Is Nothing) Then
        Marshal.ReleaseComObject(objDraft)
        objDraft = Nothing
      End If
      If Not (objDocuments Is Nothing) Then
        Marshal.ReleaseComObject(objDocuments)
        objDocuments = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Symbols Example (C#)

```
using SolidEdgeFramework;
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.Documents documents = null;
      SolidEdgeDraft.DraftDocument draft = null;
      SolidEdgeDraft.Sheet sheet = null;
      SolidEdgeFramework.Symbols symbols = null;
      SolidEdgeFramework.Symbol2d symbol1 = null;
      SolidEdgeFramework.Symbol2d symbol2 = null;
      SolidEdgeFramework.Symbol2d symbol3 = null;
      double x = 0;
      double y = 0;
      string strSouceDoc;
      object wordDoc = null;
      object wordApp = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the documents collection
        documents = application.Documents;

        // Add a Draft document
        draft = (SolidEdgeDraft.DraftDocument)
          documents.Add("SolidEdge.DraftDocument", Missing.Value);

        // Get a reference to the active sheet
        sheet = draft.ActiveSheet;
```

```
// Get a reference to the symbols collection
symbols = (SolidEdgeFramework.Symbols)sheet.Symbols;

// Create a linked symbol
symbol1 = symbols.Add(
  (int)OLEInsertionTypeConstant.igOLELinked,
  @"C:\MyFile.doc",
  0.1,
  0.1,
  Missing.Value);

// Create a embedded symbol
symbol2 = symbols.Add(
  (int)OLEInsertionTypeConstant.igOLEEmbedded,
  @"C:\MyFile.doc",
  0.1,
  0.2,
  Missing.Value);

// Create a shared embedded symbol
symbol3 = symbols.Add(
  (int)OLEInsertionTypeConstant.igOLESharedEmbedded,
  @"C:\MyFile.doc",
  0.1,
  0.3,
  Missing.Value);

// Retrieve the origin of the first symbol
symbol1.GetOrigin(out x, out y);

// Modify the first symbol's origin
symbol1.SetOrigin(x + 0.2, y + 0.2);

// Set the angle of rotation of the first symbol to 45 degrees
// (in radians)
symbol1.Angle = 45 * (Math.PI / 180);

// Find the path to the linked document
if (symbol1.OLEType == OLEInsertionTypeConstant.igOLELinked)
{
  strSouceDoc = symbol1.SourceDoc;
}

// Open the source document to modify it
symbol1.DoVerb(
  SolidEdgeConstants.StandardOLEVerbConstants.igOLEOpen);

// In this case, we know that we're dealing with a word document
// Get a reference the source document dispatch interface
// At this point, you can use the Word API to manipulate the document
wordDoc = symbol1.Object;

// Get a reference to the word application object
wordApp = wordDoc.GetType().InvokeMember(
  "Application", BindingFlags.GetProperty, null, wordDoc, null);
```

```
        // Save and close the word document
        wordDoc.GetType().InvokeMember(
          "Save", BindingFlags.InvokeMethod, null, wordDoc, null);
        wordDoc.GetType().InvokeMember(
          "Close", BindingFlags.InvokeMethod, null, wordDoc, null);

        // Quit word
        wordApp.GetType().InvokeMember(
          "Quit", BindingFlags.InvokeMethod, null, wordApp, null);
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (wordApp != null)
        {
          Marshal.ReleaseComObject(wordApp);
          wordApp = null;
        }
        if (wordDoc != null)
        {
          Marshal.ReleaseComObject(wordDoc);
          wordDoc = null;
        }
        if (symbol3 != null)
        {
          Marshal.ReleaseComObject(symbol3);
          symbol3 = null;
        }
        if (symbol2 != null)
        {
          Marshal.ReleaseComObject(symbol2);
          symbol2 = null;
        }
        if (symbol1 != null)
        {
          Marshal.ReleaseComObject(symbol1);
          symbol1 = null;
        }
        if (symbols != null)
        {
          Marshal.ReleaseComObject(symbols);
          symbols = null;
        }
        if (sheet != null)
        {
          Marshal.ReleaseComObject(sheet);
          sheet = null;
        }
        if (draft != null)
        {
          Marshal.ReleaseComObject(draft);
          draft = null;
        }
        if (documents != null)
        {
          Marshal.ReleaseComObject(documents);
          documents = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

DrawingViews

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.

DrawingViews Example (Visual Basic .NET)

```
Imports SolidEdgeFramework
Imports System.Runtime.InteropServices

Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objDocuments As SolidEdgeFramework.Documents = Nothing
    Dim objDraft As SolidEdgeDraft.DraftDocument = Nothing
    Dim objSheet As SolidEdgeDraft.Sheet = Nothing
    Dim objModelLinks As SolidEdgeDraft.ModelLinks = Nothing
    Dim objModelLink As SolidEdgeDraft.ModelLink = Nothing
    Dim objDrawingViews As SolidEdgeDraft.DrawingViews = Nothing
    Dim objDrawingView As SolidEdgeDraft.DrawingView = Nothing


    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the documents collection
      objDocuments = objApplication.Documents

      ' Add a Draft document
      objDraft = objDocuments.Add("SolidEdge.DraftDocument")

      ' Get a reference to the active sheet
      objSheet = objDraft.ActiveSheet

      ' Get a reference to the model links collection
      objModelLinks = objDraft.ModelLinks
      ' Add a new model link
      objModelLink = objModelLinks.Add("C:\Part1.par")

      ' Get a reference to the drawing views collection
      objDrawingViews = objSheet.DrawingViews

      ' Add a new drawing view
      objDrawingView = objDrawingViews.AddPartView( _
        objModelLink, _
        SolidEdgeDraft.ViewOrientationConstants.igFrontView, _
        1, _
        0.1, _
        0.1, _
        SolidEdgeDraft.PartDrawingViewTypeConstants.sePartDesignedView)

      ' Assign a caption
      objDrawingView.Caption = "My New Drawing View"

      ' Ensure caption is displayed
      objDrawingView.DisplayCaption = True

    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objDrawingView Is Nothing) Then
        Marshal.ReleaseComObject(objDrawingView)
        objDrawingView = Nothing
      End If
      If Not (objDrawingViews Is Nothing) Then
        Marshal.ReleaseComObject(objDrawingViews)
        objDrawingViews = Nothing
      End If
      If Not (objModelLink Is Nothing) Then
        Marshal.ReleaseComObject(objModelLink)
        objModelLink = Nothing
      End If
      If Not (objModelLinks Is Nothing) Then
        Marshal.ReleaseComObject(objModelLinks)
        objModelLinks = Nothing
      End If
      If Not (objSheet Is Nothing) Then
        Marshal.ReleaseComObject(objSheet)
        objSheet = Nothing
      End If
      If Not (objDraft Is Nothing) Then
        Marshal.ReleaseComObject(objDraft)
        objDraft = Nothing
      End If
      If Not (objDocuments Is Nothing) Then
        Marshal.ReleaseComObject(objDocuments)
        objDocuments = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
```

```
  End Sub
End Module
```

DrawingViews Example (C#)

```
using SolidEdgeFramework;
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.Documents documents = null;
      SolidEdgeDraft.DraftDocument draft = null;
      SolidEdgeDraft.Sheet sheet = null;
      SolidEdgeDraft.ModelLinks modelLinks = null;
      SolidEdgeDraft.ModelLink modelLink = null;
      SolidEdgeDraft.DrawingViews drawingViews = null;
      SolidEdgeDraft.DrawingView drawingView = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the documents collection
        documents = application.Documents;

        // Add a Draft document
        draft = (SolidEdgeDraft.DraftDocument)
          documents.Add("SolidEdge.DraftDocument", Missing.Value);

        // Get a reference to the active sheet
        sheet = draft.ActiveSheet;

        // Get a reference to the model links collection
        modelLinks = draft.ModelLinks;

        // Add a new model link
        modelLink = modelLinks.Add(@"C:\Part1.par");

        // Get a reference to the drawing views collection
        drawingViews = sheet.DrawingViews;

        // Add a new drawing view
        drawingView = drawingViews.AddPartView(
          modelLink,
          SolidEdgeDraft.ViewOrientationConstants.igFrontView,
          1,
          0.1,
          0.1,
          SolidEdgeDraft.PartDrawingViewTypeConstants.sePartDesignedView);

        // Assign a caption
        drawingView.Caption = "My New Drawing View";

        // Ensure caption is displayed
        drawingView.DisplayCaption = true;
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (drawingView != null)
        {
          Marshal.ReleaseComObject(drawingView);
          drawingView = null;
        }
        if (drawingViews != null)
        {
          Marshal.ReleaseComObject(drawingViews);
          drawingViews = null;
        }
        if (modelLink != null)
        {
          Marshal.ReleaseComObject(modelLink);
          modelLink = null;
        }
        if (modelLinks != null)
        {
          Marshal.ReleaseComObject(modelLinks);
          modelLinks = null;
        }
        if (sheet != null)
        {
          Marshal.ReleaseComObject(sheet);
          sheet = null;
        }
        if (draft != null)
        {
          Marshal.ReleaseComObject(draft);
          draft = null;
        }
        if (documents != null)
        {
          Marshal.ReleaseComObject(documents);
          documents = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
    }
}
```

Chapter 9 - Handling Events

Many objects in Solid Edge provide event sets. These 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.

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.

Application Events

Solid Edge has a core set of events that it fires during program execution. You can obtain a reference to these events by attaching to the ApplicationEvents property of the Application object.

The interface SolidEdgeFramework.ISEApplicationEvents contains the definitions that you will need to implement the events.

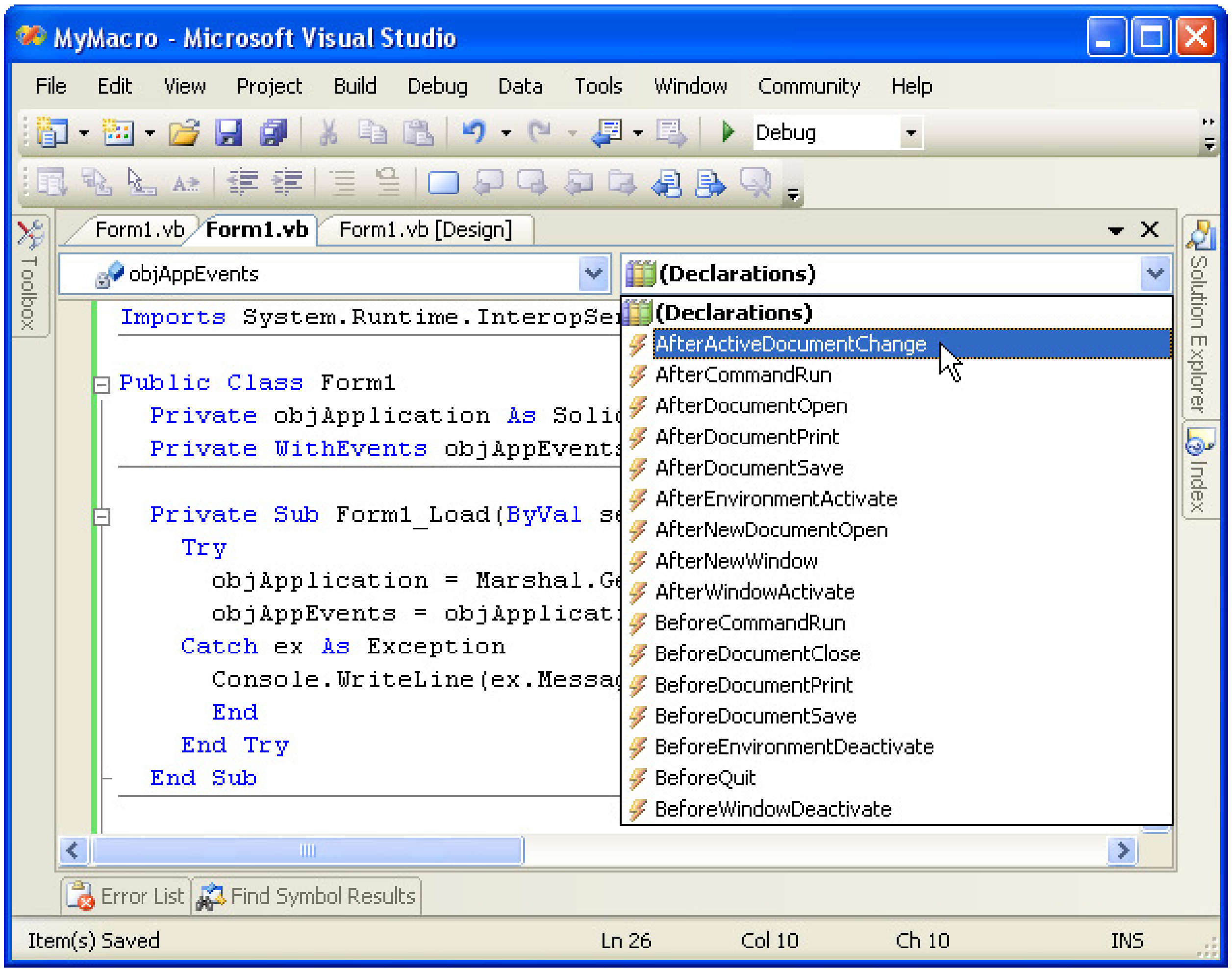
Table of Application Events

Event	Description
AfterActiveDocumentChange	Occurs after the active document changes.
AfterCommandRun	Occurs after a specified command is run.
AfterDocumentOpen	Occurs after a specified document is opened.
AfterDocumentPrint	Occurs after a specified document is printed.
AfterDocumentSave	Occurs when a specified document is saved.
AfterEnvironmentActivate	Occurs when a specified environment is activated.
AfterNewDocumentOpen	Occurs after a new document is opened
AfterNewWindow	Occurs after a new document is opened.
AfterWindowActivate	Occurs after a specified window is activated.
BeforeCommandRun	Occurs before a specified command is run.
BeforeDocumentClose	Occurs before a specified document is closed.
BeforeDocumentPrint	Occurs before a specified document is printed.
BeforeDocumentSave	Occurs before a specified document is saved.
BeforeEnvironmentDeactivate	Occurs before a specified environment is deactivated.
BeforeQuit	Occurs before the associated application is closed.
BeforeWindowDeactivate	Occurs before a specified window is deactivated.

Sinking Application Events (Visual Basic.NET)

```
Imports System.Runtime.InteropServices

Public Class Form1
  Private objApplication As SolidEdgeFramework.Application
  Private WithEvents objAppEvents As
SolidEdgeFramework.ISEApplicationEvents_Event

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the application events
      objAppEvents = objApplication.ApplicationEvents
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    End Try
  End Sub
  Private Sub Form1_FormClosing(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.FormClosingEventArgs) Handles MyBase.FormClosing
    If Not (objAppEvents Is Nothing) Then
      Marshal.ReleaseComObject(objAppEvents)
      objAppEvents = Nothing
    End If
    If Not (objApplication Is Nothing) Then
      Marshal.ReleaseComObject(objApplication)
      objApplication = Nothing
    End If
  End Sub
```

Once you have a valid reference to the events, you can select the specific event that you wish to watch from the Declarations dropdown.



```
Imports System.Runtime.InteropSe

Public Class Form1
    Private objApplication As Solic
    Private WithEvents objAppEvents

    Private Sub Form1_Load(ByVal se
        Try
            objApplication = Marshal.Ge
            objAppEvents = objApplicat
        Catch ex As Exception
            Console.WriteLine(ex.Messag
            End
        End Try
    End Sub
```

(Declarations)
AfterActiveDocumentChange
AfterCommandRun
AfterDocumentOpen
AfterDocumentPrint
AfterDocumentSave
AfterEnvironmentActivate
AfterNewDocumentOpen
AfterNewWindow
AfterWindowActivate
BeforeCommandRun
BeforeDocumentClose
BeforeDocumentPrint
BeforeDocumentSave
BeforeEnvironmentDeactivate
BeforeQuit
BeforeWindowDeactivate

Figure 9-1 - Sinking Application Events

Sinking Application Events (C#)

```
using System;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace MyMacro
{
  public partial class Form1 : Form
  {
    private SolidEdgeFramework.Application application;
    private SolidEdgeFramework.ISEApplicationEvents_Event appEvents;

    public Form1()
    {
      InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the application events
        appEvents = (SolidEdgeFramework.ISEApplicationEvents_Event)
          application.ApplicationEvents;

        // Example sink of AfterCommandRun
        appEvents.AfterCommandRun += new
SolidEdgeFramework.ISEApplicationEvents_AfterCommandRunEventHandler(appEvents
_AfterCommandRun);
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
    }

    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
      if (appEvents != null)
      {
        Marshal.ReleaseComObject(appEvents);
        appEvents = null;
      }
      if (application != null)
      {
        Marshal.ReleaseComObject(application);
        application = null;
      }
    }

    void appEvents_AfterCommandRun(int theCommandID)
    {
    }
  }
}
```

Application Window Events

The application window events allow you to watch messages being sent to the Solid Edge main window. This is similar subclassing a window and watching for SendMessage() calls. The WindowProc event is a "chatty" event so care should be taken when sinking this event. Long running code in this event can severely slow down the application. You can obtain a reference to these events by attaching to the ApplicationWindowEvents property of the Application object.

The interface SolidEdgeFramework.ISEApplicationWindowEvents contains the definitions that you will need to implement the events.

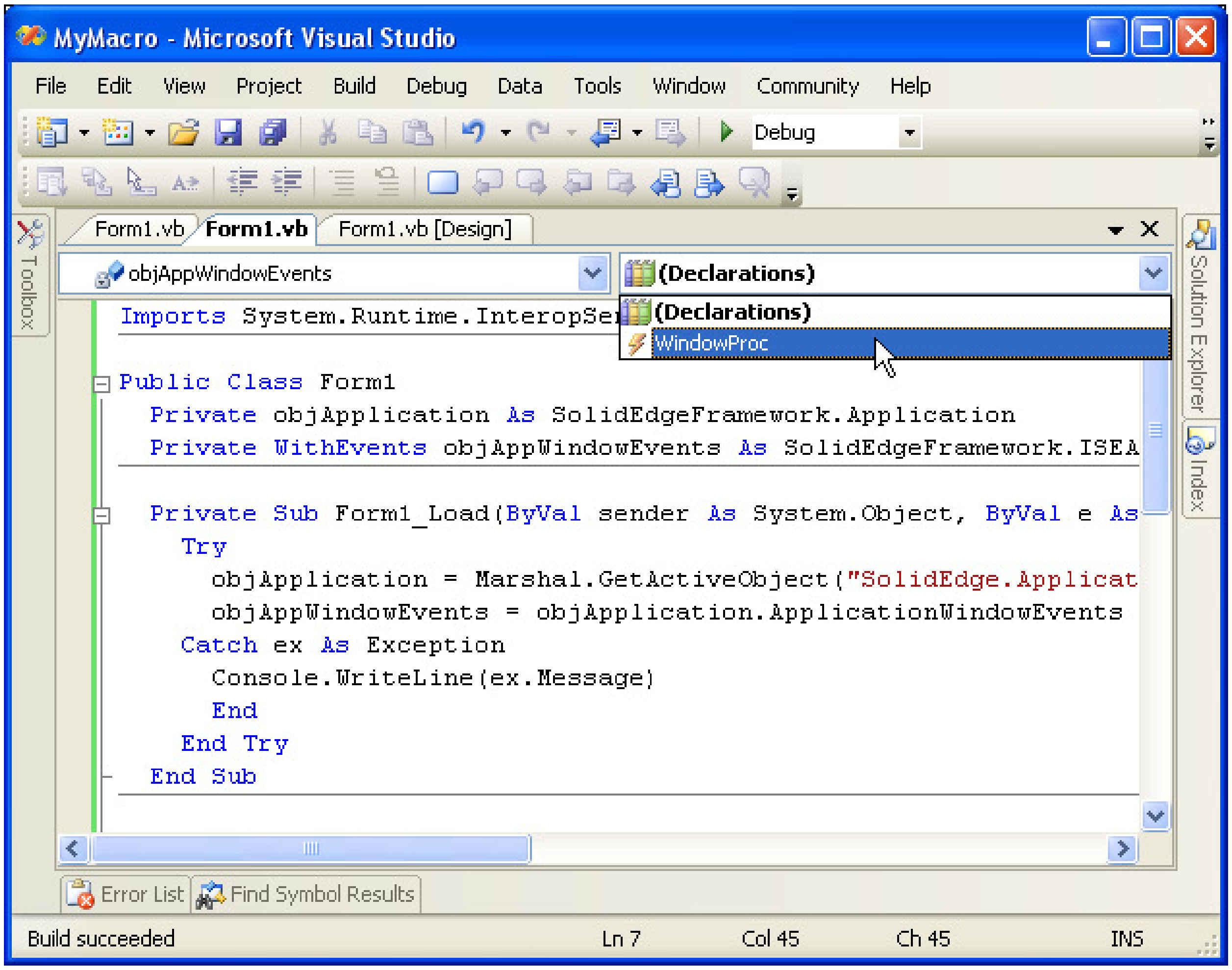
Table of Application Window Events

Event	Description
WindowProc	Occurs when the application receives a window event.

Sinking Application Window Events (Visual Basic.NET)

```
Imports System.Runtime.InteropServices
Public Class Form1
  Private objApplication As SolidEdgeFramework.Application
  Private WithEvents objAppWindowEvents As
SolidEdgeFramework.ISEApplicationWindowEvents_Event

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the application window events
      objAppWindowEvents = objApplication.ApplicationWindowEvents
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    End Try
  End Sub

  Private Sub Form1_FormClosing(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.FormClosingEventArgs) Handles MyBase.FormClosing
    If Not (objAppWindowEvents Is Nothing) Then
      Marshal.ReleaseComObject(objAppWindowEvents)
      objAppWindowEvents = Nothing
    End If
    If Not (objApplication Is Nothing) Then
      Marshal.ReleaseComObject(objApplication)
      objApplication = Nothing
    End If
  End Sub
  Private Sub objAppWindowEvents_WindowProc(ByVal hWnd As Integer, ByVal nMsg
As Integer, ByVal wParam As Integer, ByVal lParam As Integer) Handles
objAppWindowEvents.WindowProc

  End Sub
End Class
```

Once you have a valid reference to the events, you can select the specific event that you wish to watch from the Declarations dropdown.



```
Imports System.Runtime.InteropSe

Public Class Form1
  Private objApplication As SolidEdgeFramework.Application
  Private WithEvents objAppWindowEvents As SolidEdgeFramework.ISEA

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
    Try
      objApplication = Marshal.GetActiveObject("SolidEdge.Applicat
      objAppWindowEvents = objApplication.ApplicationWindowEvents
    Catch ex As Exception
      Console.WriteLine(ex.Message)
      End
    End Try
  End Sub
```

Figure 9-2 - Sinking Application Window Events

Sinking Application Window Events (C#)

```
using System;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace MyMacro
{
  public partial class Form1 : Form
  {
    private SolidEdgeFramework.Application application;
    private SolidEdgeFramework.ISEApplicationWindowEvents_Event
appWindowEvents;

    public Form1()
    {
      InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the application window events
        appWindowEvents =
          (SolidEdgeFramework.ISEApplicationWindowEvents_Event)
          application.ApplicationWindowEvents;

        // Example sink of WindowProc
        appWindowEvents.WindowProc += new
         SolidEdgeFramework.ISEApplicationWindowEvents_WindowProcEventHandler
         (appWindowEvents_WindowProc);
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
    }

    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
      if (appWindowEvents != null)
      {
        Marshal.ReleaseComObject(appWindowEvents);
        appWindowEvents = null;
      }
      if (application != null)
      {
        Marshal.ReleaseComObject(application);
        application = null;
      }
    }

    void appWindowEvents_WindowProc
     (int hWnd, int nMsg, int wParam, int lParam)
    {
    }
  }
}
```

Document Events

The document events allow you to watch events for a specific document object. You can obtain a reference to these events by attaching to the DocumentEvents property of any document object.

The interface SolidEdgeFramework. ISEDocumentEvents contains the definitions that you will need to implement the events.

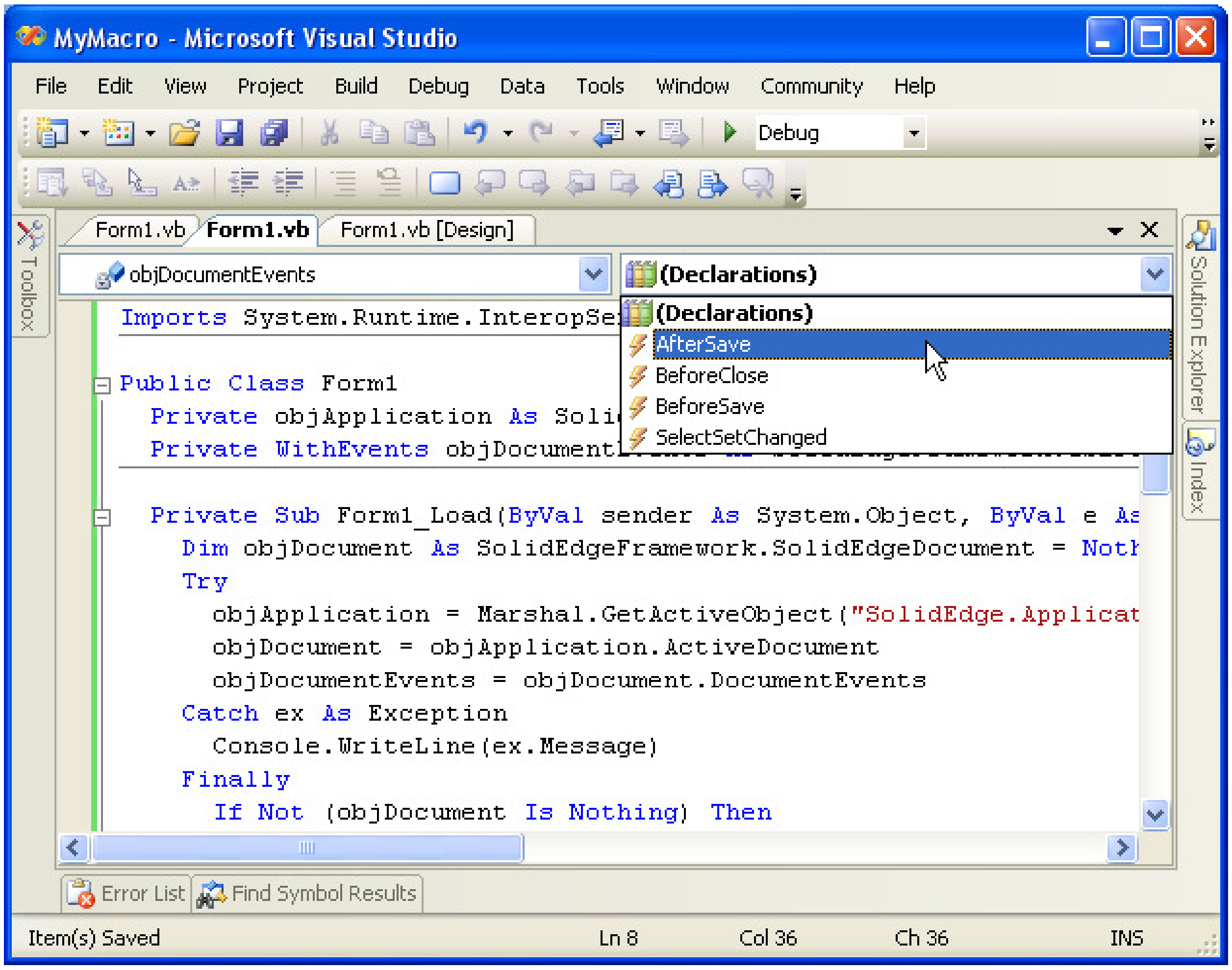
Table of Document Events

Event	Description
AfterSave	Occurs after the associated document is saved.
BeforeClose	Occurs before the associated document is closed.
BeforeSave	Occurs before the associated document is saved.
SelectSetChanged	Occurs when the specified selection set changes.

Sinking Document Events (Visual Basic.NET)

```
Imports System.Runtime.InteropServices
Public Class Form1
  Private objApplication As SolidEdgeFramework.Application
  Private WithEvents objDocumentEvents As
SolidEdgeFramework.ISEDocumentEvents_Event

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim objDocument As SolidEdgeFramework.SolidEdgeDocument = Nothing
    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the active document
      objDocument = objApplication.ActiveDocument

      ' Get a reference to the document events
      objDocumentEvents = objDocument.DocumentEvents
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objDocument Is Nothing) Then
        Marshal.ReleaseComObject(objDocument)
        objDocument = Nothing
      End If
    End Try
  End Sub

  Private Sub Form1_FormClosing(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.FormClosingEventArgs) Handles MyBase.FormClosing
    If Not (objDocumentEvents Is Nothing) Then
      Marshal.ReleaseComObject(objDocumentEvents)
      objDocumentEvents = Nothing
    End If
    If Not (objApplication Is Nothing) Then
      Marshal.ReleaseComObject(objApplication)
      objApplication = Nothing
    End If
  End Sub
End Class
```

Once you have a valid reference to the events, you can select the specific event that you wish to watch from the Declarations dropdown.



```
Imports System.Runtime.InteropSe

Public Class Form1
  Private objApplication As Soli
  Private WithEvents objDocument

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
    Dim objDocument As SolidEdgeFramework.SolidEdgeDocument = Noth
    Try
      objApplication = Marshal.GetActiveObject("SolidEdge.Applicat
      objDocument = objApplication.ActiveDocument
      objDocumentEvents = objDocument.DocumentEvents
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objDocument Is Nothing) Then
```

Figure 9-3 - Sinking Document Events

Sinking Document Events (C#)

```
using System;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace MyMacro
{
  public partial class Form1 : Form
  {
    private SolidEdgeFramework.Application application;
    private SolidEdgeFramework.ISEDocumentEvents_Event docEvents;
    public Form1()
    {
      InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
      SolidEdgeFramework.SolidEdgeDocument document = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the active document
        document =
          (SolidEdgeFramework.SolidEdgeDocument)
          application.ActiveDocument;

        // Get a reference to the document events
        docEvents = (SolidEdgeFramework.ISEDocumentEvents_Event)
          document.DocumentEvents;
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (document != null)
        {
          Marshal.ReleaseComObject(document);
        }
      }
    }

    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
      if (docEvents != null)
      {
        Marshal.ReleaseComObject(docEvents);
        docEvents = null;
      }
      if (application != null)
      {
        Marshal.ReleaseComObject(application);
        application = null;
      }
    }
  }
}
```

File UI Events

The File UI events allow you to watch for specific UI events. The File UI events are somewhat different in other events because if you do not wish to override the default functionality, you must throw a System.NotImplementedException. Failure to do so will indicate to Solid Edge that you have a custom UI that you wish to display to the user. You can obtain a reference to these events by attaching to the FileUIEvents property of the Application object.

The interface SolidEdgeFramework. ISEFileUIEvents contains the definitions that you will need to implement the events.

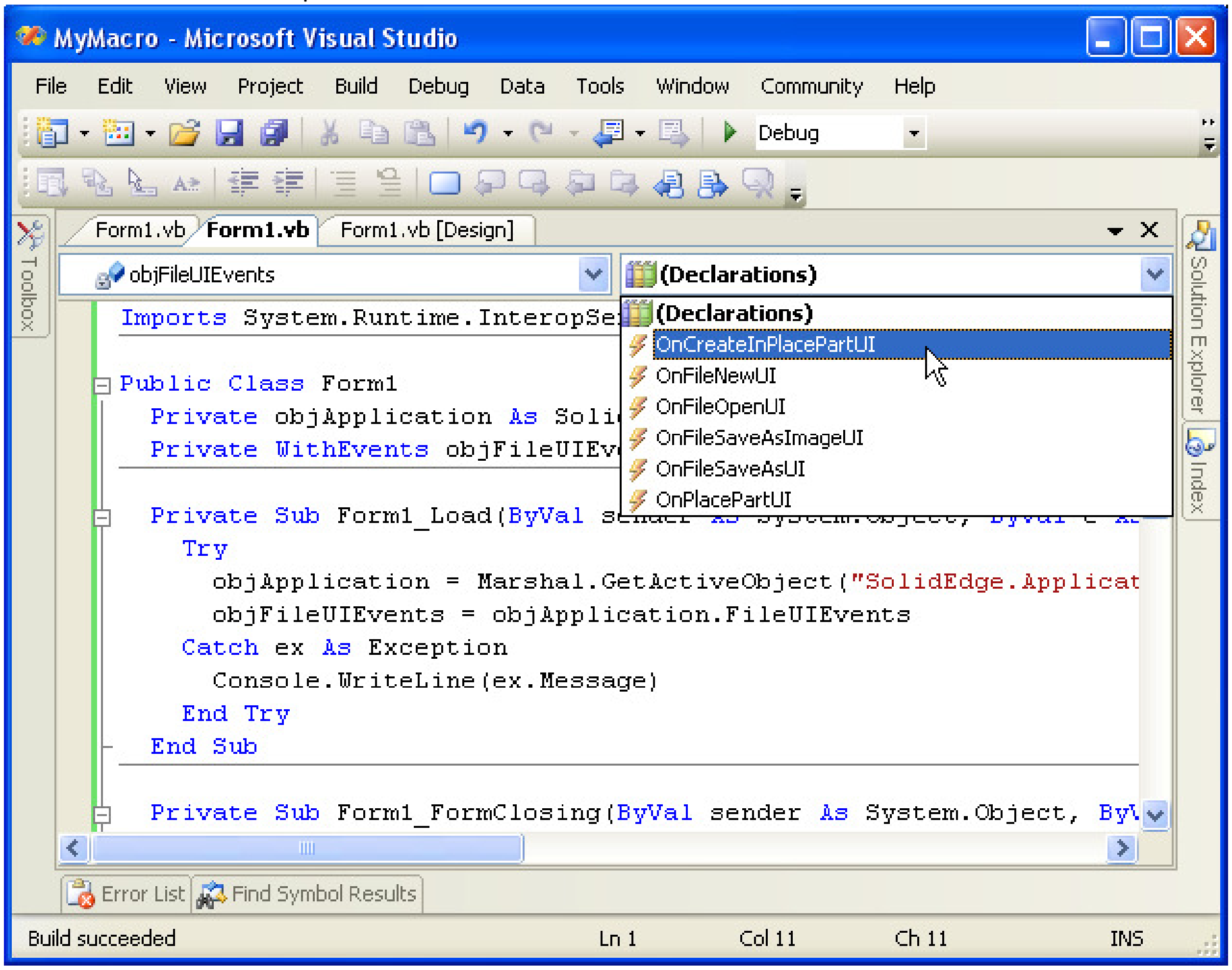
Table of File UI Events

Event	Description
OnCreateInPlacePartUI	Occurs before the user interface is displayed for a part created in place by Solid Edge Assembly.
OnFileNewUI	Occurs before the user interface is created for a new Solid Edge file.
OnFileOpenUI	Occurs before the creation of the user interface for the file opened by Solid Edge.
OnFileSaveAsImageUI	Occurs before the user interface is created for the file saved as image by Solid Edge.
OnFileSaveAsUI	Occurs before the user interface is created by the file saved by Solid Edge as another file.
OnPlacePartUI	Occurs before the user interface is created for a part placed by Solid Edge Assembly.

Sinking File UI Events (Visual Basic.NET)

```
Imports System.Runtime.InteropServices

Public Class Form1
  Private objApplication As SolidEdgeFramework.Application
  Private WithEvents objFileUIEvents As
SolidEdgeFramework.ISEFileUIEvents_Event

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the file UI events
      objFileUIEvents = objApplication.FileUIEvents
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    End Try
  End Sub

  Private Sub Form1_FormClosing(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.FormClosingEventArgs) Handles MyBase.FormClosing
    If Not (objFileUIEvents Is Nothing) Then
      Marshal.ReleaseComObject(objFileUIEvents)
      objFileUIEvents = Nothing
    End If
    If Not (objApplication Is Nothing) Then
      Marshal.ReleaseComObject(objApplication)
      objApplication = Nothing
    End If
  End Sub

  Private Sub objFileUIEvents_OnFileSaveAsUI(ByRef Filename As String, ByRef
AppendToTitle As String) Handles objFileUIEvents.OnFileSaveAsUI
    'If you do not plan on implementing your own UI but wish to allow
    'the default UI, you must throw a NotImplementedException exception.
    Throw New System.NotImplementedException()
  End Sub
End Class
```

Once you have a valid reference to the events, you can select the specific event that you wish to watch from the Declarations dropdown.



```
Imports System.Runtime.InteropSe

Public Class Form1
  Private objApplication As Soli
  Private WithEvents objFileUIEv

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
    Try
      objApplication = Marshal.GetActiveObject("SolidEdge.Applicat
      objFileUIEvents = objApplication.FileUIEvents
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    End Try
  End Sub

  Private Sub Form1_FormClosing(ByVal sender As System.Object, ByV
```

Figure 9-4 - Sinking File UI Events

Sinking File UI Events (C#)

```
using System;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace MyMacro
```

```
{
  public partial class Form1 : Form
  {
    private SolidEdgeFramework.Application application;
    private SolidEdgeFramework.ISEFileUIEvents_Event fileUIEvents;

    public Form1()
    {
      InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the file UI events
        fileUIEvents = (SolidEdgeFramework.ISEFileUIEvents_Event)
          application.FileUIEvents;

        // Example sink of OnFileSaveAsUI event
        fileUIEvents.OnFileSaveAsUI += new
SolidEdgeFramework.ISEFileUIEvents_OnFileSaveAsUIEventHandler(fileUIEvents_On
FileSaveAsUI);
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
    }

    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
      if (fileUIEvents != null)
      {
        Marshal.ReleaseComObject(fileUIEvents);
        fileUIEvents = null;
      }
      if (application != null)
      {
        Marshal.ReleaseComObject(application);
        application = null;
      }
    }

    void fileUIEvents_OnFileSaveAsUI(out string Filename, out string
AppendToTitle)
    {
      // If you do not plan on implementing your own UI but wish to allow
      // the default UI, you must throw a NotImplementedException exception.
      throw new System.NotImplementedException();
    }
  }
}
```

View Events

The view events allow you to watch for view events for a specific window's 3D view. You can obtain a reference to these events by attaching to the ViewEvents property of a View object.

The interface SolidEdgeFramework. ISEViewEvents contains the definitions that you will need to implement the events.

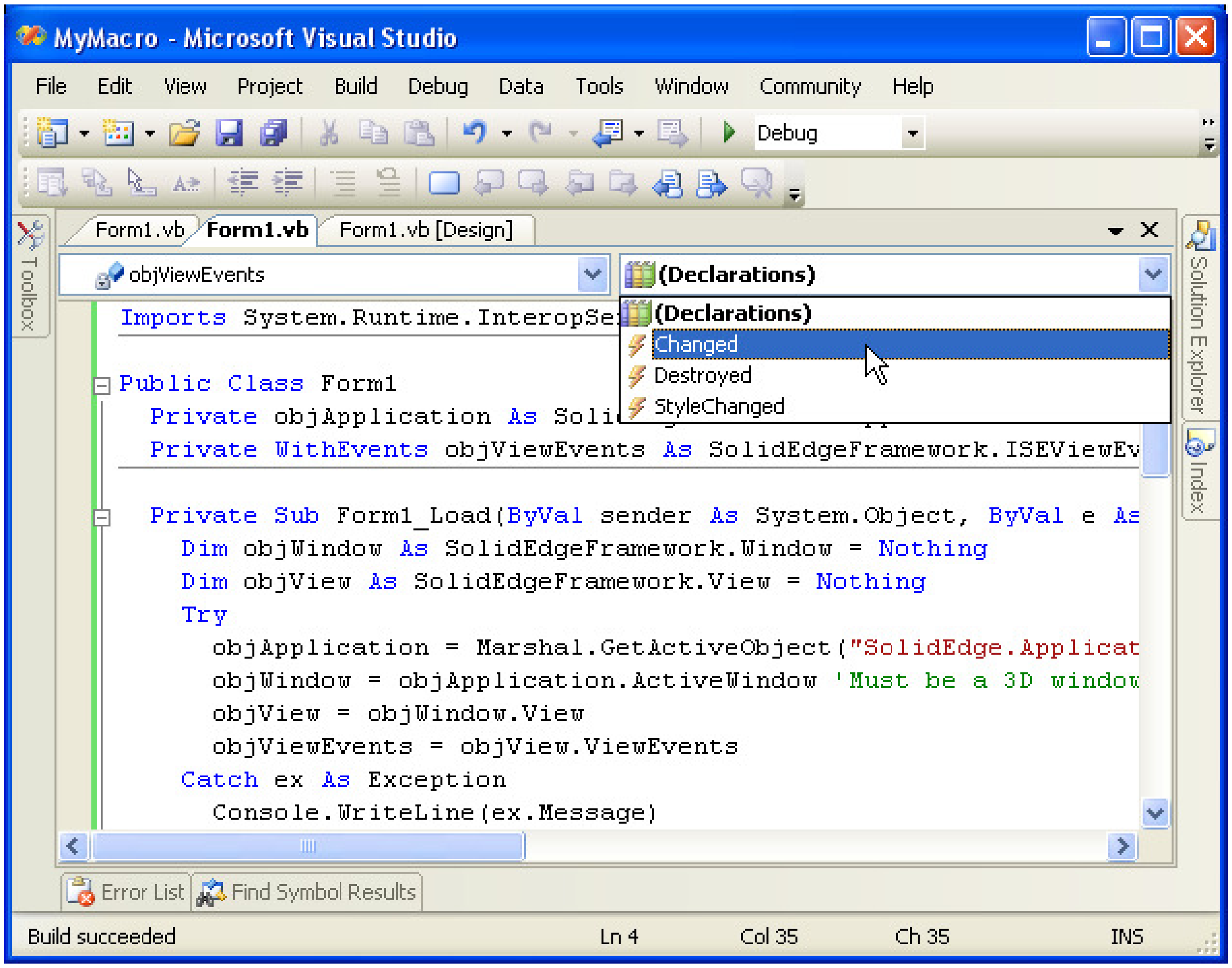
Table of View Events

Event	Description
Changed	Occurs when the associated view definition changes.
Destroyed	Occurs when the associated view is destroyed.
StyleChanged	Occurs when a view style changes.

Sinking View Events (Visual Basic.NET)

```
Imports System.Runtime.InteropServices

Public Class Form1
  Private objApplication As SolidEdgeFramework.Application
  Private WithEvents objViewEvents As SolidEdgeFramework.ISEViewEvents_Event

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim objWindow As SolidEdgeFramework.Window = Nothing
    Dim objView As SolidEdgeFramework.View = Nothing
    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the active window
      objWindow = objApplication.ActiveWindow 'Must be a 3D window

      ' Get a reference to the window's view
      objView = objWindow.View

      ' Get a reference to the view events
      objViewEvents = objView.ViewEvents
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objView Is Nothing) Then
        Marshal.ReleaseComObject(objView)
        objView = Nothing
      End If
      If Not (objWindow Is Nothing) Then
        Marshal.ReleaseComObject(objWindow)
        objWindow = Nothing
      End If
    End Try
  End Sub
  Private Sub Form1_FormClosing(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.FormClosingEventArgs) Handles MyBase.FormClosing
    If Not (objViewEvents Is Nothing) Then
      Marshal.ReleaseComObject(objViewEvents)
      objViewEvents = Nothing
    End If
    If Not (objApplication Is Nothing) Then
      Marshal.ReleaseComObject(objApplication)
      objApplication = Nothing
    End If
  End Sub
End Class
```

Once you have a valid reference to the events, you can select the specific event that you wish to watch from the Declarations dropdown.



```
Imports System.Runtime.InteropSe

Public Class Form1
  Private objApplication As Soli
  Private WithEvents objViewEvents As SolidEdgeFramework.ISEViewEv

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
    Dim objWindow As SolidEdgeFramework.Window = Nothing
    Dim objView As SolidEdgeFramework.View = Nothing
    Try
      objApplication = Marshal.GetActiveObject("SolidEdge.Applicat
      objWindow = objApplication.ActiveWindow 'Must be a 3D window
      objView = objWindow.View
      objViewEvents = objView.ViewEvents
    Catch ex As Exception
      Console.WriteLine(ex.Message)
```

Figure 9-5 - Sinking View Events

Sinking View Events (Visual Basic.NET)

```
using System;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace MyMacro
{
  public partial class Form1 : Form
  {
    private SolidEdgeFramework.Application application;
    private SolidEdgeFramework.ISEViewEvents_Event viewEvents;

    public Form1()
    {
      InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
      SolidEdgeFramework.Window window = null;
      SolidEdgeFramework.View view = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the active window
        window = (SolidEdgeFramework.Window)
          application.ActiveWindow; //Must be a 3D window

        // Get a reference to the window's view
        view = window.View;

        // Get a reference to the view events
        viewEvents = (SolidEdgeFramework.ISEViewEvents_Event)
          view.ViewEvents;

        // Example sink of the Chaged event
        viewEvents.Changed += new
SolidEdgeFramework.ISEViewEvents_ChangedEventHandler(viewEvents_Changed);
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (view != null)
        {
          Marshal.ReleaseComObject(view);
          view = null;
        }
        if (window != null)
        {
          Marshal.ReleaseComObject(window);
          window = null;
        }
      }
    }

    void viewEvents_Changed()
    {
    }

    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
      if (viewEvents != null)
      {
        Marshal.ReleaseComObject(viewEvents);
        viewEvents = null;
      }
      if (application != null)
      {
        Marshal.ReleaseComObject(application);
        application = null;
      }
    }
  }
}
```

Display Events

The display events allow you to watch for display events for a specific window's 3D view. You can obtain a reference to these events by attaching to the DisplayEvents property of a View object.

The interface SolidEdgeFramework. ISEhDCDisplayEvents contains the definitions that you will need to implement the events.

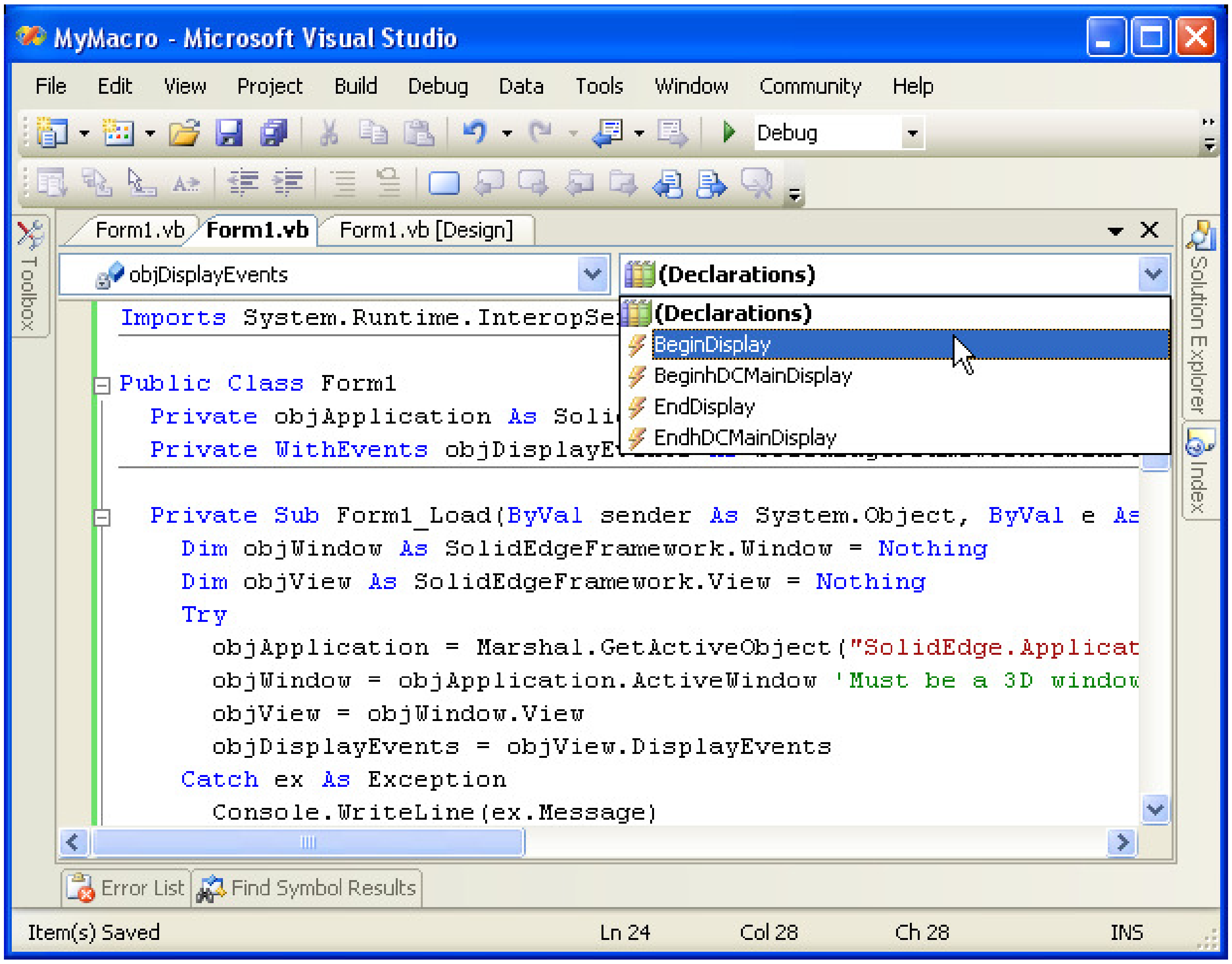
Table of Display Events

Event	Description
BeginDisplay	Occurs before Solid Edge display begins.
BeginhDCMainDisplay	Occurs before the main display to enable HDC-based clients to support underlay.
EndDisplay	Occurs after Solid Edge displays ends.
EndhDCMainDisplay	Occurs after the main display to enable HDC-based clients to support overlay.

Sinking Display Events (Visual Basic.NET)

```
Imports System.Runtime.InteropServices

Public Class Form1
  Private objApplication As SolidEdgeFramework.Application
  Private WithEvents objDisplayEvents As
SolidEdgeFramework.ISEhDCDisplayEvents_Event

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim objWindow As SolidEdgeFramework.Window = Nothing
    Dim objView As SolidEdgeFramework.View = Nothing
    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")
      ' Get a reference to the active window
      objWindow = objApplication.ActiveWindow 'Must be a 3D window

      ' Get a reference to the window's view
      objView = objWindow.View

      ' Get a reference to the view's display events
      objDisplayEvents = objView.DisplayEvents
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objView Is Nothing) Then
        Marshal.ReleaseComObject(objView)
        objView = Nothing
      End If
      If Not (objWindow Is Nothing) Then
        Marshal.ReleaseComObject(objWindow)
        objWindow = Nothing
      End If
    End Try
  End Sub

  Private Sub Form1_FormClosing(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.FormClosingEventArgs) Handles MyBase.FormClosing
    If Not (objDisplayEvents Is Nothing) Then
      Marshal.ReleaseComObject(objDisplayEvents)
      objDisplayEvents = Nothing
    End If
    If Not (objApplication Is Nothing) Then
      Marshal.ReleaseComObject(objApplication)
      objApplication = Nothing
    End If
  End Sub
End Class
```

Once you have a valid reference to the events, you can select the specific event that you wish to watch from the Declarations dropdown.


MyMacro - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Window Community Help
Debug
Form1.vb Form1.vb Form1.vb [Design]
objDisplayEvents
(Declarations)
(Declarations)
BeginDisplay
BeginhDCMainDisplay
EndDisplay
EndhDCMainDisplay
Toolbox
Solution Explorer
Index
Error List
Find Symbol Results
Item(s) Saved Ln 24 Col 28 Ch 28 INS

```
Imports System.Runtime.InteropSe

Public Class Form1
  Private objApplication As Soli
  Private WithEvents objDisplayE

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
    Dim objWindow As SolidEdgeFramework.Window = Nothing
    Dim objView As SolidEdgeFramework.View = Nothing
    Try
      objApplication = Marshal.GetActiveObject("SolidEdge.Applicat
      objWindow = objApplication.ActiveWindow 'Must be a 3D window
      objView = objWindow.View
      objDisplayEvents = objView.DisplayEvents
    Catch ex As Exception
      Console.WriteLine(ex.Message)
```

Figure 9-6 - Sinking Display Events

Sinking Display Events (C#)

```
using System;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace MyMacro
{
  public partial class Form1 : Form
  {
    private SolidEdgeFramework.Application application;
    private SolidEdgeFramework.ISEhDCDisplayEvents_Event displayEvents;

    public Form1()
    {
      InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
      SolidEdgeFramework.Window window = null;
      SolidEdgeFramework.View view = null;
      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the active window
        window = (SolidEdgeFramework.Window)
          application.ActiveWindow; //Must be a 3D window

        // Get a reference to the window's view
        view = window.View;

        // Get a reference to the view's display events
        displayEvents = (SolidEdgeFramework.ISEhDCDisplayEvents_Event)
          view.DisplayEvents;

        // Example sink of the BeginDisplay Event
        displayEvents.BeginDisplay += new
SolidEdgeFramework.ISEhDCDisplayEvents_BeginDisplayEventHandler(displayEvents
_BeginDisplay);
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (view != null)
        {
          Marshal.ReleaseComObject(view);
          view = null;
        }
        if (window != null)
        {
          Marshal.ReleaseComObject(window);
          window = null;
        }
      }
    }

    void displayEvents_BeginDisplay()
    {
    }

    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
      if (displayEvents != null)
      {
        Marshal.ReleaseComObject(displayEvents);
        displayEvents = null;
      }
      if (application != null)
      {
        Marshal.ReleaseComObject(application);
        application = null;
```

```
            }
        }
    }
}
```

GL Display Events

The GL display events allow you to watch for OpenGL related display events for a specific window's 3D view. You can obtain a reference to these events by attaching to the GLDisplayEvents property of a View object.

The interface SolidEdgeFramework. ISEIGLDisplayEvents contains the definitions that you will need to implement the events.

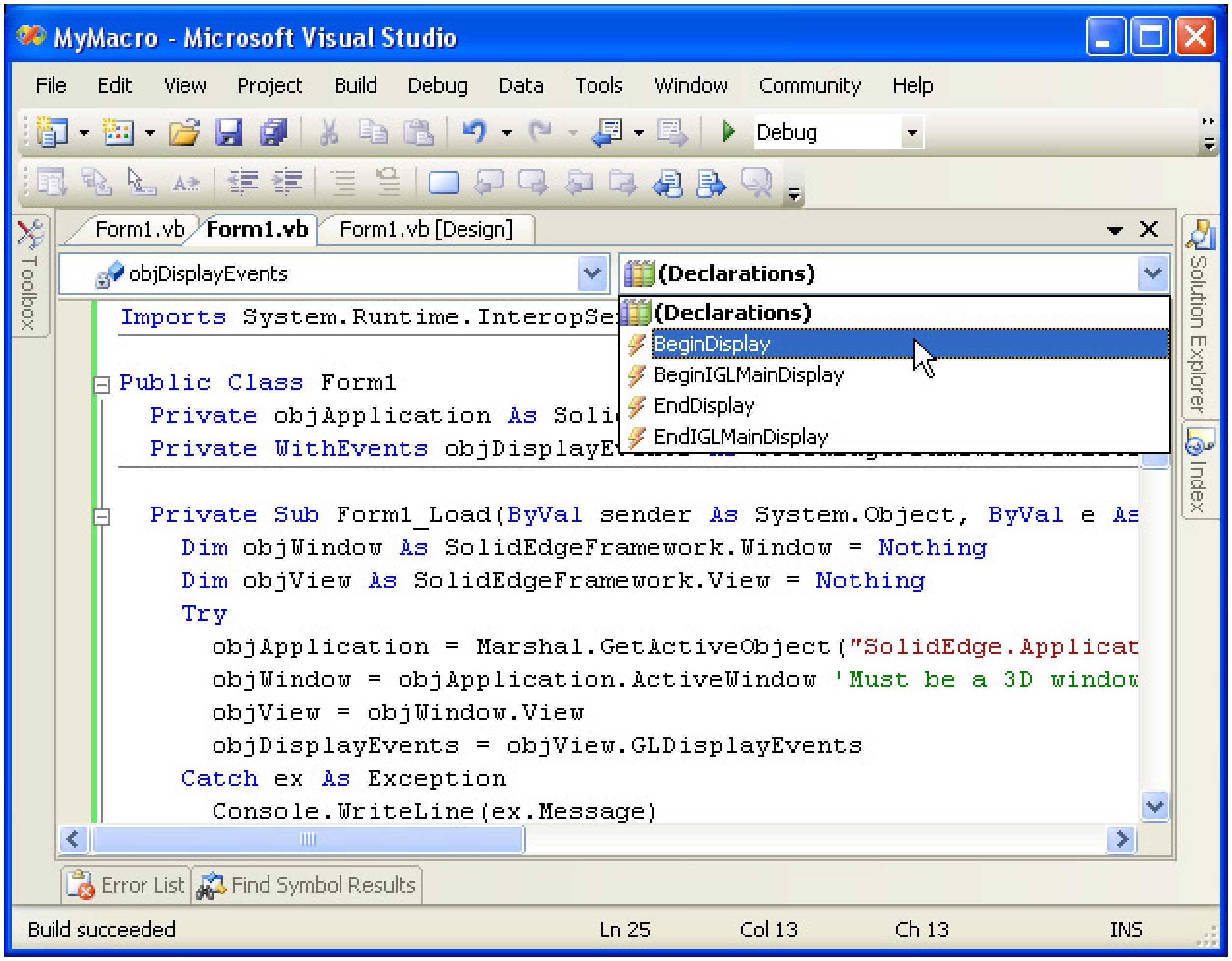
Table of GL Display Events

Event	Description
BeginDisplay	Occurs before Solid Edge display begins.
BeginIGLMainDisplay	Occurs before the main display to enable GL-based clients to support underlay.
EndDisplay	Occurs after Solid Edge displays ends.
EndIGLMainDisplay	Occurs after the main display to enable GL-based clients to support overlay.

Sinking GL Display Events (Visual Basic.NET)

```
Imports System.Runtime.InteropServices

Public Class Form1
  Private objApplication As SolidEdgeFramework.Application
  Private WithEvents objDisplayEvents As
SolidEdgeFramework.ISEIGLDisplayEvents_Event

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    Dim objWindow As SolidEdgeFramework.Window = Nothing
    Dim objView As SolidEdgeFramework.View = Nothing
    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the active window
      objWindow = objApplication.ActiveWindow 'Must be a 3D window

      ' Get a reference to the window's view
      objView = objWindow.View

      ' Get a reference to the view's display events
      objDisplayEvents = objView.GLDisplayEvents
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objView Is Nothing) Then
        Marshal.ReleaseComObject(objView)
        objView = Nothing
      End If
      If Not (objWindow Is Nothing) Then
        Marshal.ReleaseComObject(objWindow)
        objWindow = Nothing
      End If
    End Try
  End Sub

  Private Sub Form1_FormClosing(ByVal sender As System.Object, ByVal e As
System.Windows.Forms.FormClosingEventArgs) Handles MyBase.FormClosing
    If Not (objDisplayEvents Is Nothing) Then
      Marshal.ReleaseComObject(objDisplayEvents)
      objDisplayEvents = Nothing
    End If
    If Not (objApplication Is Nothing) Then
      Marshal.ReleaseComObject(objApplication)
      objApplication = Nothing
    End If
  End Sub
End Class
```

Once you have a valid reference to the events, you can select the specific event that you wish to watch from the Declarations dropdown.



```
Imports System.Runtime.InteropSe

Public Class Form1
  Private objApplication As Soli
  Private WithEvents objDisplayE

  Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
    Dim objWindow As SolidEdgeFramework.Window = Nothing
    Dim objView As SolidEdgeFramework.View = Nothing
    Try
      objApplication = Marshal.GetActiveObject("SolidEdge.Applicat
      objWindow = objApplication.ActiveWindow 'Must be a 3D window
      objView = objWindow.View
      objDisplayEvents = objView.GLDisplayEvents
    Catch ex As Exception
      Console.WriteLine(ex.Message)
```

9-7 - Sinking GL Display Events

Sinking GL Display Events (C#)

```
using System;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace MyMacro
{
  public partial class Form1 : Form
  {
    private SolidEdgeFramework.Application application;
    private SolidEdgeFramework.ISEIGLDisplayEvents_Event displayEvents;

    public Form1()
    {
      InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
      SolidEdgeFramework.Window window = null;
      SolidEdgeFramework.View view = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the acrtive window
        window = (SolidEdgeFramework.Window)
          application.ActiveWindow; //Must be a 3D window

        // Get a reference to the window's view
        view = window.View;

        // Get a reference to the view's display events
        displayEvents = (SolidEdgeFramework.ISEIGLDisplayEvents_Event)
          view.GLDisplayEvents;

        // Example sink of the BeginDisplay event
        displayEvents.BeginDisplay += new
SolidEdgeFramework.ISEIGLDisplayEvents_BeginDisplayEventHandler(displayEvents
_BeginDisplay);
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (view != null)
        {
          Marshal.ReleaseComObject(view);
          view = null;
        }
        if (window != null)
        {
          Marshal.ReleaseComObject(window);
          window = null;
        }
      }
    }

    void displayEvents_BeginDisplay()
    {
    }

    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
      if (displayEvents != null)
      {
        Marshal.ReleaseComObject(displayEvents);
        displayEvents = null;
      }
      if (application != null)
      {
        Marshal.ReleaseComObject(application);
        application = null;
      }
    }
  }
}
```

Chapter 10 - File 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.

Although each Solid Edge file type is different, the following table lists common file properties:

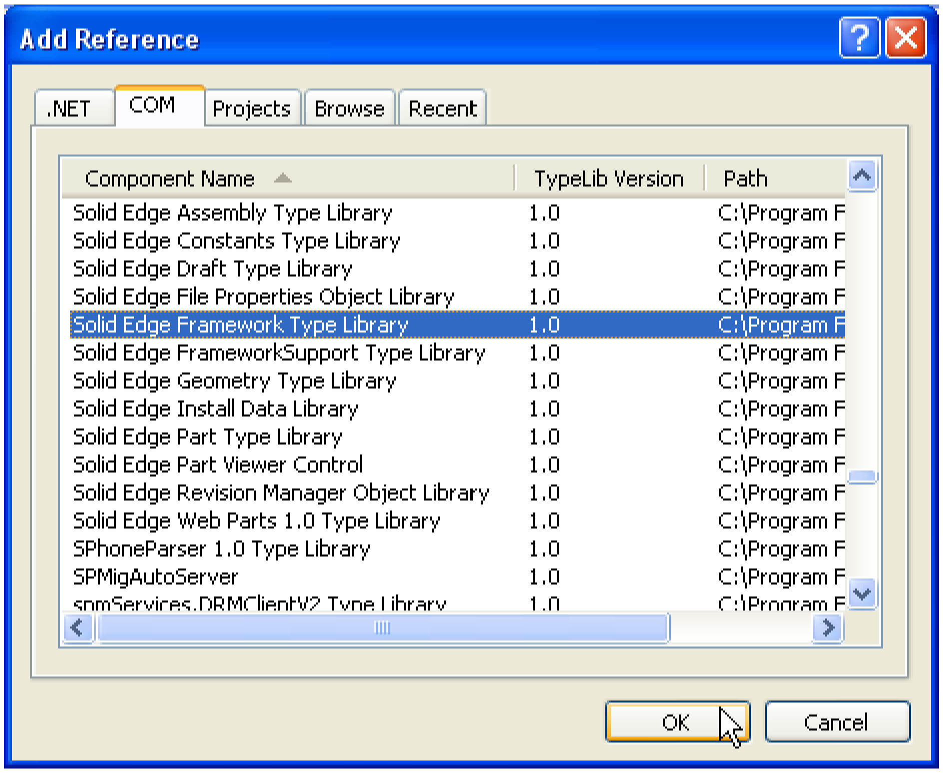
Table of File Properties

Property Set	Property Name	Maintained	Variant Type	.NET Type
Custom	<User Defined>	X	Any	Any
DocumentSummaryInformation	Bye Count		VT_I4	System.Int32
DocumentSummaryInformation	Category	X	VT_LPSTR	System.String
DocumentSummaryInformation	Company	X	VT_LPSTR	System.String
DocumentSummaryInformation	Hidden Objects		VT_I4	System.Int32
DocumentSummaryInformation	Lines		VT_I4	System.Int32
DocumentSummaryInformation	Manager	X	VT_LPSTR	System.String
DocumentSummaryInformation	Multimedia Clips		VT_I4	System.Int32
DocumentSummaryInformation	Notes		VT_I4	System.Int32
DocumentSummaryInformation	Paragraphs		VT_I4	System.Int32
DocumentSummaryInformation	Presentation Format		VT_LPSTR	System.String
DocumentSummaryInformation	Slides		VT_I4	System.Int32
ExtendedSummaryInformation	CreationLocale	X	VT_I4	System.Int32
ExtendedSummaryInformation	DocumentID	X	VT_CLSID	System.Guid
ExtendedSummaryInformation	Hardware	X	VT_BOOL	System.Boolean
ExtendedSummaryInformation	Name of Saving Application	X	VT_LPWSTR	System.String
ExtendedSummaryInformation	Status	X	VT_I4	System.Int32
ExtendedSummaryInformation	Username	X	VT_LPWSTR	System.String
MechanicalModeling	Material	X	VT_LPWSTR	System.String
ProjectInformation	Document Number	X	VT_LPWSTR	System.String
ProjectInformation	Project Name	X	VT_LPWSTR	System.String
ProjectInformation	Revision	X	VT_LPWSTR	System.String
SummaryInformation	Application Name	X	VT_LPSTR	System.String
SummaryInformation	Author	X	VT_LPSTR	System.String
SummaryInformation	Comments	X	VT_LPSTR	System.String
SummaryInformation	Creation Date	X	VT_FILETIME	System.DateTime
SummaryInformation	Keywords	X	VT_LPSTR	System.String
SummaryInformation	Last Author	X	VT_LPSTR	System.String
SummaryInformation	Last Print Date		VT_FILETIME	System.DateTime
SummaryInformation	Last Save Date	X	VT_FILETIME	System.DateTime
SummaryInformation	Number of characters		VT_I4	System.Int32

SummaryInformation	Number of pages		VT_I4	System.Int32
SummaryInformation	Number of words		VT_I4	System.Int32
SummaryInformation	Origination Date		VT_FILETIME	System.DateTime
SummaryInformation	Revision Number		VT_LPSTR	System.String
SummaryInformation	Security		VT_I4	System.Int32
SummaryInformation	Subject	X	VT_LPSTR	System.String
SummaryInformation	Template	X	VT_LPSTR	System.String
SummaryInformation	Title	X	VT_LPSTR	System.String
SummaryInformation	Total Editing Time		VT_FILETIME	System.DateTime

SolidEdgeFramework API

You can access the file properties of documents that are open in Solid Edge via the SolidEdgeFramework API. You will need to add a reference to the Solid Edge Framework Type Library to access this API.



Component Name	TypeLib Version	Path
Solid Edge Assembly Type Library	1.0	C:\Program F
Solid Edge Constants Type Library	1.0	C:\Program F
Solid Edge Draft Type Library	1.0	C:\Program F
Solid Edge File Properties Object Library	1.0	C:\Program F
Solid Edge Framework Type Library	1.0	C:\Program F
Solid Edge FrameworkSupport Type Library	1.0	C:\Program F
Solid Edge Geometry Type Library	1.0	C:\Program F
Solid Edge Install Data Library	1.0	C:\Program F
Solid Edge Part Type Library	1.0	C:\Program F
Solid Edge Part Viewer Control	1.0	C:\Program F
Solid Edge Revision Manager Object Library	1.0	C:\Program F
Solid Edge Web Parts 1.0 Type Library	1.0	C:\Program F
SPhoneParser 1.0 Type Library	1.0	C:\Program F
SPMigAutoServer	1.0	C:\Program F
spmServices.DRMClientV2 Type Library	1.0	C:\Program F

10-1 - Add Reference

The following console programs demonstrate how to enumerate all available file properties for a given document using the SolidEdgeFramework API.

Reading File Properties (Visual Basic.NET)

```
Imports System.Runtime.InteropServices
Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objDocument As SolidEdgeFramework.SolidEdgeDocument = Nothing
    Dim objPropertySets As SolidEdgeFramework.PropertySets = Nothing
    Dim objProperties As SolidEdgeFramework.Properties = Nothing
    Dim objProperty As SolidEdgeFramework.Property = Nothing
    Dim i As Integer
    Dim j As Integer
    Dim vt As VarEnum
    Dim strFormat1 As String = "[{0}]"
    Dim strFormat2 As String = "{0} = {1} ({2})"

    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")
      ' Get a reference to the active document
      objDocument = objApplication.ActiveDocument

      ' Get a reference to the document's property sets collection
      objPropertySets = objDocument.Properties

      ' Example: Loop through all properties
      ' Note that indexes are one based
      For i = 1 To objPropertySets.Count
        objProperties = objPropertySets.Item(i)
        Console.WriteLine(String.Format(strFormat1, objProperties.Name))
        ' Note that indexes are one based
        For j = 1 To objProperties.Count
          objProperty = objProperties.Item(j)
          vt = objProperty.Type
          Console.WriteLine(String.Format(strFormat2, _
            objProperty.Name, objProperty.Value, vt))
        Next
        Console.WriteLine()
      Next

      ' Example: Read property by name
      objProperties = objPropertySets.Item("SummaryInformation")
      objProperty = objProperties.Item("Title")

    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objProperty Is Nothing) Then
        Marshal.ReleaseComObject(objProperty)
        objProperty = Nothing
      End If
      If Not (objProperties Is Nothing) Then
        Marshal.ReleaseComObject(objProperties)
        objProperties = Nothing
      End If
      If Not (objPropertySets Is Nothing) Then
        Marshal.ReleaseComObject(objPropertySets)
        objPropertySets = Nothing
      End If
      If Not (objDocument Is Nothing) Then
        Marshal.ReleaseComObject(objDocument)
        objDocument = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Reading File Properties (C#)

```
using System;
using System.Runtime.InteropServices;
```

```
namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.SolidEdgeDocument document = null;
      SolidEdgeFramework.PropertySets propertySets = null;
      SolidEdgeFramework.Properties properties = null;
      SolidEdgeFramework.Property property = null;
      string format1 = "[{0}]";
      string format2 = "{0} = {1} ({2})";
      VarEnum vt;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");

        // Get a reference to the active document
        document = (SolidEdgeFramework.SolidEdgeDocument)
          application.ActiveDocument;

        // Get a reference to the document's property sets collection
        propertySets = (SolidEdgeFramework.PropertySets)document.Properties;

        // Example: Loop through all properties
        // Note that indexes are one based
        for (int i = 1; i <= propertySets.Count; i++)
        {
          properties = propertySets.Item(i);
          Console.WriteLine(string.Format(format1, properties.Name));
          // Note that indexes are one based
          for (int j = 1; j <= properties.Count; j++)
          {
            property = properties.Item(j);
            vt = (VarEnum)property.Type;

            Console.WriteLine(
              string.Format(
                format2, property.Name, property.get_Value(), vt));
          }
          Console.WriteLine();
        }

        // Example: Read property by name
        properties = propertySets.Item("SummaryInformation");
        property = properties.Item("Title");
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (property != null)
        {
          Marshal.ReleaseComObject(property);
          property = null;
        }
        if (properties != null)
        {
          Marshal.ReleaseComObject(properties);
          properties = null;
        }
        if (propertySets != null)
        {
          Marshal.ReleaseComObject(propertySets);
          propertySets = null;
        }
        if (document != null)
        {
          Marshal.ReleaseComObject(document);
          document = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

The following console programs demonstrate how to add your own custom properties to a given document using the SolidEdgeFramework API.

Working with Custom Properties (Visual Basic.NET)

```
Imports System.Runtime.InteropServices
Module Module1
  Sub Main()
    Dim objApplication As SolidEdgeFramework.Application = Nothing
    Dim objDocument As SolidEdgeFramework.SolidEdgeDocument = Nothing
    Dim objPropertySets As SolidEdgeFramework.PropertySets = Nothing
    Dim objProperties As SolidEdgeFramework.Properties = Nothing
    Dim objProperty As SolidEdgeFramework.Property = Nothing

    Try
      ' Connect to a running instance of Solid Edge
      objApplication = Marshal.GetActiveObject("SolidEdge.Application")

      ' Get a reference to the active document
      objDocument = objApplication.ActiveDocument

      ' Get a reference to the document's property set collection
      objPropertySets = objDocument.Properties

      ' Get a reference to the Custom properties
      objProperties = objPropertySets.Item("Custom")

      ' Add custom properties
      objProperty = objProperties.Add("My String", "Hello")
      objProperty = objProperties.Add("My Integer", 123)
      objProperty = objProperties.Add("My Date", DateTime.Now)
      objProperty = objProperties.Add("My Boolean", True)
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objProperty Is Nothing) Then
        Marshal.ReleaseComObject(objProperty)
        objProperty = Nothing
      End If
      If Not (objProperties Is Nothing) Then
        Marshal.ReleaseComObject(objProperties)
        objProperties = Nothing
      End If
      If Not (objPropertySets Is Nothing) Then
        Marshal.ReleaseComObject(objPropertySets)
        objPropertySets = Nothing
      End If
      If Not (objDocument Is Nothing) Then
        Marshal.ReleaseComObject(objDocument)
        objDocument = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

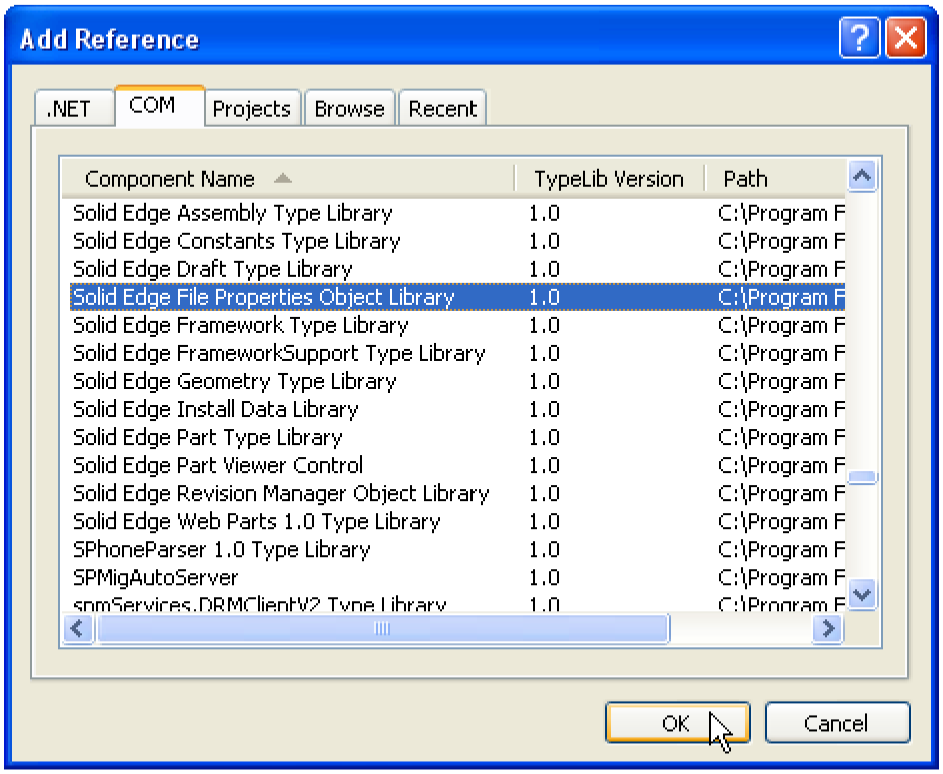
Working with Custom Properties (C#)

```
using System;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFramework.Application application = null;
      SolidEdgeFramework.SolidEdgeDocument document = null;
      SolidEdgeFramework.PropertySets propertySets = null;
      SolidEdgeFramework.Properties properties = null;
      SolidEdgeFramework.Property property = null;

      try
      {
        // Connect to a running instance of Solid Edge
        application = (SolidEdgeFramework.Application)
          Marshal.GetActiveObject("SolidEdge.Application");
        // Get a reference to the active document
        document = (SolidEdgeFramework.SolidEdgeDocument)
          application.ActiveDocument;

        // Get a reference to the document's property set collection
        propertySets = (SolidEdgeFramework.PropertySets)document.Properties;

        // Get a reference to the Custom properties
        properties = propertySets.Item("Custom");

        // Add custom properties
        property = properties.Add("My String", "Hello");
        property = properties.Add("My Integer", 123);
        property = properties.Add("My Date", DateTime.Now);
        property = properties.Add("My Boolean", true);
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (property != null)
        {
          Marshal.ReleaseComObject(property);
          property = null;
        }
        if (properties != null)
        {
          Marshal.ReleaseComObject(properties);
          properties = null;
        }
        if (propertySets != null)
        {
          Marshal.ReleaseComObject(propertySets);
          propertySets = null;
        }
        if (document != null)
        {
          Marshal.ReleaseComObject(document);
          document = null;
        }
        if (application != null)
        {
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

SolidEdgeFileProperties API

The SolidEdgeFileProperties API is a lightweight API used when you need to access file properties of documents outside of Solid Edge.

You will need to add a reference to the Solid Edge File Properties Object Library to access this API.



Component Name	TypeLib Version	Path
Solid Edge Assembly Type Library	1.0	C:\Program F
Solid Edge Constants Type Library	1.0	C:\Program F
Solid Edge Draft Type Library	1.0	C:\Program F
Solid Edge File Properties Object Library	1.0	C:\Program F
Solid Edge Framework Type Library	1.0	C:\Program F
Solid Edge FrameworkSupport Type Library	1.0	C:\Program F
Solid Edge Geometry Type Library	1.0	C:\Program F
Solid Edge Install Data Library	1.0	C:\Program F
Solid Edge Part Type Library	1.0	C:\Program F
Solid Edge Part Viewer Control	1.0	C:\Program F
Solid Edge Revision Manager Object Library	1.0	C:\Program F
Solid Edge Web Parts 1.0 Type Library	1.0	C:\Program F
SPhoneParser 1.0 Type Library	1.0	C:\Program F
SPMigAutoServer	1.0	C:\Program F
spmServices.DRMClientV2 Type Library	1.0	C:\Program F

10-2 - Add Reference

The following console programs demonstrate how to enumerate all available file properties for a given document using the SolidEdgeFileProperties API.

Reading File Properties (Visual Basic.NET)

```
Imports System.Runtime.InteropServices
Module Module1
  Sub Main()
    Dim objPropertySets As SolidEdgeFileProperties.PropertySets = Nothing
    Dim objProperties As SolidEdgeFileProperties.Properties = Nothing
    Dim objProperty As SolidEdgeFileProperties.Property = Nothing
    Dim i As Integer
    Dim j As Integer
    Dim strFormat1 As String = "[{0}]"
    Dim strFormat2 As String = "{0} = {1}"

    Try
      ' Create new instance of the PropertySets object
      objPropertySets = New SolidEdgeFileProperties.PropertySets

      ' Open a file
      objPropertySets.Open("C:\Part1.par", True)

      ' Example: Loop through all properties
      ' Note that indexes are zero based
      For i = 0 To objPropertySets.Count - 1
        objProperties = objPropertySets.Item(i)
        Console.WriteLine(String.Format(strFormat1, objProperties.Name))
        ' Note that indexes are zero based
        For j = 0 To objProperties.Count - 1
          objProperty = objProperties.Item(j)
          ' .Value property may throw an exception
          Try
            Console.WriteLine(String.Format(strFormat2, _
              objProperty.Name, objProperty.Value))
          Catch ex As Exception
            Console.WriteLine(String.Format(strFormat2, _
              objProperty.Name, "(ERROR)"))
          End Try
        Next
        Console.WriteLine()
      Next

      ' Get a reference to the SummaryInformation properties
      objProperties = objPropertySets.Item("SummaryInformation")

      ' Get a reference to the Title property by name
      objProperty = objProperties.Item("Title")
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objProperty Is Nothing) Then
        Marshal.ReleaseComObject(objProperty)
        objProperty = Nothing
      End If
      If Not (objProperties Is Nothing) Then
        Marshal.ReleaseComObject(objProperties)
        objProperties = Nothing
      End If
      If Not (objPropertySets Is Nothing) Then
        ' Close underlying property storage
        objPropertySets.Close()
        Marshal.ReleaseComObject(objPropertySets)
        objPropertySets = Nothing
      End If
    End Try
  End Sub
End Module
```

Reading File Properties (C#)

```
using System;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFileProperties.PropertySets propertySets = null;
      SolidEdgeFileProperties.Properties properties = null;
      SolidEdgeFileProperties.Property property = null;
      string format1 = "[{0}]";
      string format2 = "{0} = {1}";

      try
      {
        // Create new instance of the PropertySets object
        propertySets = new SolidEdgeFileProperties.PropertySets();

        // Open a file
        propertySets.Open(@"C:\Part1.par", true);

        // Example: Loop through all properties
        // Note that indexes are zero based
        for (int i = 0; i < propertySets.Count; i++)
        {
          properties = (SolidEdgeFileProperties.Properties)propertySets[i];
          // Note that indexes are zero based
          for (int j = 0; j < properties.Count; j++)
          {
            property = (SolidEdgeFileProperties.Property)properties[j];
            try
            {
              Console.WriteLine(
                              string.Format(
                                format2, property.Name, property.Value));
            }
            catch
            {
              Console.WriteLine(
                              string.Format(
                                format2, property.Name, "ERROR"));
            }
          }
        }

        // Get a reference to the SummaryInformation properties
        properties = (SolidEdgeFileProperties.Properties)
          propertySets["SummaryInformation"];

        // Get a reference to the Title property by name
        property = (SolidEdgeFileProperties.Property)
          properties["Title"];
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (property != null)
        {
          Marshal.ReleaseComObject(property);
          property = null;
        }
        if (properties != null)
        {
          Marshal.ReleaseComObject(properties);
          properties = null;
        }
        if (propertySets != null)
        {
          propertySets.Close();
          Marshal.ReleaseComObject(propertySets);
          propertySets = null;
        }
      }
    }
  }
}
```

The following console programs demonstrate how to add your own custom properties to a given document using the SolidEdgeFileProperties API.

Working with Custom Properties (Visual Basic.NET)

```
Imports System.Runtime.InteropServices
Module Module1
  Sub Main()
    Dim objPropertySets As SolidEdgeFileProperties.PropertySets = Nothing
    Dim objProperties As SolidEdgeFileProperties.Properties = Nothing
    Dim objProperty As SolidEdgeFileProperties.Property = Nothing

    Try
      ' Create new instance of the PropertySets object
      objPropertySets = New SolidEdgeFileProperties.PropertySets

      ' Open a file
      objPropertySets.Open("C:\Part1.par", False)

      ' Get reference to Custom Property Set
      objProperties = objPropertySets.Item("Custom")

      ' Add custom file properties
      objProperty = objProperties.Add("My String", "Hello")
      objProperty = objProperties.Add("My Integer", 123)
      objProperty = objProperties.Add("My Date", DateTime.Now)
      objProperty = objProperties.Add("My Boolean", True)

      ' Delete last custom property
      objProperty.Delete()

      ' Save changes
      objPropertySets.Save()
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
```

```
      If Not (objProperty Is Nothing) Then
        Marshal.ReleaseComObject(objProperty)
        objProperty = Nothing
      End If
      If Not (objProperties Is Nothing) Then
        Marshal.ReleaseComObject(objProperties)
        objProperties = Nothing
      End If
      If Not (objPropertySets Is Nothing) Then
        ' Close underlying property storage
        objPropertySets.Close()
        Marshal.ReleaseComObject(objPropertySets)
        objPropertySets = Nothing
      End If
    End Try
  End Sub
End Module
```

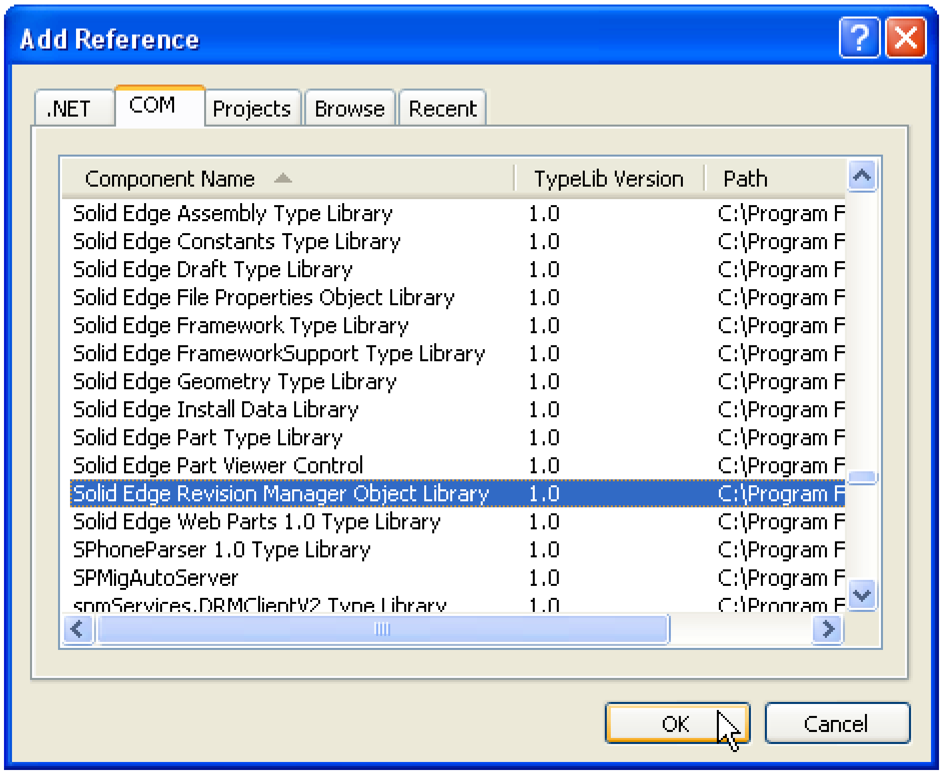
Working with Custom Properties (C#)

```
using System;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SolidEdgeFileProperties.PropertySets propertySets = null;
      SolidEdgeFileProperties.Properties properties = null;
      SolidEdgeFileProperties.Property property = null;

      try
      {
        // Create new instance of the PropertySets object
        propertySets = new SolidEdgeFileProperties.PropertySets();

        // Open a file
        propertySets.Open(@"C:\Part1.par", false);

        // Get reference to Custom Property Set
        properties = (SolidEdgeFileProperties.Properties)
          propertySets["Custom"];

        // Add custom file properties
        property = (SolidEdgeFileProperties.Property)
          properties.Add("My String", "Hello");
        property = (SolidEdgeFileProperties.Property)
          properties.Add("My Integer", 123);
        property = (SolidEdgeFileProperties.Property)
          properties.Add("My Date", DateTime.Now);
        property = (SolidEdgeFileProperties.Property)
          properties.Add("My Boolean", true);

        // Delete last custom property
        property.Delete();
        // Save changes
        propertySets.Save();
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (property != null)
        {
          Marshal.ReleaseComObject(property);
          property = null;
        }
        if (properties != null)
        {
          Marshal.ReleaseComObject(properties);
          properties = null;
        }
        if (propertySets != null)
        {
          Marshal.ReleaseComObject(propertySets);
          propertySets = null;
        }
      }
    }
  }
}
```

RevisionManager API

The RevisionManager API is another lightweight API used when you need to access file properties of documents outside of Solid Edge.

You will need to add a reference to the Solid Edge Revision Manager Object Library to access this API.


Add Reference

.NET | COM | Projects | Browse | Recent

Component Name	TypeLib Version	Path
Solid Edge Assembly Type Library	1.0	C:\Program F
Solid Edge Constants Type Library	1.0	C:\Program F
Solid Edge Draft Type Library	1.0	C:\Program F
Solid Edge File Properties Object Library	1.0	C:\Program F
Solid Edge Framework Type Library	1.0	C:\Program F
Solid Edge FrameworkSupport Type Library	1.0	C:\Program F
Solid Edge Geometry Type Library	1.0	C:\Program F
Solid Edge Install Data Library	1.0	C:\Program F
Solid Edge Part Type Library	1.0	C:\Program F
Solid Edge Part Viewer Control	1.0	C:\Program F
Solid Edge Revision Manager Object Library	1.0	C:\Program F
Solid Edge Web Parts 1.0 Type Library	1.0	C:\Program F
SPhoneParser 1.0 Type Library	1.0	C:\Program F
SPMigAutoServer	1.0	C:\Program F
spmServices.DRMClientV2 Type Library	1.0	C:\Program F

10-3 - Add Reference

The following console programs demonstrate how to enumerate all available file properties for a given document using the RevisionManager API.

Reading File Properties (Visual Basic.NET)

```
Imports System.Runtime.InteropServices
Module Module1
  Sub Main()
    Dim objApplication As RevisionManager.Application = Nothing
    Dim objPropertySets As RevisionManager.PropertySets = Nothing
    Dim objProperties As RevisionManager.Properties = Nothing
    Dim objProperty As RevisionManager.Property = Nothing
    Dim i As Integer
    Dim j As Integer
    Dim strFormat1 As String = "[{0}]"
    Dim strFormat2 As String = "{0} = {1}"

    Try
      ' Create new instance of the Revision Manager Application object
      objApplication = New RevisionManager.Application

      ' Open a file
      objApplication.Open("C:\Part1.par")

      ' Get a reference to the property sets collection
      objPropertySets = objApplication.PropertySets

      ' Example: Loop through all properties.
      ' Note that indexes are zero based
      For i = 0 To objPropertySets.Count - 1
        objProperties = objPropertySets.Item(i)
        Console.WriteLine(String.Format(strFormat1, objProperties.Name))
        ' Note that indexes are zero based
        For j = 0 To objProperties.Count - 1
          objProperty = objProperties.Item(j)
          ' .Value property may throw an exception
          Try
            Console.WriteLine(String.Format(strFormat2, _
              objProperty.Name, objProperty.Value))
          Catch ex As Exception
            Console.WriteLine(String.Format(strFormat2, _
              objProperty.Name, "(ERROR)"))
          End Try
        Next
        Console.WriteLine()
      Next

      ' Get a reference to the SummaryInformation properties
      objProperties = objPropertySets.Item("SummaryInformation")

      ' Get a reference to the Title property by name
      objProperty = objProperties.Item("Title")

    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objProperty Is Nothing) Then
        Marshal.ReleaseComObject(objProperty)
        objProperty = Nothing
      End If
      If Not (objProperties Is Nothing) Then
        Marshal.ReleaseComObject(objProperties)
        objProperties = Nothing
      End If
      If Not (objPropertySets Is Nothing) Then
        Marshal.ReleaseComObject(objPropertySets)
        objPropertySets = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        objApplication.Quit()
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If
    End Try
  End Sub
End Module
```

Reading File Properties (C#)

```
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyMacro
{
```

```
class Program
{
  static void Main(string[] args)
  {
    RevisionManager.Application application = null;
    RevisionManager.PropertySets propertySets = null;
    RevisionManager.Properties properties = null;
    RevisionManager.Property property = null;
    string strFormat1 = "[{0}]";
    string strFormat2 = "{0} = {1}";

    try
    {
      // Create new instance of the Revision Manager Application object
      application = new RevisionManager.Application();

      // Open a file
      propertySets.Open(@"C:\Part1.par", true);

      // Get a reference to the property sets collection
      propertySets = (RevisionManager.PropertySets)
        application.PropertySets;

      // Example: Loop through all properties.
      // Note that indexes are zero based
      for (int i = 0; i < propertySets.Count; i++)
      {
        properties = (RevisionManager.Properties)propertySets.get_Item(i);
        Console.WriteLine(String.Format(strFormat1, properties.Name));
        // Note that indexes are zero based
        for (int j = 0; j < properties.Count; j++)
        {
          property = (RevisionManager.Property)
            properties.get_Item(j);
          try
          {
            Console.WriteLine(
              String.Format(strFormat2, property.Name, property.Value));
          }
          catch
          {
            Console.WriteLine(
              String.Format(strFormat2, property.Name, "(ERROR)"));
          }
        }
      }

      // Get a reference to the SummaryInformation properties
      properties = (RevisionManager.Properties)
        propertySets.get_Item("SummaryInformation");

      // Get a reference to the Title property by name
      property = (RevisionManager.Property)properties.get_Item("Title");
    }
    catch (System.Exception ex)
    {
      Console.WriteLine(ex.Message);
      }
      finally
      {
        if (property != null)
        {
          Marshal.ReleaseComObject(property);
          property = null;
        }
        if (properties != null)
        {
          Marshal.ReleaseComObject(properties);
          properties = null;
        }
        if (propertySets != null)
        {
          Marshal.ReleaseComObject(propertySets);
          propertySets = null;
        }
        if (application != null)
        {
          application.Quit();
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

The following console programs demonstrate how to add your own custom properties to a given document using the RevisionManager API.

Working with Custom Properties (Visual Basic.NET)

```
Imports System.Runtime.InteropServices
Module Module1
  Sub Main()
    Dim objApplication As RevisionManager.Application = Nothing
    Dim objPropertySets As RevisionManager.PropertySets = Nothing
    Dim objProperties As RevisionManager.Properties = Nothing

    Try
      ' Create new instance of the Revision Manager Application object
      objApplication = New RevisionManager.Application

      ' Open a file
      objPropertySets.Open("C:\Part1.par", False)

      ' Get a reference to the property sets collection
      objPropertySets = objApplication.PropertySets

      ' Get reference to Custom property set
      objProperties = objPropertySets.Item("Custom")

      ' Add custom file properties
      objProperties.Add("My String", "Hello")
      objProperties.Add("My Integer", 123)
      objProperties.Add("My Date", DateTime.Now)
      objProperties.Add("My Boolean", True)

      ' Save changes
      objPropertySets.Save()
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objProperties Is Nothing) Then
        Marshal.ReleaseComObject(objProperties)
        objProperties = Nothing
      End If
      If Not (objPropertySets Is Nothing)
        Marshal.ReleaseComObject(objPropertySets)
        objPropertySets = Nothing
      End If
      If Not (objApplication Is Nothing) Then
        objApplication.Quit()
        Marshal.ReleaseComObject(objApplication)
        objApplication = Nothing
      End If

    End Try
  End Sub
End Module
```

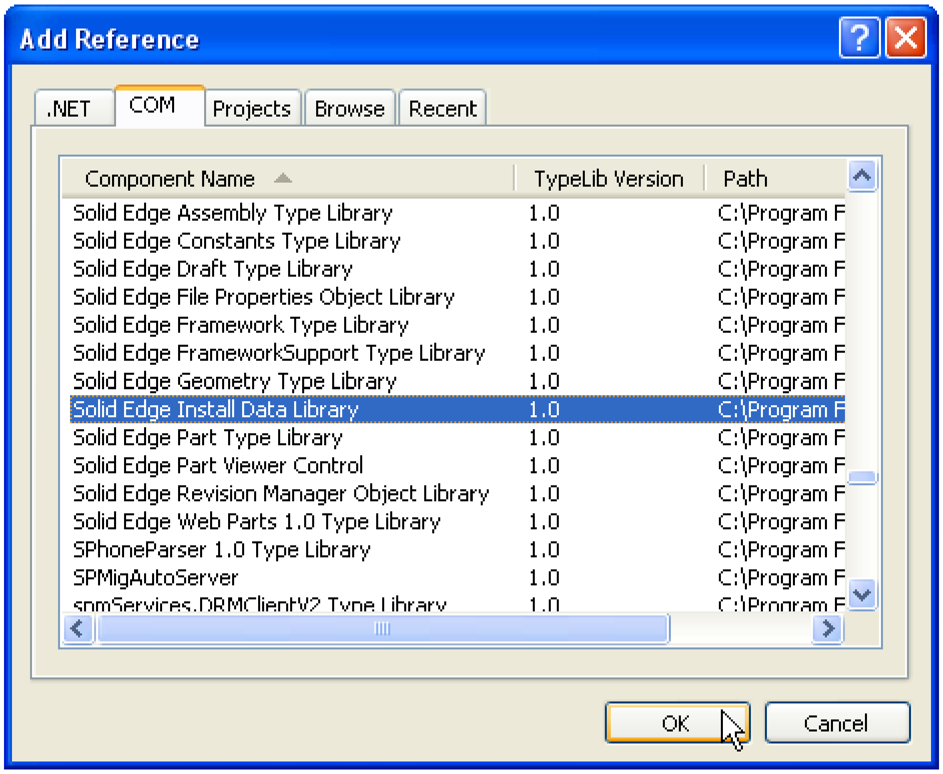
Working with Custom Properties (C#)

```
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      RevisionManager.Application application = null;
      RevisionManager.PropertySets propertySets = null;
      RevisionManager.Properties properties = null;

      try
      {
        // Create new instance of the Revision Manager Application object
        application = new RevisionManager.Application();

        // Open a file
        propertySets.Open(@"C:\Part1.par", false);

        // Get a reference to the property sets collection
        propertySets = (RevisionManager.PropertySets)
          application.PropertySets;

        // Get reference to Custom property set
        properties = (RevisionManager.Properties)
          propertySets.get_Item("Custom");

        // Add custom file properties
        properties.Add("My String", "Hello");
        properties.Add("My Integer", 123);
        properties.Add("My Date", DateTime.Now);
        properties.Add("My Boolean", true);

        // Save changes
        propertySets.Save();
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (properties != null)
        {
          Marshal.ReleaseComObject(properties);
          properties = null;
        }
        if (propertySets != null)
        {
          Marshal.ReleaseComObject(propertySets);
          propertySets = null;
        }
        if (application != null)
        {
          application.Quit();
          Marshal.ReleaseComObject(application);
          application = null;
        }
      }
    }
  }
}
```

Chapter 11 - Installation Information

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.

You will need to add a reference to the Solid Edge Install Data Library to access this class.



Component Name	TypeLib Version	Path
Solid Edge Assembly Type Library	1.0	C:\Program F
Solid Edge Constants Type Library	1.0	C:\Program F
Solid Edge Draft Type Library	1.0	C:\Program F
Solid Edge File Properties Object Library	1.0	C:\Program F
Solid Edge Framework Type Library	1.0	C:\Program F
Solid Edge FrameworkSupport Type Library	1.0	C:\Program F
Solid Edge Geometry Type Library	1.0	C:\Program F
Solid Edge Install Data Library	1.0	C:\Program F
Solid Edge Part Type Library	1.0	C:\Program F
Solid Edge Part Viewer Control	1.0	C:\Program F
Solid Edge Revision Manager Object Library	1.0	C:\Program F
Solid Edge Web Parts 1.0 Type Library	1.0	C:\Program F
SPhoneParser 1.0 Type Library	1.0	C:\Program F
SPMigAutoServer	1.0	C:\Program F
spmServices.DRMClientV2 Type Library	1.0	C:\Program F

11-1 - Add Reference

The following console programs demonstrate how to create an instance of the SEInstallData class. 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.

Extracting Installation Information (Visual Basic .NET)

```
Imports System.Globalization
Imports System.IO
Imports System.Runtime.InteropServices

Module Module1

  Sub Main()
    Dim objInstallData As SEInstallDataLib.SEInstallData = Nothing
    Dim iBuildNumber As Integer
    Dim objInstallFolder As DirectoryInfo
    Dim objCulture As CultureInfo
    Dim iMajorVersion As Integer
    Dim iMinorVersion As Integer
    Dim iParasolidMajorVersion As Integer
    Dim iParasolidMinorVersion As Integer
    Dim objParasolidVersion As System.Version
    Dim iServicePackVersion As Integer
    Dim objVersion As System.Version

    Try
      ' Create a new instance of the SEInstallData object
      objInstallData = New SEInstallDataLib.SEInstallData()

      ' Get the build number
      iBuildNumber = objInstallData.GetBuildNumber()

      ' Create a new instance of DirectoryInfo using GetInstalledPath()
      objInstallFolder = New DirectoryInfo(objInstallData.GetInstalledPath())

      ' Create a new instance of CultureInfo using GetLanguageID()
      objCulture = New CultureInfo(objInstallData.GetLanguageID())

      ' Get major version
      iMajorVersion = objInstallData.GetMajorVersion()

      ' Get minor version
      iMinorVersion = objInstallData.GetMinorVersion()

      ' Get parasolid major version
      iParasolidMajorVersion = objInstallData.GetParasolidMajorVersion()

      ' Get parasolid minor version
      iParasolidMinorVersion = objInstallData.GetParasolidMinorVersion()

      ' Create a new instance of Version using GetParasolidVersion()
      objParasolidVersion = New Version(objInstallData.GetParasolidVersion())

      ' Get service pack version
      iServicePackVersion = objInstallData.GetServicePackVersion()

      ' Create a new instance of Version using GetVersion()
      objVersion = New Version(objInstallData.GetVersion())

      ' Write information to screen
      Console.WriteLine(String.Format( _
        "Parasolid Version - {0}", objParasolidVersion.ToString()))
      Console.WriteLine(String.Format( _
        "Solid Edge Version - {0}", objVersion.ToString()))
      Console.WriteLine(String.Format( _
        "Install Path - {0}", objInstallFolder.FullName))
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    Finally
      If Not (objInstallData Is Nothing) Then
        Marshal.ReleaseComObject(objInstallData)
        objInstallData = Nothing
      End If
    End Try
  End Sub
End Module
```

Extracting Installation Information (C#)

```
using System;
using System.Globalization;
using System.IO;
using System.Runtime.InteropServices;

namespace MyMacro
{
  class Program
  {
    static void Main(string[] args)
    {
      SEInstallDataLib.SEInstallData installData = null;
      int builderNumber = 0;
      DirectoryInfo installFolder;
      CultureInfo cultureInfo;
      int majorVersion = 0;
      int minorVersion = 0;
      int parasolidMajorVersion = 0;
      int parasolidMinorVersion = 0;
      Version parasolidVersion;
      int servicePackVersion = 0;
      Version version;

      try
      {
        // Create a new instance of the SEInstallData object
        installData = new SEInstallDataLib.SEInstallData();

        // Get the build number
        builderNumber = installData.GetBuildNumber();

        // Create a new instance of DirectoryInfo using GetInstalledPath()
        installFolder = new DirectoryInfo(installData.GetInstalledPath());

        // Create a new instance of CultureInfo using GetLanguageID()
        cultureInfo = new CultureInfo(installData.GetLanguageID());

        // Get major version
        majorVersion = installData.GetMajorVersion();

        // Get minor version
        minorVersion = installData.GetMinorVersion();

        // Get parasolid major version
        parasolidMajorVersion = installData.GetParasolidMajorVersion();

        // Get parasolid minor version
        parasolidMinorVersion = installData.GetParasolidMinorVersion();

        // Create a new instance of Version using GetParasolidVersion()
        parasolidVersion = new Version(installData.GetParasolidVersion());
        // Get service pack version
        servicePackVersion = installData.GetServicePackVersion();

        // Create a new instance of Version using GetVersion()
        version = new Version(installData.GetVersion());

        // Write information to screen
        Console.WriteLine(string.Format(
          "Parasolid Version - {0}", parasolidVersion.ToString()));
        Console.WriteLine(string.Format(
          "Solid Edge Version - {0}", version.ToString()));
        Console.WriteLine(string.Format(
          "Install Path - {0}", installFolder.FullName));
      }
      catch (System.Exception ex)
      {
        Console.WriteLine(ex.Message);
      }
      finally
      {
        if (installData != null)
        {
          Marshal.ReleaseComObject(installData);
          installData = null;
        }
      }
    }
  }
}
```

Chapter 12 - Addins

This chapter will cover the basics of creating a Solid Edge addin in Visual Basic .NET. Since there is so much information to put into a single chapter, I will not be able to cover all that is possible when creating an addin.

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 addins. Specifically, Solid Edge defines an addin 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 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.

Requirements

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.

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

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

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 Addin'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.

OnConnectToEnvironment

Solid Edge passes in the category identifier of the environment as a string. If the addin 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.

OnDisconnection

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:

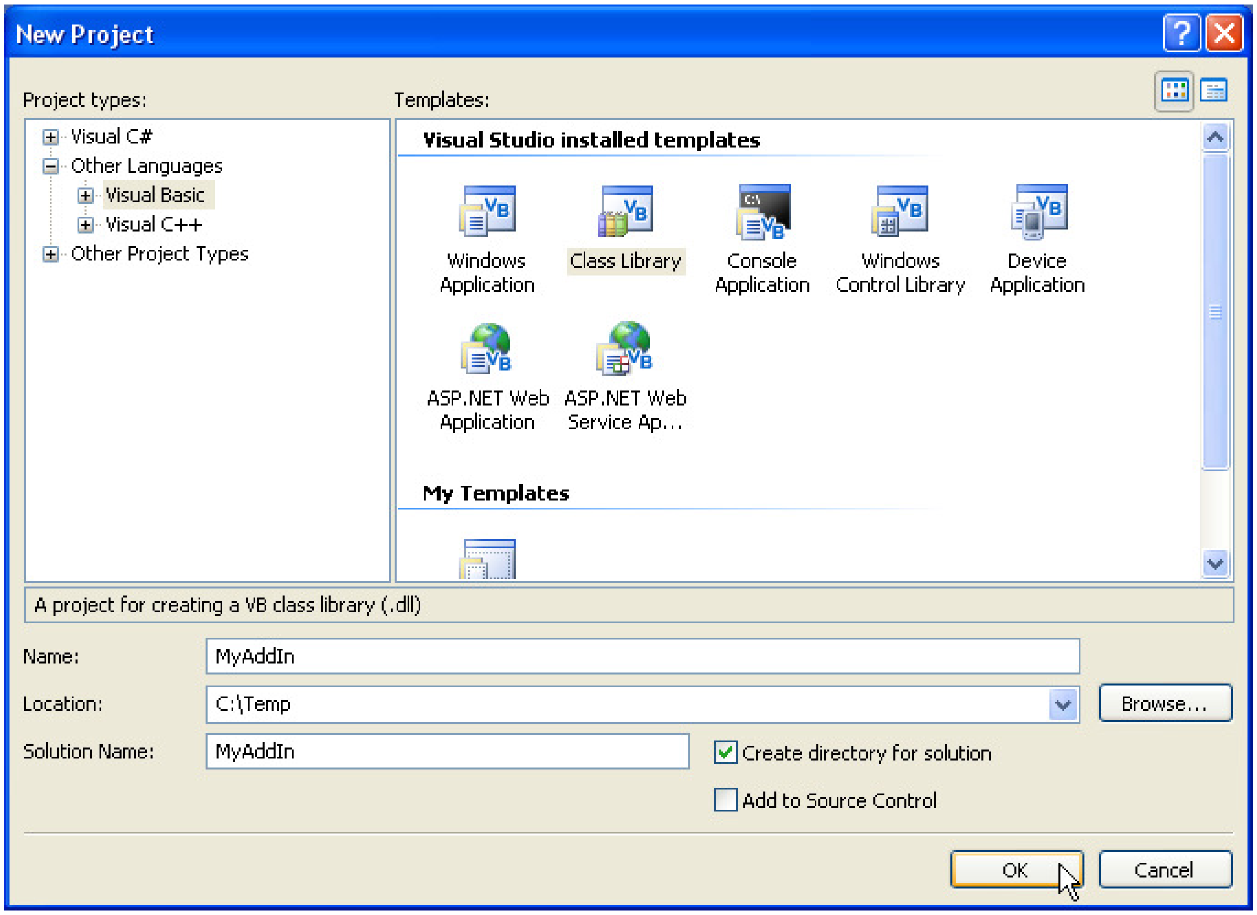
1. Disconnect from any Solid Edge event sets it may have connected to.
2. Disconnect from the Add-in event set (if connected).
3. Release any other objects or interfaces the add-in may have obtained from the application.
4. Close any storage and/or streams it may have opened in the application's document.

5. Perform any other cleanup such as freeing any resources it may have allocated.

Your first addin

Create a new Visual Basic .NET project

To create a Solid Edge addin using Visual Basic .NET, you need to start with the Class Library template offered by Visual Studio. This will produce a .NET dynamic link library that you can "COM Enable" which will allow Solid Edge to find and load your addin.



A project for creating a VB class library (.dll)

Name: MyAddIn

Location: C:\Temp

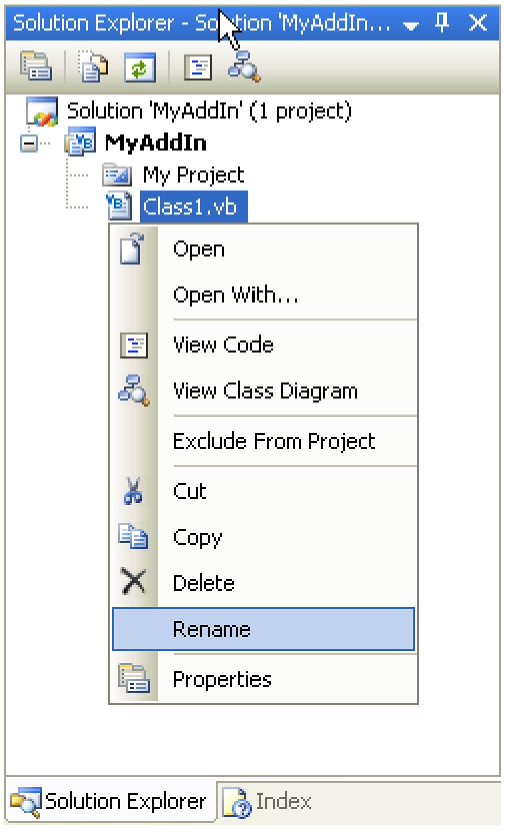
Solution Name: MyAddIn

[✓] Create directory for solution

[ ] Add to Source Control

12-1 - Create new Class Library

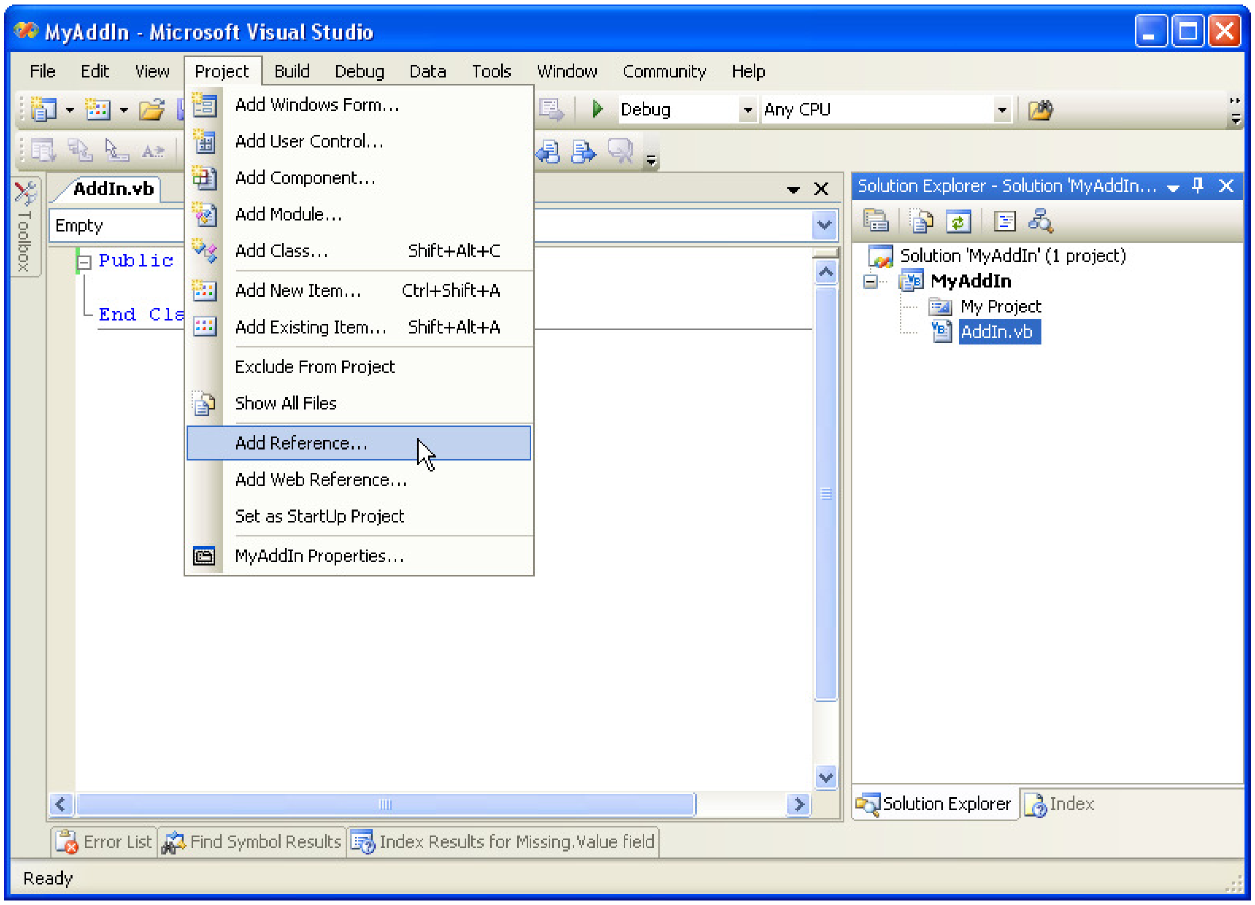
The default Class Library templates name the create you a single class file named Class1.vb. You can rename this to anything you'd like. i.e. AddIn.vb. It is fairly important to get your naming set early as changes later will not be as easy as they are at this point.



12-2 - Rename Class1.vb to AddIn.vb

Add reference to SolidEdgeFramework

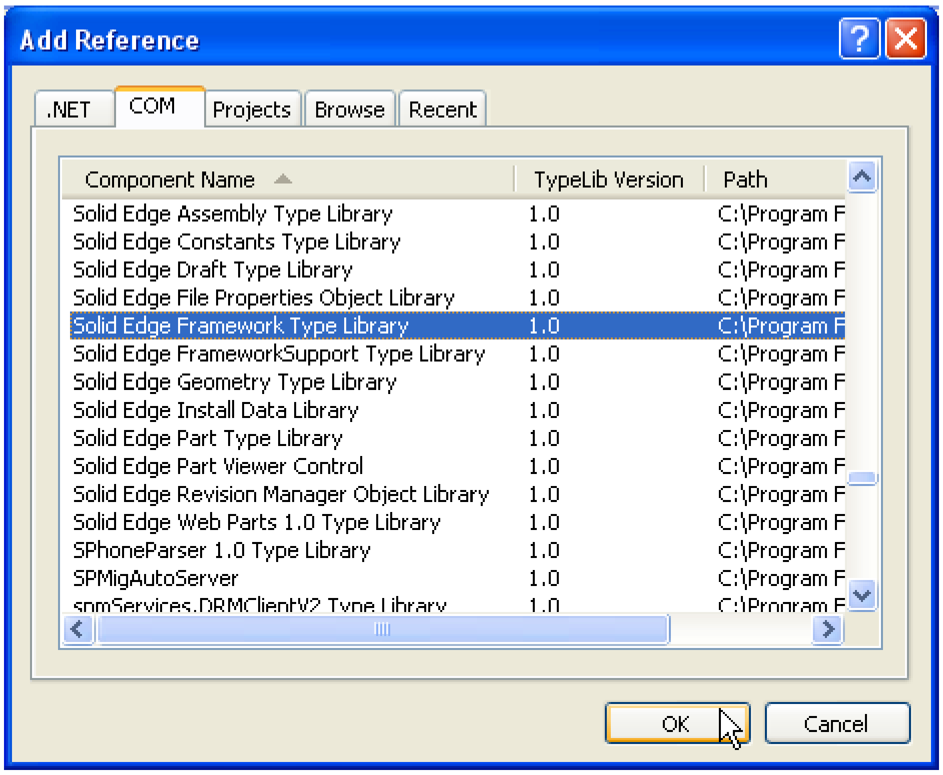
Similar to a macro project, you need to add a reference to the Solid Edge Framework Type Library. This process will generate an Interop Assembly that your addin will depend upon. It is important to remember that any Interop Assembly that you generate will need to be deployed with your addin.



```
Public
End Cla
```

12-3 - Add Reference

At a minimum, you must add a reference to the Solid Edge Framework Type Library. Depending on functionality that your addin will offer, you may need to add a reference to other Solid Edge type libraries.

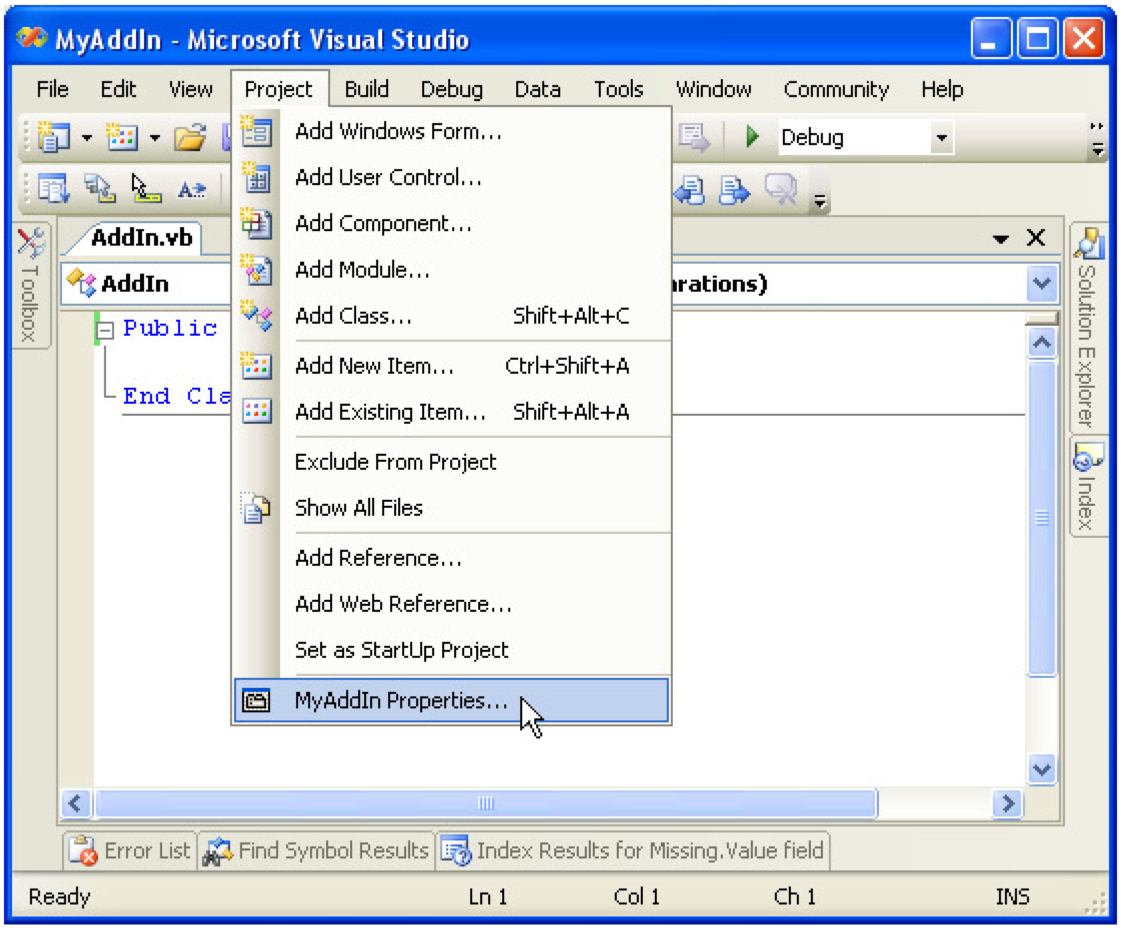


Component Name	TypeLib Version	Path
Solid Edge Assembly Type Library	1.0	C:\Program F
Solid Edge Constants Type Library	1.0	C:\Program F
Solid Edge Draft Type Library	1.0	C:\Program F
Solid Edge File Properties Object Library	1.0	C:\Program F
Solid Edge Framework Type Library	1.0	C:\Program F
Solid Edge FrameworkSupport Type Library	1.0	C:\Program F
Solid Edge Geometry Type Library	1.0	C:\Program F
Solid Edge Install Data Library	1.0	C:\Program F
Solid Edge Part Type Library	1.0	C:\Program F
Solid Edge Part Viewer Control	1.0	C:\Program F
Solid Edge Revision Manager Object Library	1.0	C:\Program F
Solid Edge Web Parts 1.0 Type Library	1.0	C:\Program F
SPhoneParser 1.0 Type Library	1.0	C:\Program F
SPMigAutoServer	1.0	C:\Program F
spmServices.DRMClientV2 Type Library	1.0	C:\Program F

12-4 - Add reference to Solid Edge Framework Type Library

Configuring Project Properties

Before you begin coding, you will need to configure your project's properties. From the Project menu, select MyAddIn Properties as shown below.

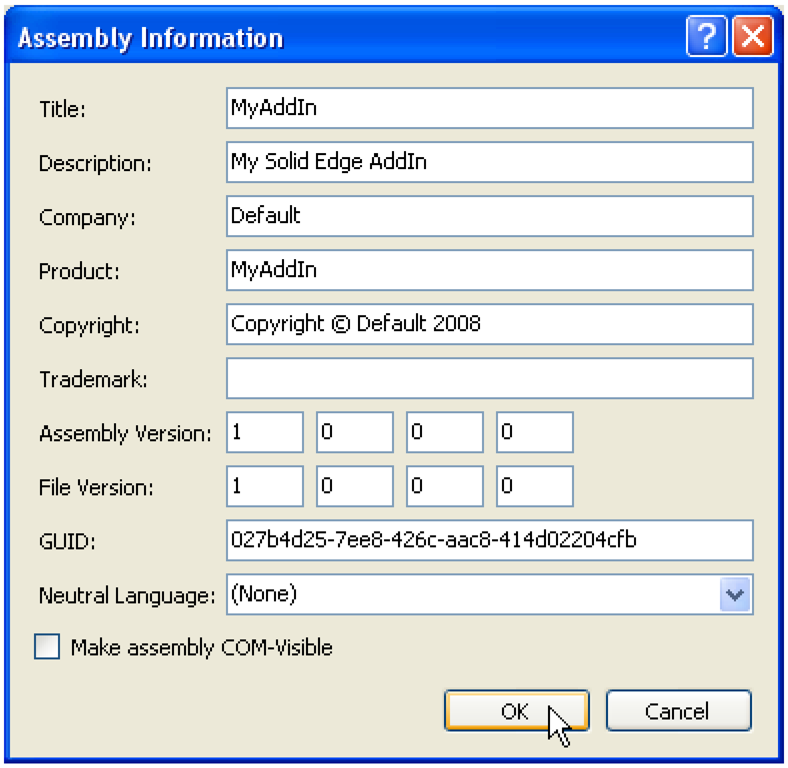


```
Public
End Cla
```

12-5 - Project Properties

Click the Assembly Information button on the first screen to set the following properties of the assembly. The Title and Description attributes will be displayed in the Solid Edge AddIn Manager.

Do not check the "Make assembly COM-Visible". This feature will make every public class, interface or enumeration visible to COM. This is not necessary as you will manually configure what is visible to COM later in this chapter.

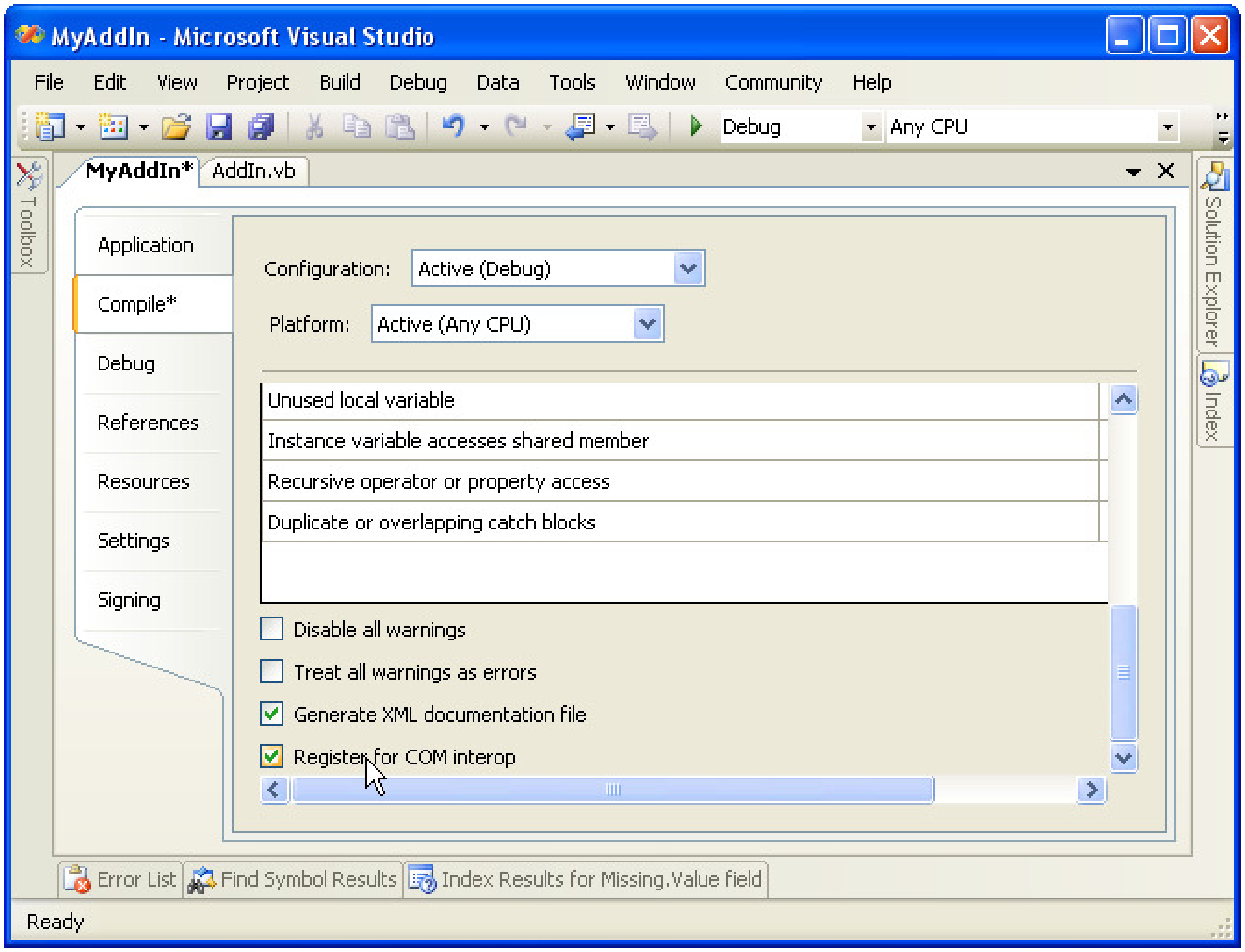


Title:	MyAddIn			
Description:	My Solid Edge AddIn			
Company:	Default			
Product:	MyAddIn			
Copyright:	Copyright © Default 2008			
Trademark:				
Assembly Version:	1	0	0	0
File Version:	1	0	0	0
GUID:	027b4d25-7ee8-426c-aac8-414d02204cfb			
Neutral Language:	(None)			

☐ Make assembly COM-Visible

12-6 - Assembly Information

In the Compile tab, scroll down to the bottom of the page and check the "Register for COM Interop" checkbox. This tells Visual Studio .NET to execute regasm.exe after you compile your project. Regasm.exe is the assembly registration tool that allows COM clients to see your assembly. This is similar to regsvr32.exe for C++ and Visual Basic 6 clients. If you need to unregister your addin, you will need to execute regasm.exe /u MyAddin.dll. Regasm.exe can typically be found in the following path: %WINDIR%\Microsoft.NET\Framework\v2.0.50727\RegAsm.exe



Configuration: Active (Debug)

Platform: Active (Any CPU)

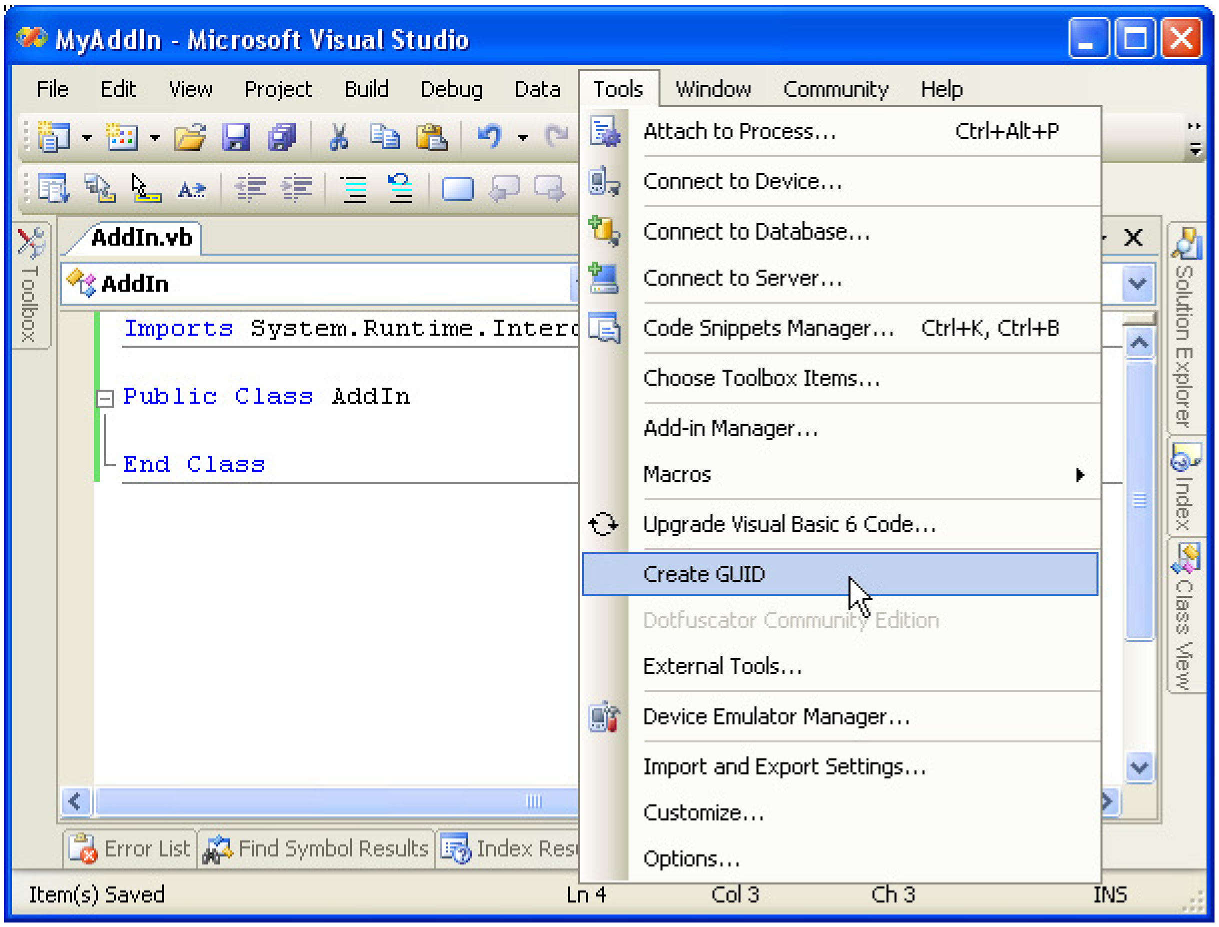
Unused local variable
Instance variable accesses shared member
Recursive operator or property access
Duplicate or overlapping catch blocks

- [ ] Disable all warnings
- [ ] Treat all warnings as errors
- [x] Generate XML documentation file
- [x] Register for COM interop

12-7 - Register for COM interop

Coding the AddIn

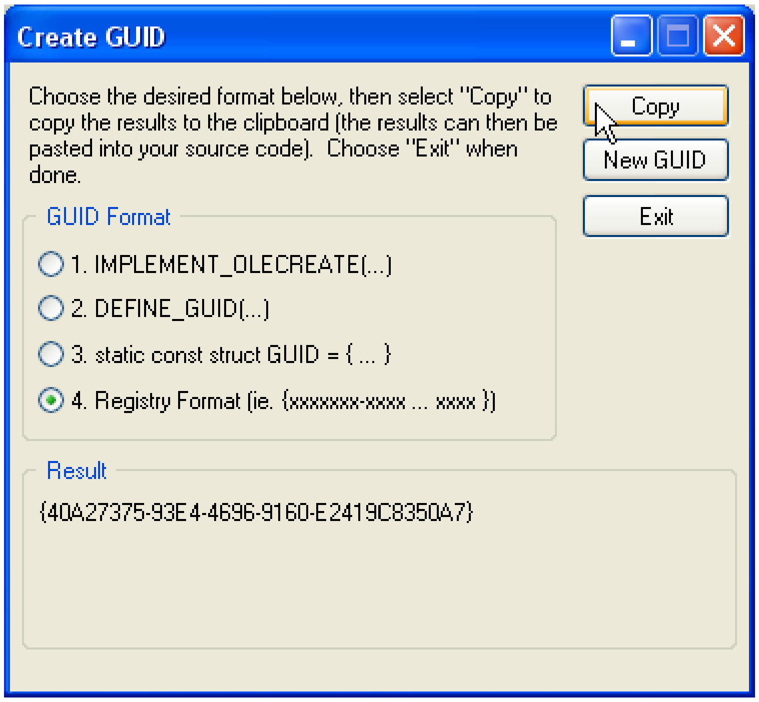
The first thing you need to do before coding the addin is to generate a unique GUID. This GUID is what will make your addin unique from other addin in Solid Edge. From the Tools menu, select Create GUID as shown below.



```
Imports System.Runtime.Intero
Public Class AddIn
End Class
```

12-8 - Create GUID

On the Create GUID screen, select option #4, Registry Format and click the Copy button. You can then click Exit. The generated GUID in this example is {40A27375-93E4-4696-9160-E2419C8350A7}. For use in the addin, you will need to remove the { and } from the GUID. i.e. You will use 40A27375-93E4-4696-9160-E2419C8350A7.



Choose the desired format below, then select "Copy" to copy the results to the clipboard (the results can then be pasted into your source code). Choose "Exit" when done.

GUID Format

- ( ) 1. IMPLEMENT_OLECREATE(...)
- ( ) 2. DEFINE_GUID(...)
- ( ) 3. static const struct GUID = { ... }
- [✓ 4. Registry Format (ie. {xxxxxxxx-xxxx ... xxxx })]

Result

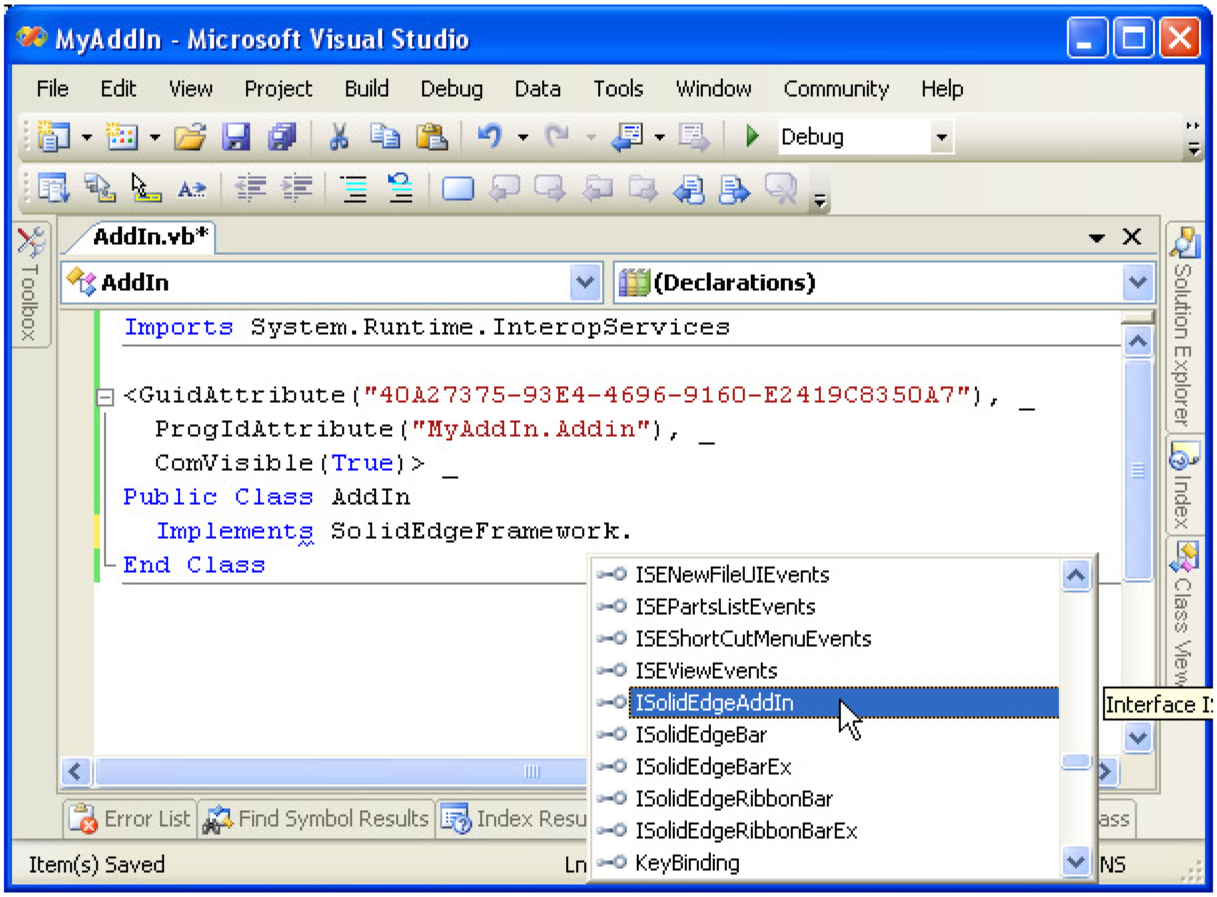
{40A27375-93E4-4696-9160-E2419C8350A7}

12-9 - Create GUID

Now you are ready to begin coding your addin. The first thing you need to do is decorate your addin class with the following attributes. This information will be used by regasm.exe when registering your assembly for COM.

```
Imports System.Runtime.InteropServices

<GuidAttribute("40A27375-93E4-4696-9160-E2419C8350A7"), _
  ProgIdAttribute("MyAddIn.Addin"), _
  ComVisible(True)> _
Public Class AddIn

End Class
```

All Solid Edge addins must have exactly one class that implements the SolidEdgeFramework.ISolidEdgeAddIn interface. The following screenshot shows the easiest approach to implementing an interface. Once intellisense comes up as shown below, highlight the ISolidEdgeAddIn interface and press your <Enter> key.

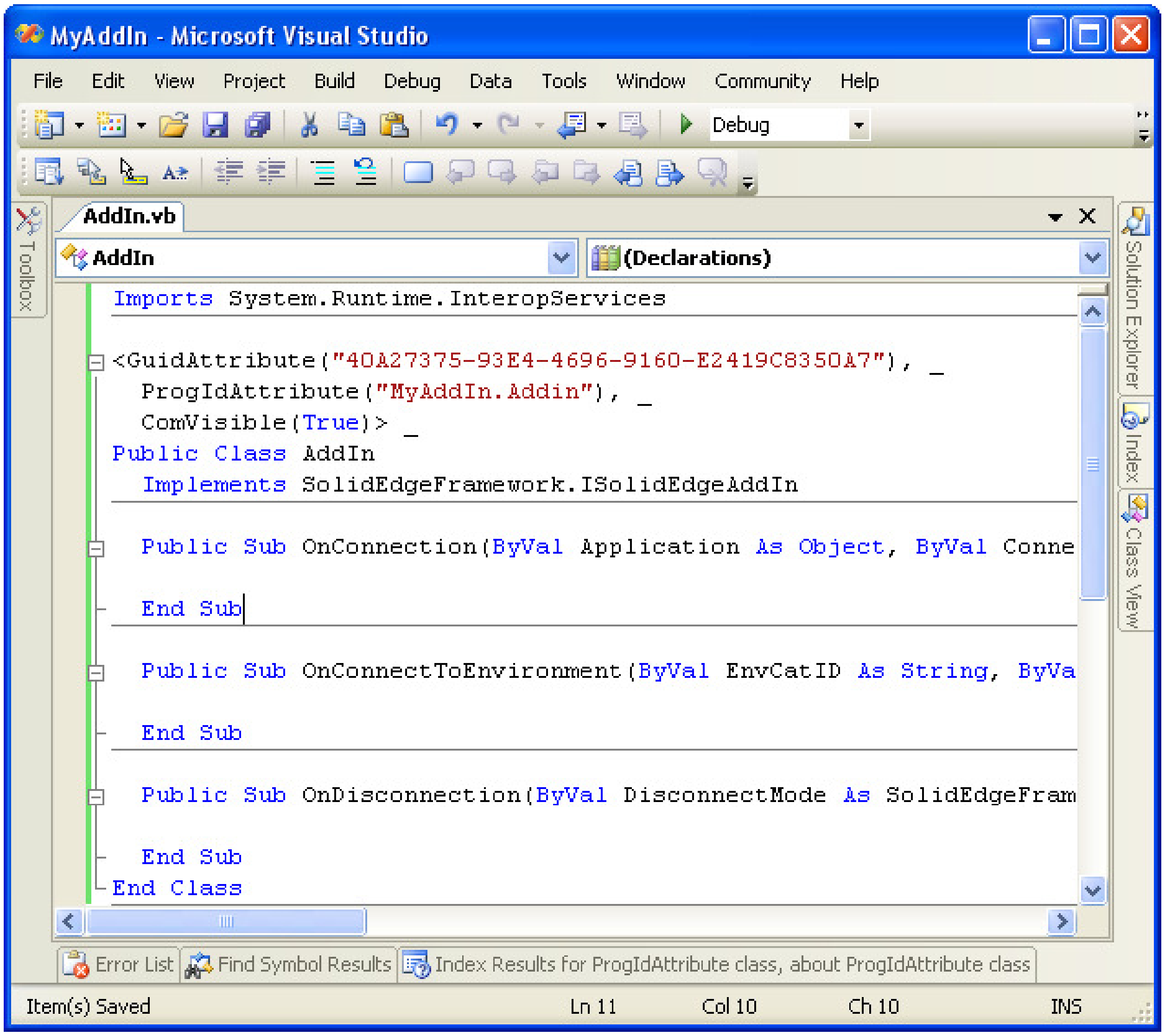


```
Imports System.Runtime.InteropServices

<GuidAttribute("40A27375-93E4-4696-9160-E2419C8350A7"), _
  ProgIdAttribute("MyAddIn.Addin"), _
  ComVisible(True)> _
Public Class AddIn
  Implements SolidEdgeFramework.
End Class
```

```
ISENewFileUIEvents
ISEPartsListEvents
ISEShortCutMenuEvents
ISEViewEvents
ISolidEdgeAddIn
ISolidEdgeBar
ISolidEdgeBarEx
ISolidEdgeRibbonBar
ISolidEdgeRibbonBarEx
KeyBinding
```

12-10 - Implementing ISolidEdgeAddIn

The next screenshot shows the ISolidEdgeAddIn interface being successfully implemented. OnConnection, OnConnectToEnvironment, and OnDisconnection are now methods in your class. Once your addin is registered and loaded by Solid Edge, it will call these three methods in your class at the appropriate times.


MyAddIn - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Window Community Help
Debug
AddIn.vb
AddIn
(Declarations)
Toolbox
Solution Explorer
Index
Class View
Error List
Find Symbol Results
Index Results for ProgIdAttribute class, about ProgIdAttribute class
Item(s) Saved
Ln 11
Col 10
Ch 10
INS

```
Imports System.Runtime.InteropServices

<GuidAttribute("40A27375-93E4-4696-9160-E2419C8350A7"), _
  ProgIdAttribute("MyAddIn.Addin"), _
  ComVisible(True)> _
Public Class AddIn
  Implements SolidEdgeFramework.ISolidEdgeAddIn

  Public Sub OnConnection(ByVal Application As Object, ByVal Conne

  End Sub

  Public Sub OnConnectToEnvironment(ByVal EnvCatID As String, ByVa

  End Sub

  Public Sub OnDisconnection(ByVal DisconnectMode As SolidEdgeFram

  End Sub
End Class
```

12-11 - ISolidEdgeAddIn implementation

The following code snippet is the bare minimum code that you need to create a Solid Edge addin. Take notice of the RegisterFunction and UnregisterFunction functions. They are decorated with the ComRegisterFunctionAttribute and ComUnregisterFunctionAttribute attributes. These special functions get executed by regasm.exe when registering \ unregistering your addin. This allows you to perform registration tasks like writing additional registry values that are required by Solid Edge.

Simple AddIn Source Code (Visual Basic .NET)

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

<GuidAttribute("40A27375-93E4-4696-9160-E2419C8350A7"), _
  ProgIdAttribute("MyAddIn.Addin"), _
  ComVisible(True)> _
Public Class AddIn
```

```
Implements SolidEdgeFramework.ISolidEdgeAddIn
Private m_addin As SolidEdgeFramework.AddIn
Private m_application As SolidEdgeFramework.Application

Public Sub OnConnection( _
  ByVal Application As Object, _
  ByVal ConnectMode As SolidEdgeFramework.SeConnectMode, _
  ByVal AddInInstance As SolidEdgeFramework.AddIn) _
  Implements SolidEdgeFramework.ISolidEdgeAddIn.OnConnection

  ' Store local variables for later use
  m_addin = AddInInstance
  m_application = Application

  ' Set Addin's GUI Version
  AddInInstance.GuiVersion = 1
End Sub

Public Sub OnConnectToEnvironment( _
  ByVal EnvCatID As String, _
  ByVal pEnvironmentDispatch As Object, _
  ByVal bFirstTime As Boolean) _
  Implements SolidEdgeFramework.ISolidEdgeAddIn.OnConnectToEnvironment

End Sub

Public Sub OnDisconnection( _
  ByVal DisconnectMode As SolidEdgeFramework.SeDisconnectMode) _
  Implements SolidEdgeFramework.ISolidEdgeAddIn.OnDisconnection

  If Not (m_addin Is Nothing) Then
    Marshal.ReleaseComObject(m_addin)
    m_addin = Nothing
  End If
  If Not (m_application Is Nothing) Then
    Marshal.ReleaseComObject(m_application)
    m_application = Nothing
  End If
End Sub

' Called by Regasm.exe
<ComRegisterFunctionAttribute()> _
Public Shared Sub RegisterFunction(ByVal t As Type)
  Dim baseKey As RegistryKey = Nothing
  Dim summaryKey As RegistryKey = Nothing
  Dim title As AssemblyTitleAttribute
  Dim description As AssemblyDescriptionAttribute

  Try
    baseKey = Registry.ClassesRoot.CreateSubKey( _
      "CLSID\{" & t.GUID.ToString & "}")

    ' Tell Solid Edge to automatically load your addin
    baseKey.SetValue("AutoConnect", 1)

    ' Write title
    If t.Assembly.IsDefined(GetType(AssemblyTitleAttribute), True) Then
```

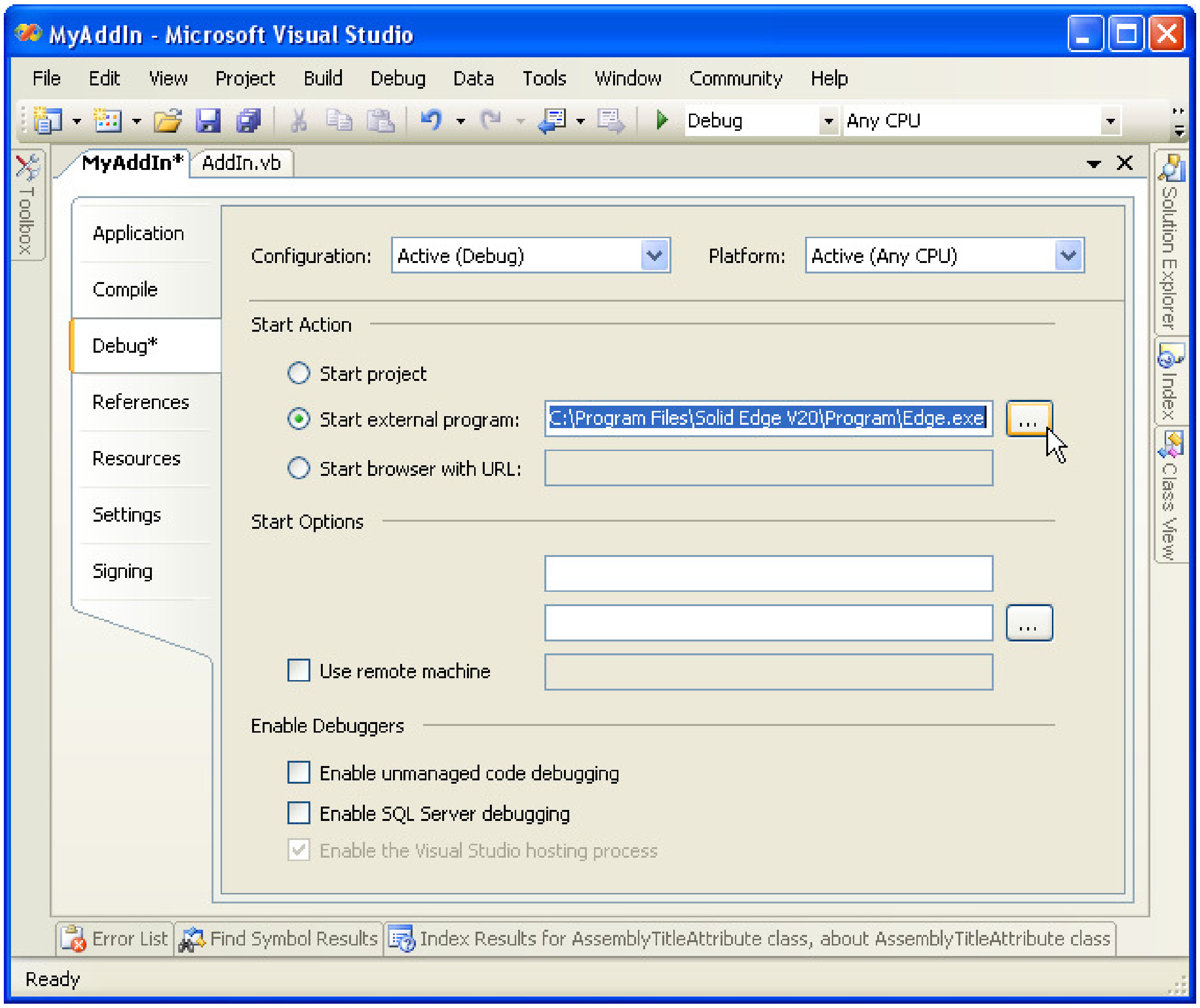
```
        title = AssemblyTitleAttribute.GetCustomAttribute( _
          t.Assembly, GetType(AssemblyTitleAttribute))
        baseKey.SetValue("409", title.Title)
      End If

      ' Write description
      If t.Assembly.IsDefined( _
        GetType(AssemblyDescriptionAttribute), True) Then

        description = AssemblyDescriptionAttribute.GetCustomAttribute( _
          t.Assembly, GetType(AssemblyDescriptionAttribute))

        summaryKey = baseKey.CreateSubKey("Summary")
        summaryKey.SetValue("409", description.Description)
        summaryKey.Close()
      End If

      ' Write required registry entries for a Solid Edge Addin.
      ' See secatids.h in C:\Program Files\Solid Edge VXX\SDK\include
      'CATID_SolidEdgeAddIn
      baseKey.CreateSubKey( _
        "Implemented Categories\{26B1D2D1-2B03-11d2-B589-080036E8B802}")

      ' Specify the Environment Categories that you want you addin to work in

      'CATID_SEApplication
      baseKey.CreateSubKey( _
        "Environment Categories\{26618394-09D6-11d1-BA07-080036230602}")

      'CATID_SEPart
      baseKey.CreateSubKey( _
        "Environment Categories\{26618396-09D6-11d1-BA07-080036230602}")
    Catch ex As Exception

    Finally
      If Not (summaryKey Is Nothing) Then
        summaryKey.Close()
      End If
      If Not (baseKey Is Nothing) Then
        baseKey.Close()
      End If
    End Try
  End Sub

  ' Called by Regasm.exe /u
  <ComUnregisterFunctionAttribute()> _
  Public Shared Sub UnregisterFunction(ByVal t As Type)
    Try
      ' Remove any previously written registry entries
      Registry.ClassesRoot.DeleteSubKeyTree( _
        "CLSID\{" + t.GUID.ToString() + "}")
    Catch ex As Exception

    End Try
  End Sub
End Class
```

Simple AddIn Source Code (C#)

```
using Microsoft.Win32;
using System;
using System.Reflection;
using System.Runtime.InteropServices;

namespace MyAddIn
{
  [GuidAttribute("40A27375-93E4-4696-9160-E2419C8350A7"),
   ProgId("MyAddIn.Addin"),
   ComVisible(true)]
  public class AddIn : SolidEdgeFramework.ISolidEdgeAddIn
  {
    private SolidEdgeFramework.AddIn m_addin;
    private SolidEdgeFramework.Application m_application;

    public AddIn()
    {
    }

    #region ISolidEdgeAddIn Members

    public void OnConnection(
      object Application,
      SolidEdgeFramework.SeConnectMode ConnectMode,
      SolidEdgeFramework.AddIn AddInInstance)
    {
      // Store local variables for later use
      m_addin = AddInInstance;
      m_application = (SolidEdgeFramework.Application)Application;

      // Set Addin's GUI Version
      AddInInstance.GuiVersion = 1;
    }

    public void OnConnectToEnvironment(
      string EnvCatID,
      object pEnvironmentDispatch,
      bool bFirstTime)
    {

    }

    public void OnDisconnection(
      SolidEdgeFramework.SeDisconnectMode DisconnectMode)
    {
      if (m_addin != null)
      {
        Marshal.ReleaseComObject(m_addin);
        m_addin = null;
      }
      if (m_application != null)
      {
        Marshal.ReleaseComObject(m_application);
        m_application = null;
      }
    }

    #endregion

    #region "Regasm.exe functions"

    // Called by Regasm.exe
    [ComRegisterFunctionAttribute()]
    static void RegisterServer(Type t)
    {
      RegistryKey baseKey = null;
      RegistryKey summaryKey = null;
      AssemblyTitleAttribute titleAttribute = null;
      AssemblyDescriptionAttribute descriptionAttribute = null;

      try
      {
        baseKey = Registry.ClassesRoot.CreateSubKey(
                    @"CLSID\{" + t.GUID.ToString() + "}");

        if (baseKey != null)
        {
          // Tell Solid Edge to automatically load your addin
          baseKey.SetValue("AutoConnect", 1);

          // Write title
          if (t.Assembly.IsDefined(typeof(AssemblyTitleAttribute), true))
          {
            titleAttribute = (AssemblyTitleAttribute)
                AssemblyTitleAttribute.GetCustomAttribute(
                    t.Assembly, typeof(AssemblyTitleAttribute));

            baseKey.SetValue("409", titleAttribute.Title);
          }

          // Write description
          if (t.Assembly.IsDefined(typeof(AssemblyDescriptionAttribute),
true))
          {
            descriptionAttribute = (AssemblyDescriptionAttribute)
                AssemblyDescriptionAttribute.GetCustomAttribute(
                    t.Assembly, typeof(AssemblyDescriptionAttribute));

            summaryKey = baseKey.CreateSubKey("Summary");
            summaryKey.SetValue("409", descriptionAttribute.Description);
          }

          // Write required registry entries for a Solid Edge Addin
          // See secatids.h in C:\Program Files\Solid Edge VXX\SDK\include

          // CATID_SolidEdgeAddIn
          baseKey.CreateSubKey(
            @"Implemented Categories\{26B1D2D1-2B03-11d2-B589-
080036E8B802}");

          // Specify the Environment Categories that you want you addin
          // to work in
          // CATID_SEApplication
          baseKey.CreateSubKey(
            @"Environment Categories\{26618394-09D6-11d1-BA07-
080036230602}");

          // CATID_SEPart
          baseKey.CreateSubKey(
            @"Environment Categories\{26618396-09D6-11d1-BA07-
080036230602}");
        }
      }
      catch
      {
      }
      finally
      {
        if (baseKey != null)
        {
          baseKey.Close();
        }
        if (baseKey != null)
        {
          baseKey.Close();
        }
      }
    }

    // Here we cleanup any registry values left from Regasm /u.
    [ComUnregisterFunctionAttribute()]
    static void UnregisterServer(Type t)
    {
      try
      {
        Registry.ClassesRoot.DeleteSubKeyTree(
          @"CLSID\{" + t.GUID.ToString() + "}");
      }
      catch
      {
      }
    }

    #endregion
  }
}
```

Debugging your AddIn

The last step you will need to perform is to configure the debugger. Go back to the project properties as described previously and click on the Debug tab. For the Start Action, select Start external program and browse to Edge.exe. This will allow you to debug your addin during execution of Solid Edge.


MyAddIn - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Window Community Help
Debug
Any CPU
MyAddIn*
AddIn.vb
Toolbox
Solution Explorer
Index
Class View
Application
Compile
Debug*
References
Resources
Settings
Signing
Error List
Find Symbol Results
Index Results for AssemblyTitleAttribute class, about AssemblyTitleAttribute class
Ready

Configuration: Active (Debug) Platform: Active (Any CPU)

Start Action

Start project [✓ Start external program:] C:\Program Files\Solid Edge V20\Program\Edge.exe

Start browser with URL:

Start Options

☐ Use remote machine

Enable Debuggers

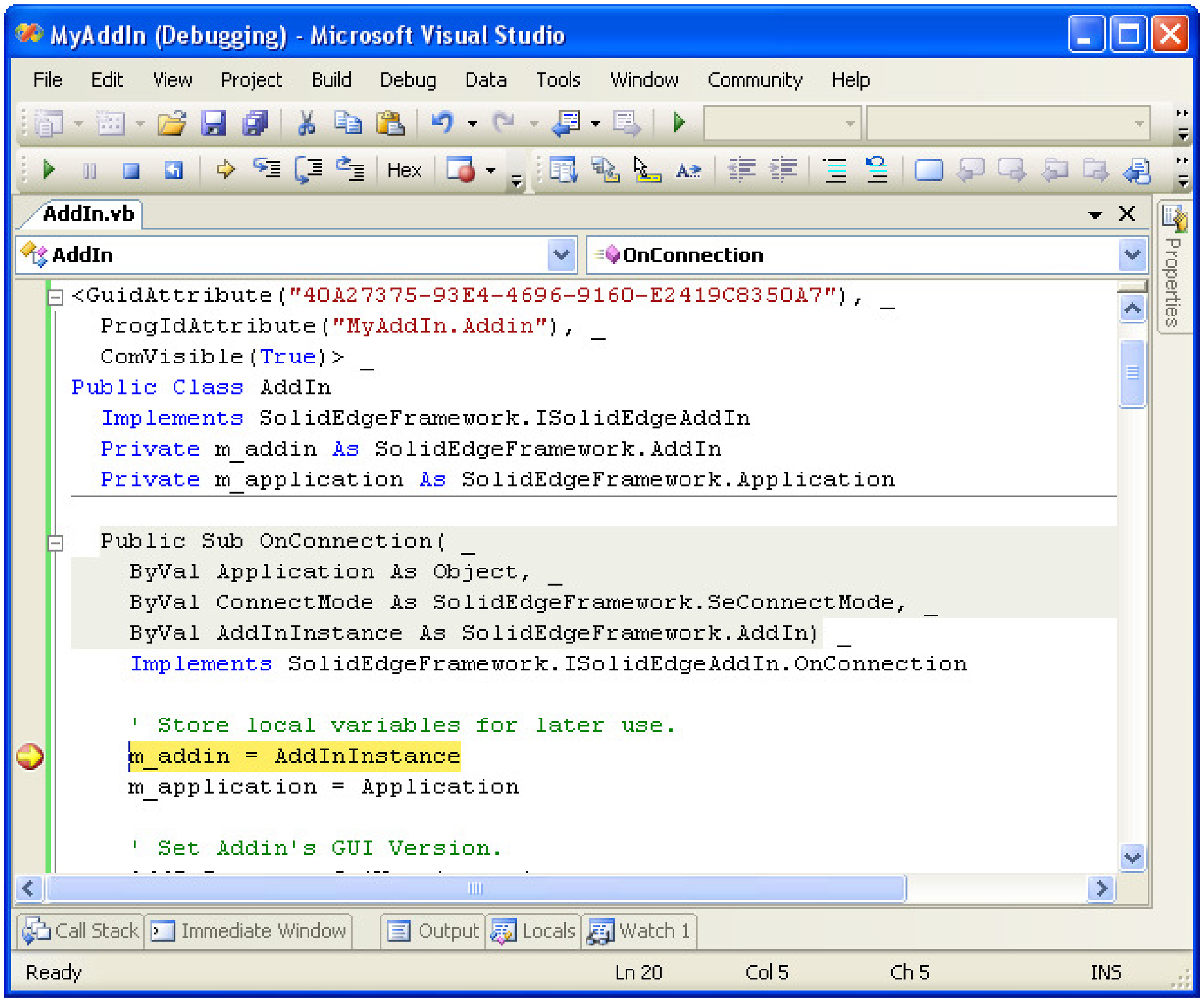
☐ Enable unmanaged code debugging

☐ Enable SQL Server debugging

☑ Enable the Visual Studio hosting process

12-12 - Debug Configuration

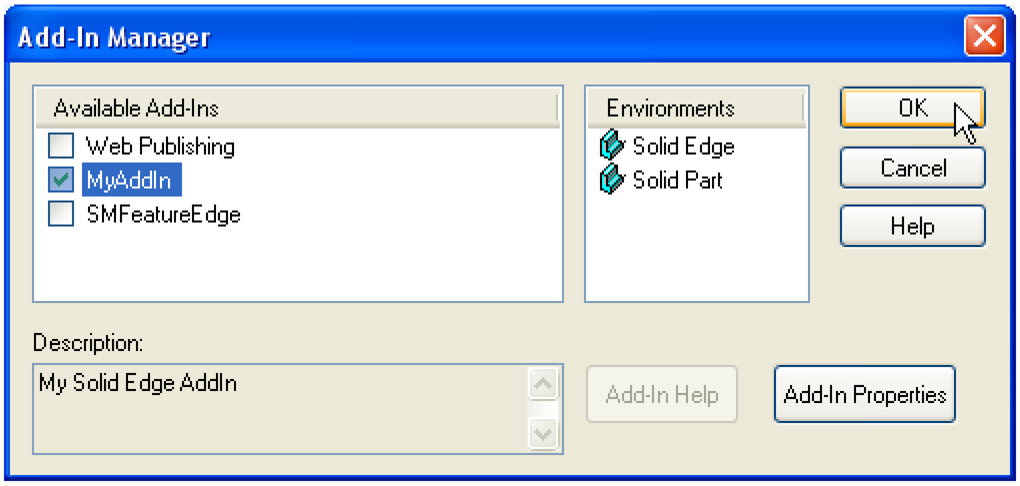
Place a breakpoint in your code as shown below and press the <F5> key to begin debugging. If your solution is setup correctly, Solid Edge will begin loading and Visual Studio .NET will break on your code.


MyAddIn (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Window Community Help
Hex
AddIn.vb
AddIn
OnConnection
Properties
Call Stack
Immediate Window
Output
Locals
Watch 1
Ready
Ln 20
Col 5
Ch 5
INS

```
<GuidAttribute("40A27375-93E4-4696-9160-E2419C8350A7"), _
  ProgIdAttribute("MyAddIn.Addin"), _
  ComVisible(True)> _
Public Class AddIn
  Implements SolidEdgeFramework.ISolidEdgeAddIn
  Private m_addin As SolidEdgeFramework.AddIn
  Private m_application As SolidEdgeFramework.Application

  Public Sub OnConnection( _
    ByVal Application As Object, _
    ByVal ConnectMode As SolidEdgeFramework.SeConnectMode, _
    ByVal AddInInstance As SolidEdgeFramework.AddIn) _
    Implements SolidEdgeFramework.ISolidEdgeAddIn.OnConnection

    ' Store local variables for later use.
    m_addin = AddInInstance
    m_application = Application

    ' Set Addin's GUI Version.
```

12-13 - Debugging your addin

Once inside Solid Edge, open the AddIn Manager. You should see your addin listed as shown below.



12-14 - AddIn Manager

Chapter 13 - Useful References

Microsoft Developer Network (MSDN) Links

Microsoft Developer Network (MSDN) - http://msdn2.microsoft.com

Visual Studio .NET Technology Map

http://msdn2.microsoft.com/library/ms973926.aspx

Visual Basic .NET Technology Map

http://msdn2.microsoft.com/library/ms973925.aspx

Microsoft .NET Framework FAQ

http://msdn2.microsoft.com/library/ms973850.aspx

MSDN Code Gallery

http://code.msdn.microsoft.com

General Naming Conventions

http://msdn2.microsoft.com/library/ms229045(VS.80).aspx

Microsoft Win32 to Microsoft .NET Framework API Map

http://msdn2.microsoft.com/library/aa302340.aspx

Microsoft .NET/COM Migration and Interoperability

http://msdn2.microsoft.com/library/ms978506.aspx

Improving .NET Application Performance and Scalability

http://msdn2.microsoft.com/library/ms998530.aspx

Exception Management in .NET

http://msdn2.microsoft.com/library/ms954599.aspx

Handling and Throwing Exceptions

http://msdn2.microsoft.com/library/aa720123.aspx

Quick Technology Finder

http://msdn2.microsoft.com/library/63bf39c2(VS.80).aspx

Deploying .NET Framework-based Applications

http://msdn2.microsoft.com/library/ms954585.aspx

Beyond (COM) Add Reference Has Anyone Seen the Bridge

http://msdn2.microsoft.com/library/ms973274.aspx

101 Samples for Visual Studio 2005

http://msdn2.microsoft.com/vs2005/aa718334.aspx

Microsoft Newsgroups

microsoft.public.dotnet.faqs

news://msnews.microsoft.com/microsoft.public.dotnet.faqs

microsoft.public.dotnet.framework

news://msnews.microsoft.com/microsoft.public.dotnet.framework

microsoft.public.dotnet.framework.interop

news://msnews.microsoft.com/microsoft.public.dotnet.framework.interop

microsoft.public.dotnet.framework.performance

news://msnews.microsoft.com/microsoft.public.dotnet.framework.performance

microsoft.public.dotnet.framework.windowsforms

news://msnews.microsoft.com/microsoft.public.dotnet.framework.windowsforms

microsoft.public.dotnet.framework.windowsforms.controls

news://msnews.microsoft.com/microsoft.public.dotnet.framework.windowsforms.controls

microsoft.public.dotnet.general

news://msnews.microsoft.com/microsoft.public.dotnet.general

microsoft.public.dotnet.languages.csharp

news://msnews.microsoft.com/microsoft.public.dotnet.languages.csharp

microsoft.public.dotnet.languages.vb

news://msnews.microsoft.com/microsoft.public.dotnet.languages.vb

microsoft.public.dotnet.languages.vb.controls

news://msnews.microsoft.com/microsoft.public.dotnet.languages.vb.controls

Solid Edge Newsgroups

* These newsgroups require a GTAC login.

solid_edge.binaries

news://bbsnotes.ugs.com/solid_edge.binaries

solid_edge.insight
news://bbsnotes.ugs.com/solid_edge.insight

solid_edge.misc
news://bbsnotes.ugs.com/solid_edge.misc

solid_edge.programming
news://bbsnotes.ugs.com/solid_edge.programming

Programming Links

The Code Project
http://www.codeproject.com

PINVOKE.NET
http://pinvoke.net

DotNetJunkies
http://www.dotnetjunkies.com

VB.NET Heaven
http://www.vbdotnetheaven.com

vbAccelerator
http://www.vbaccelerator.com

CSharpFriends
http://www.csharpfriends.com

ic#code
http://www.icsharpcode.net

JasonNewell.NET
http://www.jasonnewell.net