Paper 298-2012

Not Just for Scheduling:
Doing More with SAS® Enterprise Guide® Automation

Chris Hemedinger, SAS Institute Inc, Cary, NC

ABSTRACT

SAS Enterprise Guide supports a rich automation model that allows you to "script" almost every aspect of opening and running SAS Enterprise Guide projects. While this is most often used with the built-in "scheduler" function to simply run a project or process flow unattended, you can also write your own scripts that automate much more. You can create new projects on the fly, supply values for project prompts, run individual tasks, export results, and more. You are not limited to the traditional VB Script method of automation; you can also use more sophisticated frameworks such as Windows PowerShell or Microsoft .NET.

This paper describes the concepts behind automation for SAS Enterprise Guide, and provides examples to accomplish a variety of tasks using the different scripting technologies.

INTRODUCTION

Most people use SAS Enterprise Guide through its user interface, which allows point-and-click access to all of its features. But sometimes you need to use those features *outside* of the user interface, perhaps replaying some of the work that you have captured within a project on a scheduled basis or as part of another process. To accomplish this, you use the SAS Enterprise Guide automation interface.

Ever since its first release, SAS Enterprise Guide has offered a mechanism for the unattended "running" of projects. Have you ever wondered what happens when you select **File->Schedule Project** in SAS Enterprise Guide? First, SAS Enterprise Guide generates a script file. The script file contains the instructions for launching the "engine" of the application, for loading a project file, and for running the project through to completion. Next, SAS Enterprise Guide hooks into the Windows scheduler, allowing you to specify a date and time to run the script.

This paper is divided into two main parts. In the brief first part, we discuss the architecture and concepts behind SAS Enterprise Guide automation. The remainder of the paper is dedicated to showing specific examples of driving the automation using scripting languages and other applications. The examples were developed with SAS Enterprise Guide 4.3, but with minor changes (as noted) they can work with SAS Enterprise Guide 5.1.

THE AUTOMATION ARCHITECTURE

SAS Enterprise Guide allows for automation via an application programming interface (API), which is surfaced using a special module named SASEGScripting.dll. You can access the API in two ways: by using the COM-based object model or by using Microsoft .NET. Whether you use COM or Microsoft .NET, the automation object model is the same.

When you use scripting languages such as VBScript or Windows PowerShell, you will access the COM-based object model. When you use Microsoft .NET (with programming languages such as C# or Visual Basic .NET), you add a direct reference to the SASEGScripting.dll and bypass the COM layer.

Figure 1 shows a simple view of the automation architecture. As the diagram shows, the automation interface is a peer to the main user interface for SAS Enterprise Guide. Like the happy gentleman pictured at the top of the diagram, most users will interact with the main windows of SAS Enterprise Guide. Users gain access to this user interface by using the primary executable (SEGuide.exe), often by way of a desktop shortcut icon. With automation, you forgo the SAS Enterprise Guide user interface entirely, and you instead script every action using the automation API.

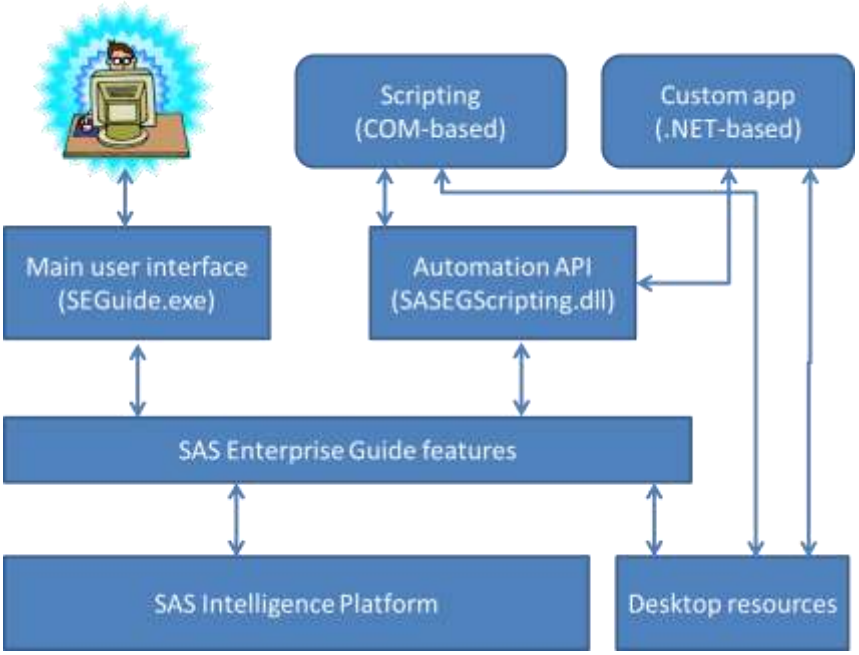

Figure 1. Simple View of the Automation Architecture

THE AUTOMATION OBJECT MODEL

The automation API is a collection of methods and properties that are organized into an object-oriented programming model. There are dozens of discrete object types within the automation model. The full set of objects is documented within the Scripting Reference documentation that is available on support.sas.com. (See References.)

In addition to its "familiar" name, each object type has an interface that it implements. The interface can be thought of as a contract: a set of documented methods and properties that the object promises to support. Each interface follows a naming convention of ISASEG<*object-type*>, where "ISASEG" signifies "an interface in the SAS Enterprise Guide namespace". You do not need to know the interface name while you build scripts for automation. However, it helps to be familiar with the interface names as you navigate the reference documentation.

Figure 2 shows an example of the documentation for the Code object (referenced as ISASEGCode in the documentation). The documentation offers brief descriptions of each method and property. In some cases, the documentation contains a brief example of how to use the object in a script.

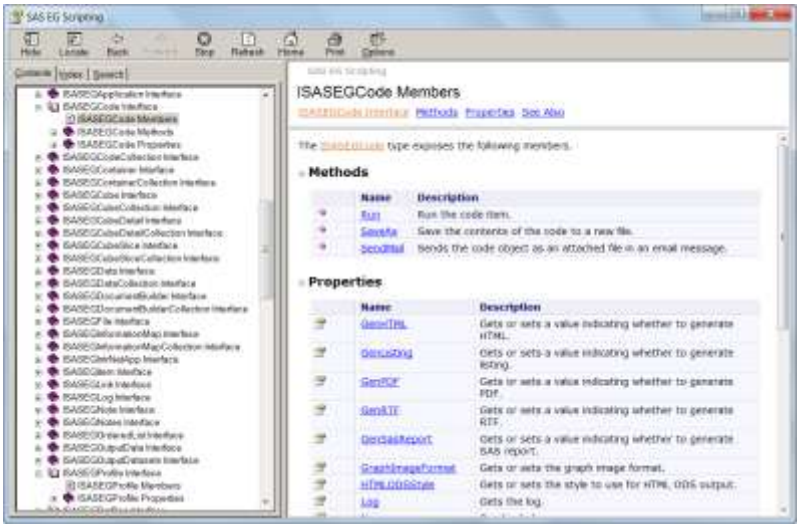

Figure 2. Example of Scripting Reference Documentation

This paper includes descriptions of just a few types of objects that are necessary to get started. The following

paragraphs describe each object at a high level. Each paragraph heading features the object name with its interface name in parentheses.

Application (ISASEGApplication)

The Application object is the main instance of the SAS Enterprise Guide automation interface. Using the Application object, you can open projects, create new projects, set the current SAS environment (mapping to a named metadata profile), and set application-level options (as you would do with the **Tools->Options** menu in the full SAS Enterprise Guide application). The Application object is the gateway to all other automation features in SAS Enterprise Guide.

Project (ISASEGProject)

The Project object represents an instance of a SAS Enterprise Guide project file. You can discover the contents of the project, including its process flows, data references, SAS programs, tasks, and results.

Each item within the project is represented as an object of its own type. All of these objects share two common properties: Name and Type. The Name value is the name of the object as you would see it within the SAS Enterprise Guide interface – its label. The Type value is number that maps to an enumerated list of types. Depending on the type, the methods and properties that apply to the object will be different.

Container (ISASEGContainer)

A process flow is represented as a Container object; it is a top-level "container" within the SAS Enterprise Guide project. There are other types of Container objects (specialized for reports or distribution activities), but the process flow is the most useful type to automate. It supports the Run method, which allows you to run just one process flow from the active project, or to run a series of process flows in whatever sequence you want.

Profile (ISASEGProfile)

A Profile object represents a SAS environment that you connect to using SAS Enterprise Guide. The profile consists of a machine host name for a SAS metadata server, a port number, and optional stored credentials for authenticating to the environment. Many users must connect to different environments in the course of their work. For example, they might have a "development" environment as well as a "production" environment. Or they might sometimes use a local installation of SAS (with no metadata server) instead of an enterprise environment.

The automation API provides a method to set the active profile (by name), so that you can point your instance of SAS Enterprise Guide to the correct environment for your current work.

Code (ISASEGCode)

The Code object represents a SAS program in the project. For automation users, the Code object is a very powerful entry point to the SAS platform. With the Code object, you can drive your SAS session to perform whatever work you can represent in a SAS program. You can use automation to run existing SAS programs within a project, or to add new programs with your own code content.

You run the Code object by using the Run method. In addition to the Run method, the Code object supports several properties that allow you to programmatically examine the output of a program, including the Log, Results (including various forms of SAS ODS output), and OutputDatasets.

Data (ISASEGData)

The Data object represents a reference to a data source in the project. Typically, this is a data set within a SAS library. It might be a data reference that was added when the project was designed, or it might be part of the output data that is created by a Code object, Query object, or Task object.

Perhaps the most useful method on the Data object is the SaveAs method, which allows you to save the SAS data set to your local machine in a format that any application can use, such as CSV or Microsoft Excel.

Task (ISASEGTask) and Query (ISASEGQuery)

The Task and Query objects are similar in many respects. They both represent work that has been "designed" by a user in the SAS Enterprise Guide user interface. Typically, they both generate SAS programs. They are "runnable"; that is, they support a Run method. When they are run, they each generate various results that you can access and export to external files (with the Results, Log, and OutputDatasets properties).

There are dozens of other object types that you can access by using automation. Some are very useful, and you will encounter them in the examples throughout this paper. Others are more obscure and seldom used.

SELECTING THE CORRECT "BITNESS" FOR SCRIPTING

SAS Enterprise Guide 4.3 is a 32-bit application. Many users now have 64-bit machines that run a 64-bit version of Microsoft Windows. The built-in scripting engines (cscript.exe for VBScript and powershell.exe for PowerShell) are supplied in two flavors: 32-bit and 64-bit. In order to run scripts properly on a 64-bit operating system, be sure to use the 32-bit version of your scripting engine. For example, to run a VBScript file, use a command such as:

```
c:\Windows\SysWOW64\cscript.exe c:\projects\AutomationNewProgram.vbs
```

Note the counterintuitive folder name (SysWOW64) for the location of the 32-bit version of cscript.exe. On 64-bit versions of Windows, the built-in 32-bit tools are in the SysWOW64 directory, while installed 32-bit applications are usually in a directory marked "x86", as in "C:\Program Files (x86)\".

SAS Enterprise Guide 5.1 offers both a 32-bit version and a 64-bit version. If you are automating SAS Enterprise Guide 5.1, make sure that you select the correct version of the scripting engine to match the "bitness " of SAS Enterprise Guide that you have installed.

SELECTING A SCRIPTING TECHNOLOGY

To build an automated process with SAS Enterprise Guide, you can use any scripting technology that allows either COM automation or interaction with a Microsoft .NET programming library. This paper addresses three of the most common mechanisms: VBScript, Windows PowerShell, and Microsoft Visual Studio (using the C# programming language). The following is a brief summary of the capabilities of each technology:

VBScript

VBScript is the most common Windows scripting language, and has the longest history. When you use the Schedule feature in SAS Enterprise Guide, the application generates a VBScript program (VBS file) to use for automation. The basic syntax of VBScript is similar to Visual Basic, and many IT professionals are familiar with it. However, the VBScript environment does not offer much of a framework. All interactions with external resources (such as the local computer, file system, or other processes) must be accomplished with late-bound objects that are invoked with `CreateObject` calls. Because VBScript is ubiquitous in Windows environments, you can depend on its capabilities without having to install additional components.

Similar to VBScript, you can also use Visual Basic for Applications (VBA) within Microsoft Office applications. For example, you can automate SAS Enterprise Guide within Microsoft Excel. This combination gives you access to SAS and SAS Enterprise Guide features, all driven from within Excel macros.

Windows PowerShell

Windows PowerShell is often touted as the next-generation of Windows scripting. If you are familiar with shell programming from UNIX operating systems, Windows PowerShell will appear similar in concept and capability. Almost every aspect of Microsoft Windows can be automated with PowerShell, and it is easy to automate COM objects. You can also incorporate calls to Microsoft .NET code libraries. It's also possible to extend PowerShell with new commands (called cmdlets), which are developed using Microsoft .NET.

Because Windows PowerShell is so flexible and powerful, it requires an additional level of trust to be granted before you can run scripts in an automated way. The PowerShell syntax is a bit tricky to grasp, but SAS programmers might be able to transfer some of their knowledge to the task. For example, PowerShell functions work in a similar way to SAS macro functions.

Microsoft .NET (C# or Visual Basic .NET)

Microsoft .NET, when accessed through Microsoft Visual Studio, offers a full application development platform. This might be the best choice if you want to build an end-user application, complete with a user interface. You can download "Express" editions of Microsoft Visual C# or Visual Basic .NET for free from the Microsoft Web site. However, if you want the best productivity with the tool, consider investing in one of the professional editions.

SCRIPTING WITH VBSCRIPT, BY EXAMPLE

This section provides a collection of scripts that perform useful tasks. For some of the longer scripts, only select sections of the source code are included in the paper due to space concerns. The complete source code for all of these scripts is available online at support.sas.com. See the References section for the details.

Note: As you read the example code, you will notice that VBScript is not like the SAS language, with its semicolons to end a statement. In VBScript (and Visual Basic) each statement must appear on its own line. If you want to break the line up to make it easier to read, you must add an underscore as the last character on each line except the final

line of a statement. For the examples within this paper, the underscores were added to break up the lines to make each program easier to read.

EXAMPLE 1: THE SIMPLEST SCRIPT

The following VBScript program does no real work, but simply creates an instance of the SAS Enterprise Guide `Application` object and reports on the `Application.Name` and `Application.Version` properties.

```
Dim Application
' Create a new SAS Enterprise Guide automation session
Set Application = WScript.CreateObject("SASEGObjectModel.Application.4.3")
WScript.Echo Application.Name & ", Version: " & Application.Version
Application.Quit
```

The `CreateObject` method in the scripting engine (WScript) creates a new instance of the Application object, which is registered with Windows using a version-specific program identifier (often abbreviated to "ProgID"). The ProgID for SAS Enterprise Guide 4.3 is `SASEGObjectModel.Application.4.3`. With the 5.1 version, the ProgID changes (predictably) to `SASEGObjectModel.Application.5.1`. The output of this script looks something like this:

```
C:\Examples> cscript NewApp.vbs

Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.

Enterprise Guide, Version: 4.3.0.0
```

Even though this script does not perform any work, it is a useful way to check that the automation mechanism is intact. If this simple script generates an error message, then there is probably something amiss with the SAS Enterprise Guide installation, and more complex scripts do not have any chance of running successfully.

EXAMPLE 2: LIST AVAILABLE METADATA PROFILES

As described earlier, the `Profile` object allows you to specify which SAS environment to use during the automation session. The following script shows how to list all of the different metadata profiles that are defined for use by the current user:

```
Dim Application
' Create a new SAS Enterprise Guide automation session
Set Application = WScript.CreateObject("SASEGObjectModel.Application.4.3")
WScript.Echo Application.Name & ", Version: " & Application.Version

' Discover the available profiles that are defined for the current user
Dim i
Dim oShell
Set oShell = CreateObject( "WScript.Shell" )
WScript.Echo "Metadata profiles available for " _
   & oShell.ExpandEnvironmentStrings("%UserName%")
WScript.Echo "----------------------------------------"
For i = 1 to Application.Profiles.Count-1
  WScript.Echo "Profile available: " _
    & Application.Profiles.Item(i).Name _
    & ", Host: " & Application.Profiles.Item(i).HostName _
    & ", Port: " & Application.Profiles.Item(i).Port
Next
Application.Quit
```

Here is a sample of the command and output:

```
C:\Examples> cscript ShowProfiles.vbs

Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.

Enterprise Guide, Version: 4.3.0.0
```

```
Metadata profiles available for sascrh
---------------------------------------
Profile available: Null Provider, Host: , Port:
Profile available: t2817, Host: t2817, Port: 8562
Profile available: uitsrv02, Host: uitsrv02.na.sas.com, Port: 8561
Profile available: uitsrv04, Host: uitsrv04.na.sas.com, Port: 8561
```

EXAMPLE 3: RUN A SAS PROGRAM AS A BATCH JOB

SAS users have been running SAS programs in batch (unattended) for decades. To run a program in batch, you typically invoke a SAS command with your SAS program file as the input argument. SAS runs the program, and writes the log output and text-based listing output to your local folder.

With SAS Enterprise Guide and a connection to a remote SAS session, many users no longer have access to the SAS command; therefore, they cannot run batch jobs in the traditional way. This example script simulates a batch session by connecting to a SAS workspace server, running a SAS program, and writing the output (log, listing, and data) to the local script folder.

```
Option Explicit ' Forces us to declare all variables

Dim Application ' Application
Dim Project     ' Project object
Dim sasProgram  ' Code object (SAS program)
Dim n           ' counter

Set Application = CreateObject("SASEGObjectModel.Application.4.3")
' Set to your metadata profile name, or "Null Provider" for just Local server
Application.SetActiveProfile("My Server")
' Create a new Project
Set Project = Application.New
' add a new code object to the Project
Set sasProgram = Project.CodeCollection.Add

' set the results types, overriding Application defaults
sasProgram.UseApplicationOptions = False
sasProgram.GenListing = True
sasProgram.GenSasReport = False

' Set the server (by Name) and text for the code
sasProgram.Server = "SASApp"
' Create the SAS program to run
sasProgram.Text = "data work.cars; set sashelp.cars; if ranuni(0)<0.85; run;"
sasProgram.Text =  sasProgram.Text & " proc means; run;"

' Run the code
sasProgram.Run
' Save the log file to LOCAL disk
sasProgram.Log.SaveAs _
  getCurrentDirectory & "\" & WScript.ScriptName & ".log"

 ' Save all output data as local Excel files
For n=0 to (sasProgram.OutputDatasets.Count -1)
  Dim dataName
  dataName = sasProgram.OutputDatasets.Item(n).Name
  sasProgram.OutputDatasets.Item(n).SaveAs _
    getCurrentDirectory & "\" & dataName & ".xls"
Next

' Filter through the results and save just the LISTING type
For n=0 to (sasProgram.Results.Count -1)
    ' Listing type is 7
    If sasProgram.Results.Item(n).Type = 7 Then
           ' Save the listing file to LOCAL disk
```

```
            sasProgram.Results.Item(n).SaveAs _
               getCurrentDirectory & "\" & WScript.ScriptName & ".lst"
    End If
Next

Application.Quit

' function to fetch the current directory
Function getCurrentDirectory
    Dim oFSO
    Set oFSO = CreateObject("Scripting.FileSystemObject")
    getCurrentDirectory = _
      oFSO.GetParentFolderName(WScript.ScriptFullName)
End Function
```

Assuming that the script is named BatchProject.vbs, you can run it with a command like:

```
C:\Examples> cscript BatchProject.vbs
```

This script does not create any output to the console, but after running it you can see all of the output in the current directory where the script resides.

```
02/02/2012  01:55 PM             1,952 BatchProject.vbs
02/02/2012  01:56 PM               917 BatchProject.vbs.log
02/02/2012  01:56 PM             1,752 BatchProject.vbs.lst
02/02/2012  01:56 PM            79,872 CARS.xls
```

Here are a few lessons to take from this script example:

- You can use automation to create a new SAS Enterprise Guide project and a Code object "on the fly". This example does not start with an existing EGP file. The program uses `Application.New` to create a project, and `CodeCollection.Add` to create a new program node.
- You can use properties on the Code object to override application-level options for ODS statements. In this example, we turned off the default SAS Report output and asked for LISTING.
- You can use automation methods to save output from the project to your local computer. In this example, we save Log and LISTING output.
- You can use automation to export SAS data to file formats such as Microsoft Excel or CSV. SAS Enterprise Guide performs the export operation, which does **not** require SAS/ACCESS to PC Files. In a traditional SAS batch program, you might use PROC EXPORT to create the Excel file.

EXAMPLE 4: EXTRACT SAS PROGRAMS FROM AN EXISTING PROJECT

SAS Enterprise Guide project files are a convenient mechanism to organize your SAS work. But project files do have a drawback: using the project "locks up" content inside the file, and you can view the content only by opening the project in the SAS Enterprise Guide interface.

The automation model provides methods to extract all types of content from your projects, including: ODS results (such as HTML and RTF), data, SAS programs, and SAS logs. This example shows how to reach into the project and pull all of the SAS programs into a folder on your local computer.

The following is a portion of the example, which has been trimmed for space considerations. You can find the complete VBScript program by visiting the sites listed in the References section.

```
Const egCode = 1
Const egData = 2
Const egFile = 23
Const flowContainer = 0
Dim item
Dim flow
For Each flow In Project.ContainerCollection
  If flow.ContainerType = flowContainer Then
    WScript.Echo "Process Flow: " & flow.Name
    For Each item in flow.Items
```

```
        Select Case item.Type
        Case egFile
          WScript.Echo "  " & item.Name & ", File Name: " & item.FileName
        Case egCode
          WScript.Echo "  " & item.Name & ", SAS Program"
             WScript.Echo "    saving program content to: " _
                & programsFolder & "\" & item.Name & ".sas"
             item.SaveAs programsFolder & "\" & item.Name & ".sas"
             If Err.Number <> 0 Then
               WScript.Echo "     ERROR: There was an error while saving " _
                 & programsFolder & "\" & item.Name & ".sas"
               Err.Clear
             End If
        Case egData
          WScript.Echo "  " & item.Name & ", Data: " & item.FileName
        End Select
      Next
    End If
  Next
```

Here are a few highlights of this example:

- The SAS Enterprise Guide project contains one or more process flows, which are called `Container` objects in the automation model. To iterate through all of the content, the script examines each `Container` in the `Project.ContainerCollection`, processing only those where the `ContainerType` indicates a process flow.
- Each process flow contains a collection of items. This script examines each item, stopping only to deal with items of certain types: `File`, `Data`, and `Code`.
- The `ContainerType` and item `Type` are enumerated values. Each value is a number that is mapped to a type constant. In VBScript it is difficult to access the actual value mappings, so they are repeated in this example script by using the Const statement. Using the type constant instead of the number helps to make the script easier to read. You can find the enumerated sequence in the scripting reference documentation. (For example, see the topic for "SASEGItemType Enumeration".)
- When an item of type `egCode` is encountered, the program uses the `Code.SaveAs` method to save the content of the program to a local folder. Note that the `SaveAs` method is similar to selecting **File->Export->Export Code** in the SAS Enterprise Guide application. It exports not only the core program, but any automatically generated macro functions and values that SAS Enterprise Guide uses. This helps to ensure that the exported program works the same in any environment that you use to run it.

Here is an example command to run the script, with a partial example of the output:

```
C:\Examples> cscript ExtractCode.vbs c:\DonorsChoose\DonorsChoose.egp

  Opening project: c:\DataSources\DonorsChoose\DonorsChoose.egp
  Process Flow: Data Quality
    ProjectCost, Data: WORK.PROJECTCOST
    ProjectCostDetails, Data: WORK.PROJECTCOSTDETAILS
    COSTLESSTHANZERO, Data: WORK.COSTLESSTHANZERO
    Data Imported from donorschoose-org-1may2011-v1-resources.csv, Data: DC.RESOURCES
    Data Imported from donorschoose-org-1may2011-v1-projects.csv, Data: DC.PROJECTS
    P99, Data: WORK.P99
    OUTLIERS, Data: WORK.OUTLIERS
    OneProject, Data: WORK.ONEPROJECT
    Distribution, SAS Program
      saving program content to:
          c:\DataSources\DonorsChoose\DonorsChoose_programs\Distribution.sas
  Process Flow: Autoexec
    Assign DC library, SAS Program
      saving program content to:
          c:\DataSources\DonorsChoose\DonorsChoose_programs\Assign DC library.sas
```

SCRIPTING WITH WINDOWS POWERSHELL, BY EXAMPLE

The Windows PowerShell scripting engine is built into the most recent versions of Microsoft Windows, including Windows 7 and Windows 2008 Server. If you still have a few older Windows XP machines kicking around, you can download a compatible version of PowerShell from Microsoft.

If you are familiar with UNIX shells (such as Korn shell or its variants), you will probably be very comfortable with Windows PowerShell. Just like its UNIX predecessors, Windows PowerShell allows you to run commands and combinations of commands from an interactive console window. You can also write PowerShell scripts (saved as PS1 files), which allows you to combine the commands and programming logic to run more sophisticated operations.

You can run PowerShell commands in several ways:

- PowerShell has a command-line console (similar to a DOS prompt) where you can run commands and script files.
- PowerShell has a development environment, called the Interactive Scripting Environment (ISE), which allows you to write scripts, run commands, and see the script output. SAS users will find it familiar, because it is a little bit like a traditional SAS Display Manager session.
- You can create script files with a .PS1 file extension, and run them by invoking the `powershell` command.

ENABLING POWERSHELL TO RUN

Here is the most baffling part about getting started with PowerShell: by default, you cannot run PowerShell scripts on your system. It's a capability that comes as *disabled* out-of-the-box. You can run script commands from the console, but you will not be able to execute scripts that are saved as PS1 files. If you try, you will see an error message with text similar to this:

```
File C:\Test\TestScript.ps1 cannot be loaded because the
  execution of scripts is disabled on this system.
  Please see "get-help about_signing" for more details.

At line:1 char:23+ .\Test\TestScript.ps1 <<<<
  + CategoryInfo
 : NotSpecified: (:) [], PSSecurityException
 + FullyQualifiedErrorId : RuntimeException
```

Presumably , this default policy setting is for your own safety. Fortunately, you can easily change the policy by using the `Set-ExecutionPolicy` command:

```
Set-ExecutionPolicy RemoteSigned
```

Run this command from the PowerShell console, select 'Y' to confirm, and you will now be able to run local PS1 files on your PC. ("RemoteSigned" indicates that local scripts will run, but scripts that you download from the Internet will run only if they are signed by a trusted party.) You can read more about setting these policies for running scripts in the *Windows PowerShell Owner's Manual*. (See References.)

EXAMPLE 1: LIST AVAILABLE PROFILES WITH POWERSHELL

The following PowerShell script invokes the SAS Enterprise Guide automation object and lists the available metadata profiles, similar to the VBScript example that was presented earlier in this paper.

```
$eguideApp = New-Object -comObject SASEGObjectModel.Application.4.3
Write-Host $eguideApp.Name  $eguideApp.Version
Write-Host "Profiles defined for:" $env:UserName
foreach ($profile in $eguideApp.Profiles())
{
  Write-Host "Profile: Name=" $profile.Name
  Write-Host "  Host=" $profile.HostName "Port=" $profile.Port
}
```

Similar to the VBScript example, we create the COM object for the automation interface. In PowerShell, this is accomplished with the `New-Object` command and the `-comObject` option.

This script writes its output to the console, which is the same as "STDOUT" in typical shell programming terminology. The output looks like this:

```
PS C:\Examples> ListProfiles.ps1
   Enterprise Guide 4.3.0.0
   Profiles defined for: sascrh
   Profile: Name= My Server
     Host= l73391.na.sas.com Port= 8561
   Profile: Name= Null Provider
     Host= Port=
   Profile: Name= t2817
     Host= t2817 Port= 8562
   Profile: Name= uitsrv02
     Host= uitsrv02.na.sas.com Port= 8561
   Profile: Name= uitsrv04
     Host= uitsrv04.na.sas.com Port= 8561
```

EXAMPLE 2: USE WINDOWS POWERSHELL TO PROCESS A COLLECTION OF PROJECTS

One of the powerful features of PowerShell is its ability to pipe the output of one process into the input of another process. This allows you to perform complex operations with "one-liner" scripts.

The following script processes a collection of SAS Enterprise Guide project files, piped in from the output of the `ls` command (which is the PowerShell version of the DOS `dir` command).

```
$eguideApp = New-Object -comObject SASEGObjectModel.Application.4.3

# use the special $input variable to pipe file objects into the script
# for processing
if ( @($input).Count -eq 0)
{
  Write-Host "Usage: ls <directoryOfProjects>\*.egp | eg43ProjectContents.ps1"
  Exit -1
}
else
{
  #reset the $input enumerator
  $input.Reset()
 }
foreach ($projName in $input)
{
    # Open the project file
    Write-Host " ---------------------------------------------- "
    Write-Host "Examining project:" $projName
    $project = $eguideApp.Open("$projName", "")

    # Show all of the process flows in the project
    $pfCollection = $project.ContainerCollection
    Write-Host "Process flows within project:"
    foreach ($pf in $pfCollection)
    {
      if ($pf.ContainerType -eq 0)
      {
           Write-Host "  " $pf.Name
           foreach ($item in $pf.Items)
           {
             # report on each item within the process flow
             Write-Host "    " $item.Name " (" $item.GetType() ")"
             if ($item.GetType().ToString() -eq "SAS.EG.Scripting.Data")
             {
               # report on each task attached to this DATA item
               foreach ($task in $item.Tasks)
                  {
                    Write-Host "      " $task.Name  " (" $task.GetType() ")"
                  }
             }
```

```
            }
        }
    }

    Write-Host " ------------------------------------------------------ "
    # Show all of the data references within the project
    $dataCollection = $project.DataCollection
    Write-Host "ALL Data used within project:"
    foreach ($dataItem in $dataCollection)
    {
      Write-Host "  " $dataItem.FileName " on server " $dataItem.Server
    }

    # Show all of the code items embedded within the project
    $codeCollection = $project.CodeCollection
    Write-Host "ALL SAS programs embedded within project:"
    foreach ($codeItem in $codeCollection)
    {
      Write-Host "  """ $codeItem.Name """ runs on server " $codeItem.Server
    }

    # Close the project file
    $project.Close()
}
# Quit (end) the application object
$eguideApp.Quit()
```

Here is an example of the partial output that this script produces:

```
PS C:\Examples> ls C:\DataSources\DonorsChoose\*.egp | .\eg43projectContents.ps1
  Examining project: C:\DataSources\DonorsChoose\DonorsChoose.egp
  …
  ALL SAS programs embedded within project:
    " Assign DC library " runs on server  SASApp
    " Optimize Data (LONG!) " runs on server  SASApp
    " Assign Library " runs on server  SASApp
    " Distribution " runs on server  SASApp
    " Assign Library " runs on server  SASApp
    " Import Essays " runs on server  SASApp
```

CUSTOM APPLICATIONS WITH MICROSOFT .NET

The VBScript and Windows PowerShell scripting engines can serve as a sort of "glue" to adhere a series of automated processes together, including those processes that do work in SAS Enterprise Guide. In contrast, Microsoft .NET offers a platform for building entire applications--complete with sophisticated user interfaces and business logic. By using the SAS Enterprise Guide automation model, you can include SAS work into such an application without exposing an end user to the complexities of SAS.

Note: This discussion assumes that you are familiar with Microsoft .NET and Microsoft Visual Studio as a development tool. If you want to learn more about that topic, Microsoft offers many resources on their Web site: http://msdn.microsoft.com/beginner/.

ADDING SAS ENTERPRISE GUIDE AUTOMATION TO YOUR VISUAL STUDIO PROJECT

As shown back in Figure 1, the automation API is surfaced in a single .NET assembly: SASEGScripting.dll. To use the automation API in your .NET application, you must add a reference to this assembly from within your Microsoft Visual Studio project.

However, while the SASEGScripting.dll contains the API for your use, it does not contain *all* of the classes and structures that are used during automation; there are dozens of other .NET assemblies (DLLs) that are used along the way. SASEGScripting.dll can be thought of as a shim – a gateway to all of the internal workings of SAS Enterprise Guide.

To ensure that your application can work effectively on any machine with SAS Enterprise Guide installed, with a minimum amount of "hassle" in distribution, you need to use a helper class that:

- detects where SAS Enterprise Guide is on the machine, and
- redirects assembly "load" events so that the proper SAS Enterprise Guide assemblies are located and loaded into the process when needed.

An example of this helper class is provided in with the resources for this paper. The example is written using C# and is in the file `SEG43AssemblyResolver.cs`.

To create a Microsoft .NET project that can access the automation API, follow these steps in Microsoft Visual Studio:

1. Create a new project of any type, appropriate for your objective (for example, a Windows Forms application, Console application, or Windows Presentation Foundation application).

   Note: SAS Enterprise Guide 4.3 requires .NET Framework 3.5 or later. You can base your application on .NET 3.5 or .NET 4.0, but not on an earlier version of .NET.

2. In your Visual Studio project, add a reference to SASEGScripting.dll. You will find the DLL file in the SAS Enterprise Guide application directory. (You can also add the SAS Enterprise Guide folder as a referenced path for use in your project.)

3. Change the properties of the reference so that **Copy Local** is set to **False**, as shown in Figure 3. This will prevent the SASEGScripting.dll (and all other assemblies that it references) from being copied to your output directory when you build the application.

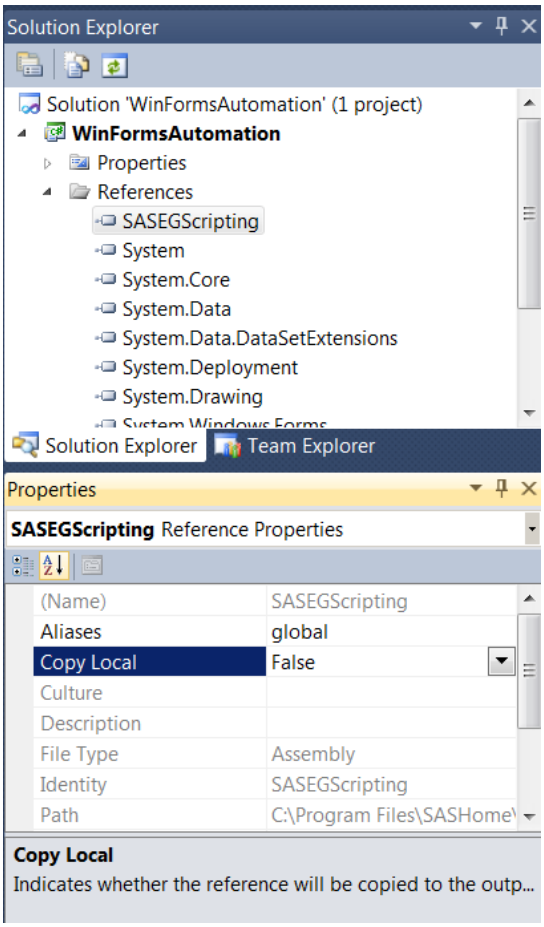

Figure 3. Properties of Assembly Reference in Visual Studio

4. Add the SEG43AssemblyResolver and EGAutomation helper classes to the project. (These are provided as SEG43AssemblyResolver.cs and EGAutomation.cs in the online resources for this paper.)

5. In the Main() function for the application, add the following statement:

```
SAS.EG.Automation.SEG43AssemblyResolver.Install();
```

This function must occur early in the program, before any other calls to the automation API. Figure 4 shows an example of a project with the class files added and the statement placed in the Main() function.

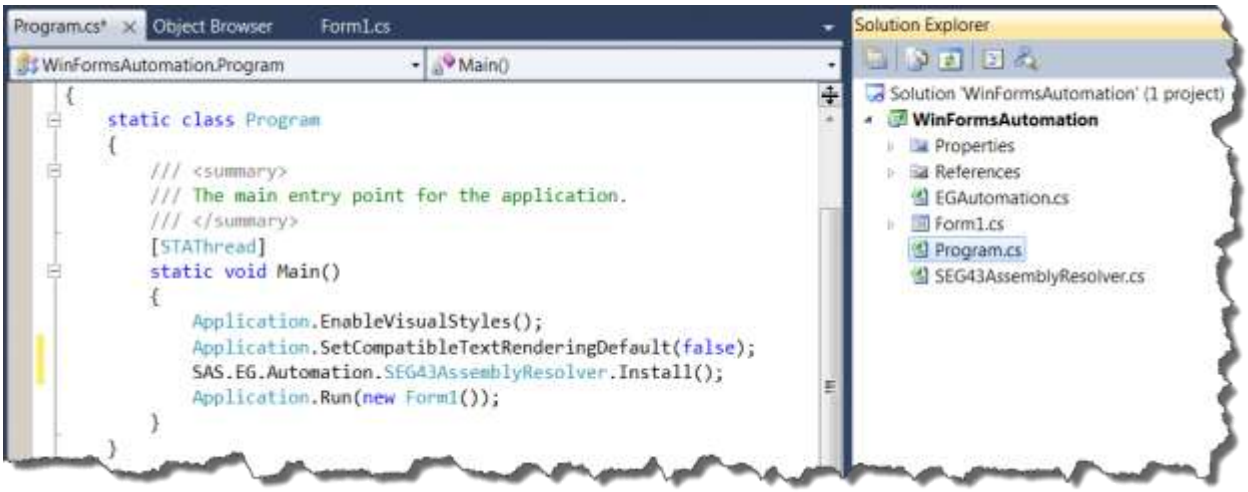

Figure 4. Helper Classes and Assembly Resolver Function Call

6. In the Project Properties, Build settings, be sure to select the platform target as x86 (for 32-bit SAS Enterprise Guide).
7. Build the project and verify that there are no errors.

These steps create a project that is ready for the SAS Enterprise Guide automation API. The EGAutomation helper class contains methods that make certain automation operations easier: creating an Application object, opening a project file, finding certain types of elements within the project, running a process flow, and more.

The online resources for this paper (see References) include a full example of a Visual Studio project (using C#) to drive SAS Enterprise Guide automation. You can open and build the project with Microsoft Visual Studio or Microsoft Visual C# Express Edition.

CONCLUSION

Automation allows you to run your business processes unattended, either by driving them with higher-level applications or scheduling scripts to run during off-peak times. This paper presents just a few ideas of the tasks you can accomplish by automating SAS Enterprise Guide. With a little bit of scripting knowledge and familiarity with the automation APIs, you can automate almost every aspect of SAS and SAS Enterprise Guide.

REFERENCES

SAS Enterprise Guide *Scripting Reference*, http://support.sas.com/eguide

Windows PowerShell Owner's Manual, http://technet.microsoft.com/en-us/library/ee221100.aspx

The SAS Dummy Blog (http://blogs.sas.com/sasdummy), select articles:

- Blogs about automation: http://blogs.sas.com/content/sasdummy/tag/automation/
- Blogs about Windows PowerShell: http://blogs.sas.com/content/sasdummy/tag/powershell/

sasCommunity.org site for this paper, with links to complete examples:
http://www.sascommunity.org/wiki/Not_Just_for_Scheduling:_Doing_More_with_SAS_Enterprise_Guide_Automation

ABOUT THE AUTHOR

Chris Hemedinger is a principal technical architect in SAS Professional Services. Chris is also a co-author of *SAS For Dummies*. Contact Chris at:

Chris Hemedinger
chris.hemedinger@sas.com
http://blogs.sas.com/sasdummy
Twitter: http://twitter.com/cjdinger