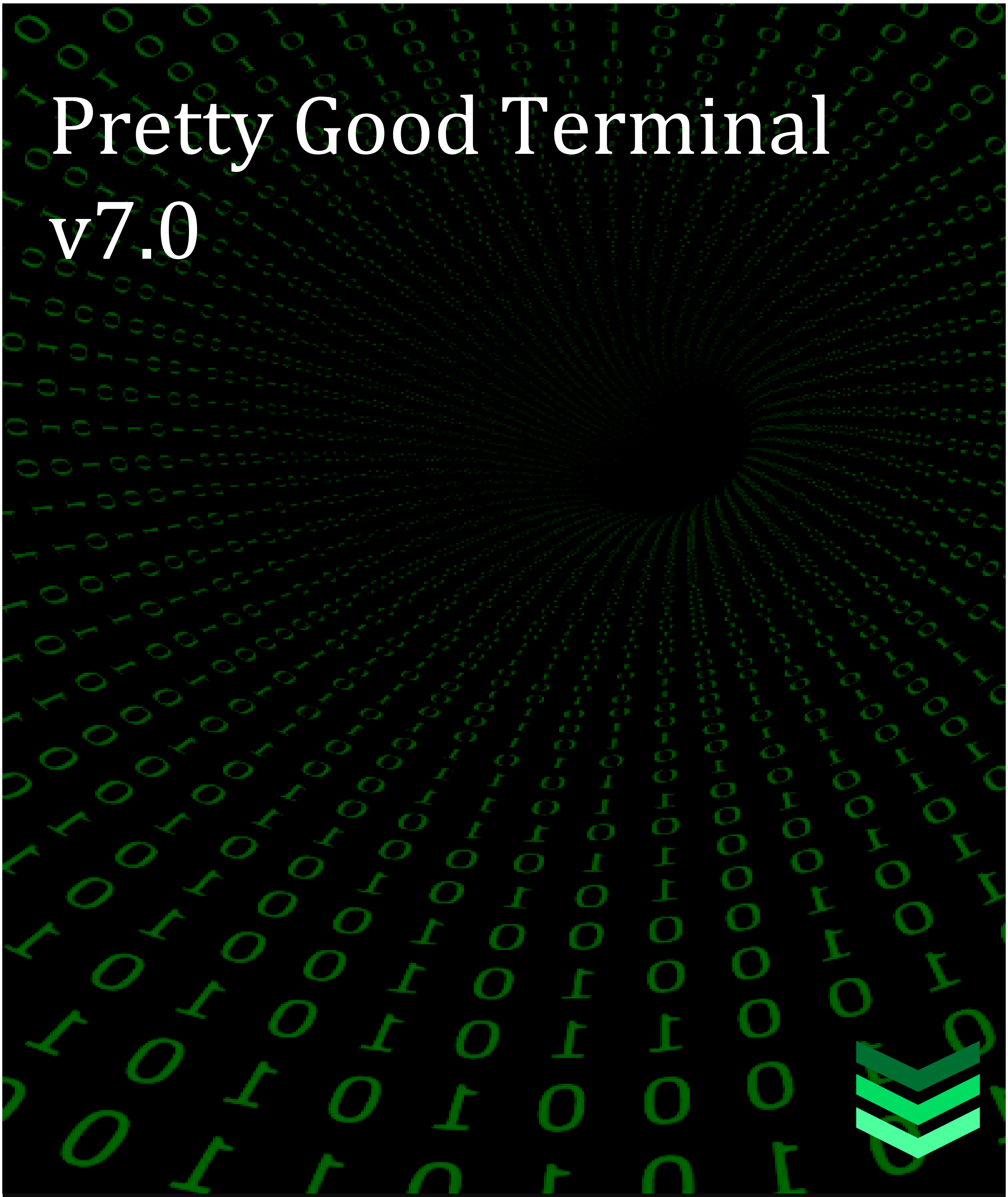


Pretty Good Terminal v7.0

Laszo Frank

Developer's reference

9/13/2016

1 CONTENTS

2 WARNING

This software – Pretty Good Terminal – is a very powerful application designed to change the configuration of lots of networking devices. Be aware, that the software is not able to determine the validity and the effect of commands it is executing. It is always the sole responsibility of the user using the application. Issuing the wrong commands may result in serious damage to the network and business.

Always use the software on your own risk !

3 PREAMBLE

This documentation focuses on the development support provided by the Professional Edition of Pretty Good Terminal.

PGT was written to provide an extensible framework for developers to write simple VBS scripts or create and run custom code for complex tasks

- VBS scripting support provides a means of automation. One can call various function of PGT from a vbs script, for instance using Excel. Using Excel, the script can easily be debugged.

- A more sophisticated approach of extensibility is creating a class library and implementing some basic interfaces. In this context, the base framework handles the script to connect to hosts as described and then transfers execution to the CustomActionHandler class in the class library. This CustomActionHandler object can then call all the built-in functions of the internal ScriptableTTYTerminal or NetconfTerminal classes to interact with the connected device. The custom action handler can perform any addition tasks as required, such as building an inventory database.

4 CHANGES FROM PREVIOUS VERSIONS

4.1 CHANGES FROM V6.2

PGT v7.0 introduced Netconf protocol support. To integrate this protocol into existing code a common base interface IScriptableSession was introduced and the former ScriptableTerminal class became ScriptableTTYTerminal. Implementing IScriptableSession the NetconfTerminal class was also introduced. Both of these classes are the implementation of the base IScriptableSession interface and therefore the formal Executor.STerminal or Terminal variables were renamed to Executor.Session and Session variables while the same services are offered by them as earlier.

4.2 CHANGES FROM V6.0

There are two important changes to the development interface in v6.2 release.

1. PGT v6.2 introduced the concept of user extensible keywords in terminal logon process. Keywords can be assigned a LogonResult value and a display message. This change required that the internal data structure returned by STerminal.LogonToHost() be also changed. The new structure, LogonResultEx, is detailed below.

2. To make scripting easier, ExecCommand() and GetFullPrompt() members of ScriptableTerminal class does not return a Boolean value whether a timeout occurred, but a CommandTimoutException is thrown instead. Also, ExecCommand can be called without passing prompt parameter. In this case prompt is determined automatically based on the last used connection parameters.

4.3 CHANGES FROM V5.0

ICustomActionHandler interface was modified and the Finalize() member renamed to Terminate(), as finalize could possibly conflict with object dispose. Existing libraries needs to be modified only by renaming implementation methods from Finalize() to Terminate().

4.4 CHANGES FROM V4.5

Visual Script execution engine demanded that more parameters are supplied to ICustomactionHandler.DoCustomAction method. This is achieved in v5.0 by the extension of DeviceConnectionInfo class with two fields:

```
public bool inPrivilegedMode;
public string VendorName;
```

You can find more information about this class in chapter 6.4.1 DoCustomAction method.
vScript advanced scripting is discussed in chapter Contents

4.5 CHANGES FROM V4.4

With the introduction of public key authentication for SSH connections the ConnectionParameters class has been extended to include AuthType and IdentityFile fields. For details refer to chapter 6.7

4.6 CHANGES FROM V4.3

The development interface has been changed mainly to resolve versioning issues. As PGT evolved and new versions were released, all of the previously written plugins had to be recompiled with the updated references. This is because when a referenced library version changes, the dependent dll can't be loaded.

To resolve this issue, the PGT interface definitions were moved to a separate dll – namely pgtinterfaces.dll – and the version of this library does not change among releases unless there really is a change in the interface definition.

This solution works as long as a plugin - a Custom Action Handler – only references the pgtinterfaces.dll. Of course, if the developed plugin has references to other PGT libraries (such as common.dll) this won't help.

So for existing plugins, there really is no big change with this release, only former references pointing to ScriptingEngine.dll should be updated to PGTINterfaces.dll.

And one more thing : the required runtime version has changed to .NET 4.0 !

5 COM OBJECT BASED SCRIPTING

The easiest way of development is through using VBS/VBA scripting. PGT supports this functionality by registering the ScriptingEngine.dll with COM. To do that, one must register this DLL by using the Windows built-in RegAsm.Exe utility :

- To register scripting COM objects use the following command :

  C:\Windows\Microsoft.NET\Framework\v4.0.30319\RegAsm/tlb /codebase " c:\program files\PGT\STerminal.dll"

  <u>The location of ScriptingEngine.dll file must be specified by using full path.</u>

- To unregister scripting COM objects :

  C:\Windows\Microsoft.NET\Framework\v4.0.30319\RegAsm/unregister /codebase " c:\program files\PGT\STerminal.dll"

You will find two batch files in the PGT installation directory with the commands.

After registering STerminal.dll with COM, one can use it like the following example :

```
Sub ParseList()
  Dim PGT
  Set PGT = CreateObject("PGT.STerminal")
  Jump = Cells(1, 2)
  j = PGT.ConnectToJumpServer("telnet", Jump, "cisco", "cisco", "#", "Password", "", “Cisco”)
  If j Then
    For Each c In Range("devices")
      DeviceIP = c.Value
      Dim LogonResult
      Set LogonResult = PGT.LogonToHost("telnet", DeviceIP, "cisco", "cisco", "cisco", "")
      If LogonResult.lResult = 0 Then
        Dim Command As String
        Command = ActiveSheet.Cells(c.Row, 2).Value
        Dim CommandResult As String
        CommandResult = PGT.ExecCommand(Command)
        ActiveSheet.Cells(c.Row, 3).Value = CommandResult
      Else: MsgBox "Unable to connect to host"
      End If
    Next
    MsgBox "Script finished"
  Else: MsgBox "Unable to connect to jump server"
  End If
End Sub
```

PGT.STerminal object supports the following functions:

- **bool ConnectToJumpServer(string Protocol, string DeviceIP, string Username, string Password, string ServerPrompt, string AuthType, string KeyFileName, string Vendor)**
  - Connects to the specified jump server with the credentials supplied over the given protocol. Protocol can be Telnet,SSH2 or SSH2. Valid AuthTypes are *Password* or *PublicKey* In case of Public Key authentication KeyFileName should point to the key file
- **object LogonToHost(string Protocol, string DeviceIP, string Username, string Password, string EnablePassword, string CurrentPrompt, string Vendor = "Cisco")**
  - Log on to the specified device with the credentials supplied. It can be called after ConnectToJumpServer, or directly if the host is reachable directly from the computer where script is running.For Vendor, the default value is Cisco.
- **bool LogOff(string Vendor, string DeviceIP, string WaitPrompt, bool WaitForResponse)**
  - Logs off from the currently connected host by sending the appropriate command depending on Vendor. Only waits for answer when WaitPrompt if specified and WaitForResponse is set
- **string ExecCommand(string Command, string waitPrompts)**
  - Execute the command and waits for the prompts specified. Multiple possible prompts must be separated by semicolon. Throws an exception in case the command execution times out.
- **string ExecCommand(string Command)**

  - The same as the previous except with waitPrompts omitted. In this case, PGT use the wait prompts as defined in Vendor settings. The Vendor information is taken from the last call to Connect() or LogonToHost()
- `int WaitForPrompt(int timeoutSeconds, string Prompts, string ExitKeywords = null, int SearchFrom = 0)`
  - Waits for the specified Prompts and returns the index of prompt found. Multiple possible prompts must be separated by semicolon. The returned value is the zero based index of the matched prompt in the list of prompts. In case of timeout -1 is returned. **ExitKeywords** is a list of strings which possibly indicate an error and are searched among the terminal lines. Multiple items must be separated by semicolon. If any of the listed string is found, the return value is calculated as 1000 plus the zero based index of the item in the list. The SearchFrom parameter specifies the sequence number of the terminal line from where to start the search for any of the **Prompts** or **ExitKeywords**. Prompts are always – and only – checked in the last terminal line, while ExitKeywords are searched for the entire text. The SearchFrom variable can be set to the value returned by a previous call to SendText()
- `int WaitForText(int timeoutSeconds, string Text, int SearchFrom)`
  - Waits the specified amount of seconds or until the specified Text can be found in Terminal output. The Searchfrom parameter specifies the sequence number of the terminal line from where to start search for the Text. The SearchFrom variable can be set to the one returned by a previous call to SendText()
- `int SendText(string Text, bool AutoLineFeed = True)`
  - Sends the specified text to the connected device but does not wait for answer. Should be combined with WaitForPrompt to wait and detect answer from remote device. The return value is the sequence number of the terminal line in which the command was sent in.AutoLineFeed controls whether a NewLine character is added automatically to the end of the command.
- `string GetHostName(string PromptDelimiter)`
  - Returns the hostname of the device. PromptDelimiter should be the character ending the prompt, like # or >

LogonToHost returns a LogonResultEx object :

```
public class LogonResultEx
{
  public LogonResult lResult;
  public string Message;
  public object Tag;
}
```

The value of the lResult variable of the returned object corresponds to the following :

Value	Acronym	Description / Message (Message variable can be customized by user)
-1	Timeout	No response from host
0	Success	
1	Fail	Primary control channel (Telnet/SSH) could not be established to host or jump server
2	TelnetTimeout	Telnet/SSH timeout from the jump server to the host
3	Method Not Supported	Enable password was supplied but vendor is not "cisco"
4	AAA Rejected	TACACS authentication error
5	Connection Refused	Connection refused by remote host
6	Opened But Timed out	Connection was opened, but no response
7	Closed By Remote Host	Connection method not accepted
8	Enable Access Denied	Enable access was rejected or timed out after issuing "enable" command, but before sending password
9	Enable Access Failed	Enable password was not accepted for some reason
10	Login Invalid	Logon Username/Password was not accepted
11	Connection Timeout	Telnet/SSH connection attempt returned timeout message
12	ResendUsername	The user name was re-requested after sending password. Could be wrong credentials.
13	Password Rejected	When only password is required but the given password is not accepted
14	DeviceUnreachable	Device is unreachable

Please note that the COM interface object – PGT.STerminal – needs to be licensed in order to function. The same license is used for this method as for PGT application.

It is also worth noting, that the COM interface does not support different PGT profiles, and hence settings within PGT. All calls will use the last loaded – or actual if you like – profile settings. As profile settings define a lot of connection parameters it can be vital to select the correct profile from PGT.

If it is inevitable to change or designate the profile from a VBA script at runtime, PGT's user.config file can be tweaked in the *%appdata%\Local\Laszlo_Frank\PrettyGoodTerminal.exe_StrongName_xyz\7.0.a.b* folder. It contains the ConfigProfileName setting that defines the profile name to be used.

6 .NET DEVELOPMENT SUPPORT

For complex scripting tasks, such as changing device configuration based on either its current configuration or external parameters one must write custom code. This custom code can then be interfaced with PGT to provide robust, quick and intuitive scripting solution.

6.1 PREREQUISITES

As PGT is using signed assemblies, it can only load signed assemblies. Please make sure the class library you create as a custom action handler is signed.

It is also a requirement, that the custom action handler assembly must be using .Net framework 3.5 or *earlier*, or otherwise PGT will not be able to load it.

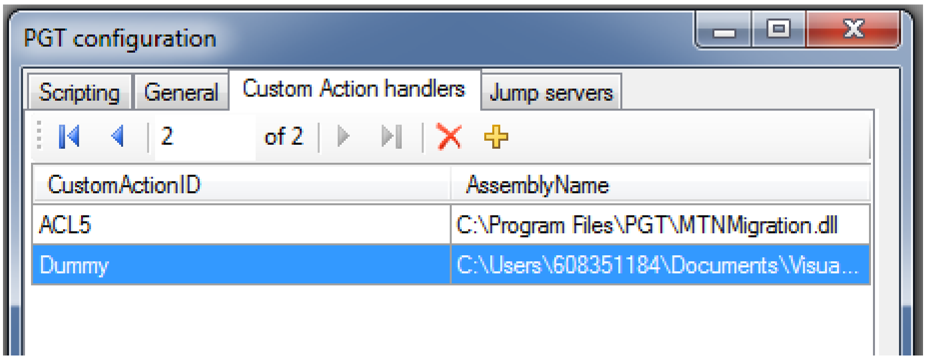
6.2 CUSTOM ACTION HANDLERS

The concept of PGT remains the same : connect to devices and execute commands line-by-line. But what is actually executed on a device after a connection was successfully made can be complex, decision driven instructions. Sometimes building some custom database from the devices capability and/or configuration is also required.

To help with this situation, PGT introduces the idea of CustomActionHandlers. A CustomActionHandler is a piece of code usually implemented by a class and compiled into a class library. This class can then be instantiated at runtime and called whenever a connection was made to a device. To identify which class to instantiate and transfer execution to, a script line can include a custom tag named CustomActionID. This ID can be any string identifying the class – or more precisely the functionality inside a class – to be called.

Actually, a CustomActionHandler class is queried against the CustomActionID-s it supports, and this way the CustomActionID is eventually mapped to a class. Later chapters and code samples will lighten this more.

Custom action handler libraries can be added by browsing for the class library. Custom action IDs handled by that library will be extracted and shown :



CustomActionID	AssemblyName
ACL5	C:\Program Files\PGT\MTNMigration.dll
Dummy	C:\Users\608351184\Documents\Visua...

6.3 CUSTOM ACTION ID

This field is used for two purposes:

1. To specify a directive on how to process a command. More than one directive may be listed, separated by the pipe "|" character. Directives are always evaluated before command execution. Valid directives are :

    - **NOWAIT** : after issuing the specified command, the program will NOT wait for the device prompt to return before proceeding to the next script item
    - **NORTIM** : normally commands are trimmed, that is, leading and trailing spaces are stripped off. There might be occasions (such as configuring banner text) when these spaces must remain untouched. Then use this directive.
    - **WAITFOR *number*** : execution will be paused for this much time expressed in milliseconds after the command was executed.
    - **WAITFOR *text*** : after executing the command, execution will be paused until the given text is found in the terminal output text, or until command timeout expires.
    - **IGNOREERROR** : directs the scripting engine to ignore any error condition detected by searching for a command failure pattern expression in the command response. See script settings for the details of command failure detection.

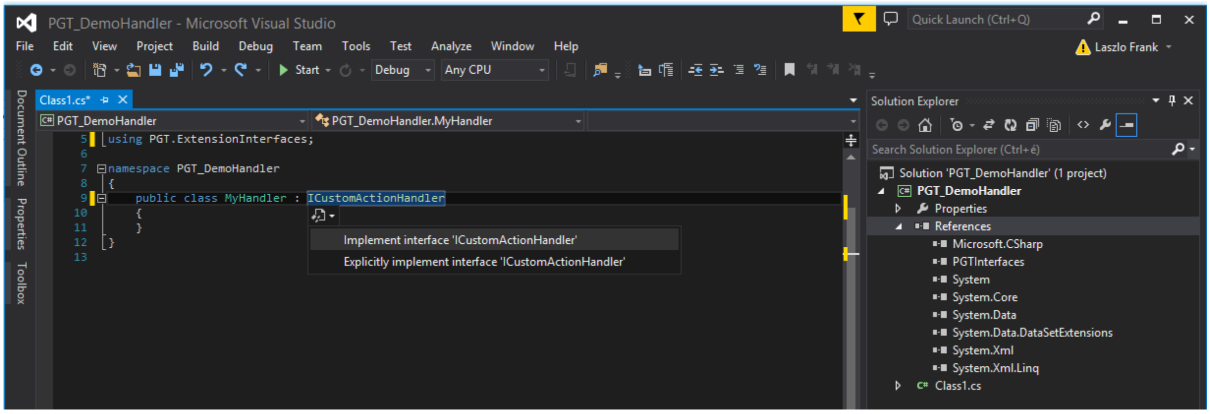
    WAITFOR *number* and WAITFOR *text* must not be present at the same time in a certain script line.

2. To specify a custom action identifier. This ID is mapped to a class library containing the external code to handle this line in the script. This is used when a complicated logic must be implemented. In these cases PGT establish the connection to the end device and then calls the specified custom action handler. These IDs are actually provided by the custom action handler class and are extracted from the class library.

6.4 WRITING A CUSTOM ACTION HANDLER

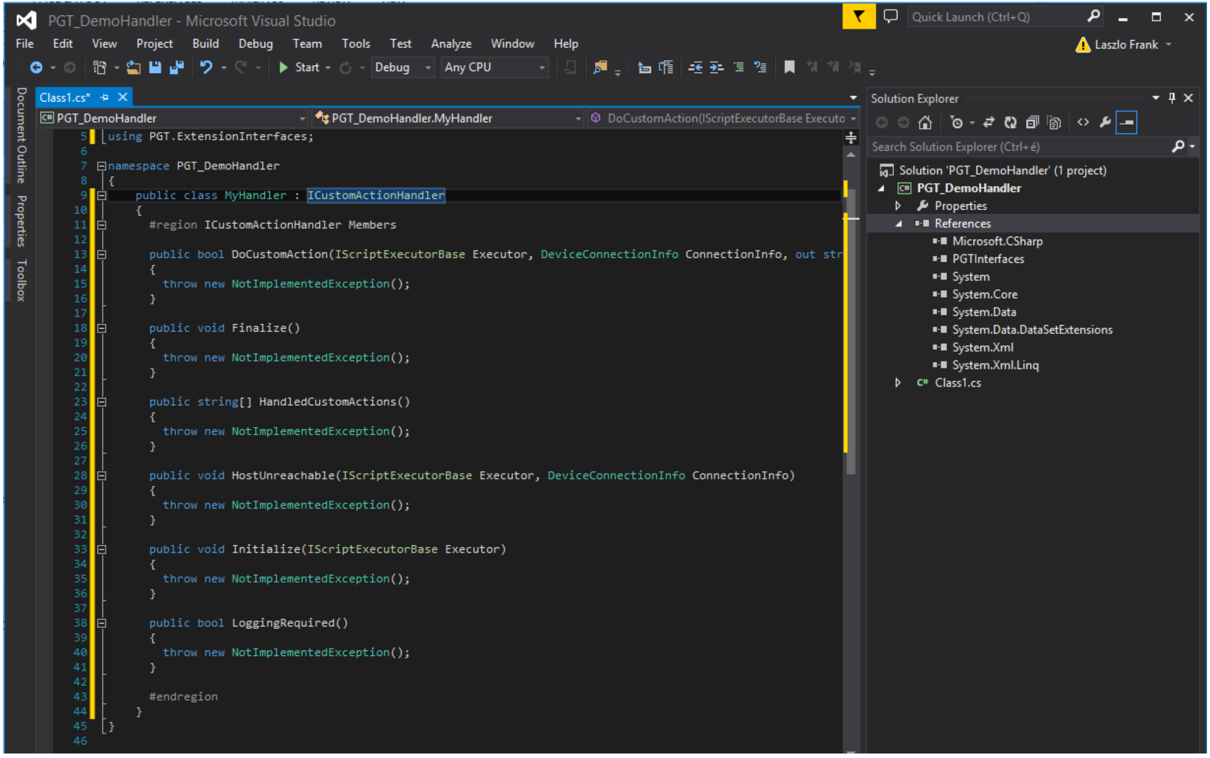
The concept is that PGT is going to connect to each host device as specified by the input script, and afterwards transfers execution to the registered custom action handler class by invoking the ICustomActionHandler interface implemented by that class.

Using Visual Studio one can create a new class library project and add a reference to PGTInterfaces.dll. The .NET target framework of the project must not be newer than 4.0, otherwise PGT will not load the assembly.
Through the reference set to PGTInterfaces.dll, the PGT.ExtensionInterfaces namespace can be accessed and used. This namespace contains the ICustomActionHandler interface, which must be implemented by a class.



```
using PGT.ExtensionInterfaces;

namespace PGT_DemoHandler
{
    public class MyHandler : ICustomActionHandler
    {
    }
}
```

Now, by clicking the blue underscore and the Implement interface menu item on ICustomActionHandler, a skeleton code will be generated with all the interface methods which need to be implemented :



```
using PGT.ExtensionInterfaces;

namespace PGT_DemoHandler
{
    public class MyHandler : ICustomActionHandler
    {
        #region ICustomActionHandler Members

        public bool DoCustomAction(IScriptExecutorBase Executor, DeviceConnectionInfo ConnectionInfo, out str
        {
            throw new NotImplementedException();
        }

        public void Finalize()
        {
            throw new NotImplementedException();
        }

        public string[] HandledCustomActions()
        {
            throw new NotImplementedException();
        }

        public void HostUnreachable(IScriptExecutorBase Executor, DeviceConnectionInfo ConnectionInfo)
        {
            throw new NotImplementedException();
        }

        public void Initialize(IScriptExecutorBase Executor)
        {
            throw new NotImplementedException();
        }

        public bool LoggingRequired()
        {
            throw new NotImplementedException();
        }

        #endregion
    }
}
```

By implementing the **ICustomActionHandler** interface these methods are created automatically.

- **Initialize** is called only once, when the class is first instantiated
- **DoCustomAction** is called for each line in the script
- **HostUnreachable** is called when a host cannot be reached
- **LoggingRequired** is called by PGT to determine whether a log file is required by this class. This log file can be set through Tools/Options.
- **Terminate** is called once the script was finished
- **HandledCustomActions** must return the list of custom action ids handled by this class. The list must be separated by semicolons. These values must not match any reserved words as described in the CustomActionID filed (such as WAITFOR, NOWAIT...)

One library should contain only one class implementing the ICustomActionHandler interface. If the library contains more classes of ICustomActionHandler, only the first one – as returned by reflection – will be queried. But a single class can handle several Custom Action IDs, and implement the appropriate action inside DoCustomAction based on the actual value of the ID passed as a parameter.

So, a very basic handler looks like this :

```
using System;
using System.Collections.Generic;
using System.Text;
using PGT.ExtensionInterfaces;

namespace PGT_DemoHandler
{
  public class MyHandler : ICustomActionHandler
  {
     public bool DoCustomAction(IScriptExecutorBase Executor, DeviceConnectionInfo ConnectionInfo,
       out string ActionResult, out bool ConnectionDropped, out bool BreakExecution)
    {
      ActionResult = "I was called";
      ConnectionDropped = false;
      BreakExecution = false;
      return true;
    }
    public void Terminate()
    {
    }
    public string[] HandledCustomActions()
    {
      return new string[]{"MyID"};
    }
    public void HostUnreachable(IScriptExecutorBase Executor, DeviceConnectionInfo ConnectionInfo)
    {
    }
    public void Initialize(IScriptExecutorBase Executor)
    {
    }
    public bool LoggingRequired()
    {
      return false;
    }
  }
}
```

As this class return “MyID” as the CustomActionID, it would be called for any line in a script where the line contains “MyID” in the CustomActionID column. For each of these lines the DoCustomAction method would be called, if a connection was successfully made to the host.

6.4.1 DOCUSTOMACTION METHOD

This method should contain the script logic and it will be executed for each line of the script where the CustomActionID references this class.

The connection information passed to the call contains vital information about the current connection and is contained by the `DeviceConenctionInfo` parameter, which class is defined as :

```
public class DeviceConnectionInfo
{
	public string JumpList;
	public string DeviceIP;
	public string HostName;
	public string Command;
	public string CustomActionID;
	public bool inPrivilegedMode; <- new in v5.0
	public string VendorName; <- new in v5.0
	public ConnectionProtocol Protocol;
}
```

Where the ConnectionProtocol is :

```
public enum ConnectionProtocol { Unknown, Telnet, SSH1, SSH2, Netconf, None };
```

Once gets called, DoCustomAction can communicate with the connected device through the passed Executor interface. This is an implementation of the `IScriptExecutorBase` interface and has the following members :

```
public interface IScriptExecutorBase
{
	IscriptableSession Session { get; }
	void ShowActivity(string Text);
	bool WriteLogEntryEx(string logStr, bool TimeStamp = false);
	void TerminateScript();
}
```

Here, the most important member is Session which can be a class of `ScriptableTTYTerminal` for Telnet and SSH connections and `NetconfTerminal` for netconf session. It has many methods to be called to interact with the connected device. `ScriptableTTYTerminal` encapsulates a virtual VT100 terminal connected to the end host through the list of jump servers. In other words, this is the terminal of the end host. (Refer to chapter 6.7 for details)

Interacting with a device through a character based terminal line is usually consists of sending a command and waiting for the answer. The terminal can identify a received answer by waiting for the terminal prompt. This can be a complex string, or just a single character. For instance, a connected Cisco router or switch in privileged mode has the # prompt terminator character. This is usually adequate to identify the prompt. Internally, the PGT scripting engine handles the full prompt of the device, or rather a stack of prompts as it goes through the jump servers, but here, connected to the end host it is enough to identify a prompt by awaiting the # character.

Said that, Session can be used to send a command and get back the result or wait for a prompt . Here is a simple code excerpt identifying the VPN ID the device is routing for :

```
#region VPN ID check
string command = "sh run | in router bgp";
Executor.ShowActivity(command);
string commandResult = Executor.Session.ExecCommand(command, "#");
Executor.ShowActivity(commandResult);
int h_VPNID = commandResult.IndexOf("65200") >= 0 ? 11 : commandResult.IndexOf("65100") >= 0 ? 10 : -1;
```

```
#endregion
```

The Executor.ShowActivity() will display the given string in the action pain of the Script Executor form. The call `commandResult = Executor.Session.ExecCommand(command, "#");` will execute the given command on the device, wait for the prompt of "#" and then return the command result as a string.

Any given handler has the possibility to configure a device directly, or, alternatively, may want to create a configuration script file which can later be loaded into PGT and executed.

For instance, the following code will write to the specified log file to create a configuration script :

```
Executor.WriteLogEntryEx(string.Format("{0},{1},{2},{3},{4},{5},{6},{7},{8},{9},{10}",   "1",   JumpList,   "cisco",
DeviceIP, HostName, "telnet", "", "yes", "conf t", "", ""), false);
```

The above example illustrates how to generate a single line of a script file, which can later be loaded by PGT. The WriteLogEntryEx() method can write text into a log file. This is the log file specified in Tools/Options in PGT, and this setting is different for each opened Script Executor. It is important to remember, that WriteLogEntryEx() will always succeed, even if there was no log file specified. This is the reason why the CustomActionHandler has the LoggingRequired member, which must return a Boolean value indicating if a log file should be specified for this handler. If LoggingRequired returns true, but the user did not select a log file, PGT will warn the user and asks if the script execution should be continued or not

When using WriteLogEntryEx() it is not necessary to open or close the log file, it is handled automatically by the scripting engine. If the log file already exists, PGT will prompt the user whether to append or overwrite the file.

6.4.1.1 RETURNED VALUES OF DOCUSTOMACTION

The Boolean return value of DoCustomAction will be used to mark a script line as success or error. The text returned in ActionResult will be copied to the script line CommandResult column.

Sometimes the action performed inside DoCustomAction disconnects the terminal line of the devices (such as reload/reset command). To notify the scripting engine that the connection was dropped on purpose, set the value of ConnectionDropped to true, otherwise return false.

When a serious error happens during the execution of DoCustomAction, it is possible to signal PGT that script execution should be terminated and so the execution engine should not pick the next line of the script. This is possible by setting the BreakExecution return value to true.

6.4.2 ABOUT INITIALIZE / TERMINATE

A CustomActionHandler class is instantiated only once for a given running script. If more Script Executor windows are opened, the class will be instantiated separately by each Script executor engine. Engines have a distinctive GUID identifiers to which the object is bounded while alive.

The instantiation occurs the first time a call is needed to the handler class. On this occasion the handler class should initialize any local variables it will use during the script execution. In this way, the Initialize is very similar to a constructor. Of course, the class's constructor can also be used for this purpose. The only difference is, that Executor parameter will be passed as a parameter to Initialize, and as a result WriteLogEntryEx() may be called from here to write a header to a log file for instance.

When the script is finished or terminated, the Terminate member will be called. At this time, any resources initialized in the call to Initialize should be handled appropriately (like closing files, writing to database, and so on).

6.5 PROVIDE A USER INTERFACE FOR A HANDLER

Sometimes CustomActionHandlers are quite complex, maybe handling databases, too. Whatever the reason is, it might be necessary to provide a custom user interface for managing a particular handler class. For this purpose, PGT introduces the `ICustomMenuHandler` interface. The purpose of this interface is to insert a menu item into the Action main menu. The interface is very simple, and has only two members :

```
public interface ICustomMenuHandler
{
  /// <summary>
  /// This is the main menu item of the plugin
  /// </summary>
  /// <returns></returns>
     ToolStripMenuItem GetMenu();
  /// <summary>
  /// PGT will pass over the reference to application main form.
  /// </summary>
  /// <param name="mainForm"></param>
  void SetMainForm(Form mainForm);
}
```

The menu is what it is : the main menu structure to be displayed on the Action menu. All menu items should have its associated event handlers set.

PGT will call SetMainForm() to pass a reference for the application's main form. It can be used to create a child MDI form for instance.

Once built, Custom Menu Handlers must be registered with PGT in Tools/Options.

6.5.1 AN EXAMPLE INTERFACE

The following example shows a complete implementation of ICustomMenuHandler for a plugin.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace PGT.SQLInterface
{
  class PGTInterface : ICustomMenuHandler
  {
    #region Fields
    private Form AppMainForm;
    System.ComponentModel.BackgroundWorker _workInProgress;
    #endregion

    #region ICustomMenuHandler Members

    public System.Windows.Forms.ToolStripMenuItem GetMenu()
    {
      ToolStripMenuItem tsmMainMenu = new ToolStripMenuItem();
      ToolStripMenuItem tsmSaveToDatabase = new ToolStripMenuItem();
      ToolStripMenuItem tsmDataManager = new ToolStripMenuItem();
      ToolStripMenuItem tsmConfigure = new ToolStripMenuItem();
      ToolStripSeparator tss1 = new ToolStripSeparator();

      tsmMainMenu.DropDownItems.AddRange(new System.Windows.Forms.ToolStripItem[] { tsmConfigure, tss1,
tsmSaveToDatabase, tsmDataManager });

      #region Menu definition
      //
      // tsmSQLInterfaceMainMenu
      //
      tsmMainMenu.Image = PGT.SQLInterface.Resource1.color_wheel_16xLG;
      tsmMainMenu.ImageTransparentColor = System.Drawing.Color.Black;
      tsmMainMenu.Name = "PGT.SQLInterface.tsmMainMenu";
      tsmMainMenu.Text = "Scripting Projects";
      //
      // tsmConfigure
      //
      tsmConfigure.Image = Resource1.ManageCounterSets_8769;
      tsmConfigure.ImageTransparentColor = System.Drawing.Color.Black;
      tsmConfigure.Name = "PGT.SQLInterface.tsmConfigure";
      tsmConfigure.Text = "Configure module";
      tsmConfigure.Click += tsmConfigure_Click;
      //
      // tsmSQLInterfaceMainMenu
      //
      tsmSaveToDatabase.Image = Resource1.build_Selection_16xLG;
      tsmSaveToDatabase.ImageTransparentColor = System.Drawing.Color.Black;
      tsmSaveToDatabase.Name = "PGT.SQLInterface.tsmSaveToDatabase";
      tsmSaveToDatabase.Text = "Save result to database";
      tsmSaveToDatabase.Click += tsmSaveToDatabase_Click;
      //
      // tsmDataManager
      //
      tsmDataManager.Image = Resource1.Guage_16xLG;
      tsmDataManager.ImageTransparentColor = System.Drawing.Color.Black;
      tsmDataManager.Name = "PGT.SQLInterface.tsmDataManager";
      tsmDataManager.ShortcutKeys = ((System.Windows.Forms.Keys)((System.Windows.Forms.Keys.Control | 
System.Windows.Forms.Keys.P)));
      tsmDataManager.Text = "Manage projects";
      tsmDataManager.Click += tsmDataManager_Click;
      #endregion
      return tsmMainMenu;
    }
    void tsmConfigure_Click(object sender, EventArgs e)
    {
      (new SqlConnectionEditor()).ShowDialog();
    }

    void tsmSaveToDatabase_Click(object sender, EventArgs e)
    {
      // First try to get the reference to the scripting form's ScriptManager whose result should be saved
      ScriptManager _ScriptManager = ScriptingFormManager.GetActiveScriptingFormManager();
      if (_ScriptManager != null) (new SaveToDatabase()).ShowDialog();
      else MessageBox.Show("The active window is not a Script Executor. Please select the Script Executor window 
before using this function.", "Unable to save results", MessageBoxButtons.OK, MessageBoxIcon.Warning);
    }

    void tsmDataManager_Click(object sender, EventArgs e)
    {
      _workInProgress = new System.ComponentModel.BackgroundWorker();
      _workInProgress.WorkerSupportsCancellation = true;
      _workInProgress.DoWork += DoWorkAnimation;
      _workInProgress.RunWorkerAsync();
      try
      {
        ScriptingProjectManager sMgr = new ScriptingProjectManager();
        sMgr.Show();
        sMgr.MdiParent = AppMainForm;
      }
      finally
      {
        _workInProgress.CancelAsync();
      }
      //sMgr.WindowState = FormWindowState.Maximized;
    }
    public void SetMainForm(System.Windows.Forms.Form mainForm)
    {
      AppMainForm = mainForm;
    }

    private void DoWorkAnimation(object sender, System.ComponentModel.DoWorkEventArgs e)
    {
      WorkInProgressAnimation L = null;
      DateTime waitStartedAt = DateTime.Now;
      while (true)
      {
        System.Threading.Thread.Sleep(20);
        if ((DateTime.Now - waitStartedAt).TotalMilliseconds > 500)
        {
          if (L == null)
          {
            L = new WorkInProgressAnimation("Loading module, please wait", "Loading");
            L.Show();
          }
        }
        Application.DoEvents();
        if (_workInProgress.CancellationPending)
        {
          if (L != null) L.Close();
          break;
        }
      }
    }

    #endregion
  }
}
```

6.6 AUTOMATE SCRIPT CREATION

PGT works fine with CSV and Excel files until a certain number of devices. When one has to work with several thousands of devices, managing Excel tables will get overwhelming. I came across this problem when had to script more than 45k devices, and realised that a solution must be made to circumvent the Excel-hell.

The solution is to create an interface for PGT, through which scripts can be generated programmatically from a database for instance. If we already has the capability to include and display a custom UI form within PGT, why not to let this form – basically external code – to create the scrip, run the script, evaluate the results and update a database based on that.

This is a very effective way of handling several thousands of devices while maintain an always up-to-date status about them.

As a result PGT provides the following functionality :

1. Open Script Executors programmatically
2. Populate a Script Executor with script
3. Run the script
4. Get back the results
5. Close the Script Executor window

To enable all of the above functionality in a class library, an additional references must be added to PrettyGoodTermal.exe. As PGT itself does not have references to the external class libraries containing the CustomActionHandler or CustomMenuHandler classes, adding a reference to the main executable will not impose a circular reference problem.

6.6.1 OPENING A SCRIPT EXECUTOR

Scripts in PGT are displayed on `ScriptingForm` forms. This is the form displayed when the user clicks on Action/New Script Executor menu item. The Form is responsible to load, display and start or stop (pause/resume) the execution of a script. This functionality of the form is managed by the `ScriptManager` internal class of the `ScriptingForm`.

To manage this kind of form from code, a `ScriptingFormManager` static class exists in the PGT namespace. This class can be used to query for opened `ScriptingForm` windows and also to open a new one :

- `public static List<ScriptManager> GetScriptManagers()` member will return a list of `ScriptManager` for all opened `ScriptingForm`

- `public static ScriptManager OpenNewScriptingForm()` will open a new `ScriptingForm` and return the reference for its `ScriptManager`

Once a reference to an `ScriptManager` is obtained, it is possible to programmatically create a script and execute it.

6.6.2 CREATING AND EXECUTING A SCRIPT

To create a script the `ScriptManager` class can be used, once obtained through a call to `ScriptingFormManager`.*GetScriptManagers()* or `ScriptingFormManager`.*OpenNewScriptingForm()*. This class is declared as :

```
namespace PGT
{
  public abstract class ScriptManager
  {
    /// <summary>
    /// Returns the name of the script
    /// </summary>
    /// <returns></returns>
    public abstract string GetScriptName();
    /// <summary>
    /// this will close the Scripting Form
    /// </summary>
    /// <param name="NotifyUser"></param>
    public abstract void CloseForm(bool NotifyUser = true);
    /// <summary>
    /// This notifies the form that the script was saved externally
    /// </summary>
    public abstract void SetScriptSaved();
    /// <summary>
    /// Clear the script lines
    /// </summary>
    public abstract void ClearItems();
    /// <summary>
    /// Before adding multiple script lines, BeginAddEntries should be called for performance reasons
    /// </summary>
    public abstract void BeginAddingEntries();
    /// <summary>
    /// Must be called after finished adding multiple entries
    /// </summary>
    public abstract void EndAddingEntries();
    /// <summary>
    /// Extends the standard(default) ListView columns with extra columns from the passed header.
Should be called before adding entries
    /// </summary>
    /// <param name="Header"></param>
    public abstract void UpdateHeader(string[] Header);
    /// <summary>
    /// Add a single script line
    /// </summary>
    /// <param name="scriptLine"></param>
    /// <returns></returns>
    public abstract bool AddEntry(string scriptLine, ScriptLineState sls =
ScriptLineState.Undetermined, Color? ForeColor = null, Color? BackColor = null);
    /// <summary>
    /// Returns all script lines
    /// </summary>
    /// <returns></returns>
    public abstract List<ScriptListViewItem> GetItems();
    /// <summary>
    /// Starts script execution
    /// </summary>
    /// <param name="NotifyUser"></param>
    public abstract void ExecuteScript(bool NotifyUser = true);
    /// <summary>
    /// Stops script execution
    /// </summary>
    public abstract void StopScript();
    /// <summary>
    /// Pauses the script execution
    /// </summary>
    public abstract void PauseScript();
    /// <summary>
    /// Resumes script execution
    /// </summary>
    public abstract void ResumeScript();
    /// <summary>
    /// Returns the text content of the ActionPane
    /// </summary>
    public abstract string GetActionPane { get; }
    /// <summary>
    /// Returns the text content od the TerminalPane
    /// </summary>
    public abstract string GetTerminalPane { get;}
    /// <summary>
    /// ScriptFinished will be called when the script execution was finished
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    public delegate void ScriptFinished(object sender, ScriptEventArgs e);
    public abstract event ScriptFinished OnScriptFinished;
    /// <summary>
    /// ScriptAborted will be called when the script execution was canceled or the form was closed
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    public delegate void ScriptAborted(object sender, ScriptEventArgs e);
    public abstract event ScriptAborted OnScriptAborted;
  }
}
```

Once a reference retrieved to the `ScriptManager` of the `ScriptingForm`, it is possible to add lines to the script as strings programmatically. The format of the string representing a script line is exactly the same as the format of the input CSV file, and must contain the following items separated by colon:

Order	Name
0	Selected (1 or 0)
1	JumpServerList
2	Vendor
3	DeviceIP
4	HostName
5	ConnectionProtocol
6	Command
7	PrivilegedModeRequired
8	ReconnectRequired
9	RegExSearch
10	CustomActionID

Example line to add: “1,192.168.1.1,cisco,172.16.0.1,,telnet,conf t,yes,,,”

So creating a script is very easy, and consist of the following three steps :

1. Acquire a reference to a `ScriptManager` by either enlisting the opened `ScriptingForm` windows or opening a new one;
2. Add script lines by calling `ScriptManager.AddEntry();`
3. Call `ScriptManager.ExecuteScript();`

The generated script may contain additional columns beyond the required columns as necessary. In this case the example string above can be extended like :

“1,192.168.1.1,cisco,172.16.0.1,,telnet,conf t,yes,,,itemIDValue”

Before adding such a line, it must be signalled that the header is not the standard one. For this purpose use the *UpdateHader()* member, like this :

```
// Get the standard headers list
string sExtenderHeader = PGT.Common.Helper.ArrayToString(Enum.GetNames(typeof(InputFileHeader)),
sepChar);
// Extend it with one extra column
sExtenderHeader += "Item ID";
// Notify the scriptManager about extended columns
_ScriptManager.UpdateHeader(sExtenderHeader.Split(sepChar.ToCharArray(),StringSplitOptions.RemoveEmpty
Entries));
```

If many lines are to be added, the *BeginAddingEntries()* should be called before and *EndAddingEntries()* at the end. These members encapsulates the *ListView.BeginUpdate()* and *EndUpdate()* methods for the same purpose.

Once script is created, it can be executed by calling the *ExecuteScript()* method.

To get notified when the script finished successfully or was aborted, use the *OnScriptFinished and OnScriptAborted* event handlers. The event handler’s sender parameter will be set to the `ScriptManager` instance sending the event. Using this sender it is possible to close the form fr instance.

6.6.3 GETTING THE RESULTS

After the script finished, the results can be retrieved by calling `ScriptManager.GetItems().` This call will return a list of `ScriptListViewItems` corresponding to the script results as can be seen in the `ScriptingForm` which executed the script. Each `ScriptListViewItem` item contains the information in its SubItems, which can be reached via the `ScriptListViewItem.SubItems` collection. In order to easily access these SubItems by the indexer property of the collection, the following helper enumeration can be used :

```
public enum LVISubItem {JumpServerList, Vendor, DeviceIP, HostName, ConnectionProtocol, Command,
PrivilegedModeRequired, ReconnectRequired, RegExSearch, CustomActionID, RegexResult, CommandResult };
```

Using the enum, one can refer for instance to DeviceIP of a `ScriptListViewItem` like this :

```
string thisDevice = lvi.SubItems[(int)LVISubItem.DeviceIP].Text;
```

where the lvi variable is a `ScriptListViewItem`.

Each item also has a LogonResult property, the type of LogonResult enumeration :

```
public enum LogonResult {Success, Fail, TelnetTimeout, MethodNotSupported, TACACSRejected,
ConnectionRefused, OpenedButTimedout, ClosedByRemoteHost, EnableAccessDenied, EnableAccessFailed,
LoginInvalid, ConnectionTimeout, ResendUsername, PasswordRejected };
```

The meaning of the above items are :

Value	Description
Success	Success
Fail	Primary control channel (Telnet/SSH) could not be established to host or jump server
TelnetTimeout	Telnet/SSH timeout from the jump server to the host
Method Not Supported	Enable password was supplied but vendor is not "cisco"
TACACS Rejected	TACACS authentication error
Connection Refused	Connection refused by remote host
Opened But Timed out	Connection was opened, but no response
Closed By Remote Host	Connection method not accepted
Enable Access Denied	Enable access was rejected or timed out after issuing "enable" command, but before sending password
Enable Access Failed	Enable password was not accepted for some reason
Login Invalid	Logon Username/Password was not accepted
Connection Timeout	Telnet/SSH connection attempt returned timeout message
Password Rejected	When only password is required but the given password is not accepted
DeviceUnreachable	When no response was received to ping

Besides examining the `LogonResult` value, each item also contains the `CommandResult` subitem, which can be checked against the returned text

Furthernore, a `ScriptLsitViewItem` has the IsSuccess property of type :

```
public enum ScriptLineSuccess { Undetermined, Success, Error};
```

Using this property it can be checked whether the execution of the script line succeeded or not. The value of Undetermined means that the line was not executed.

6.7 USING PGT AS AN SSH/TELNET LIBRARY

Another option of development is to use PGT as a high level SSH/Telnet library. This means that only limited functionality is exposed at a high level hiding protocol details completely.

When a reference is added to STerminal.dll, the `ScriptableTTYTerminal` class can be used from the PGT namespace. This class has many methods focusing on functionality rather than protocol details.

At the very basic level, the `ScriptableTTYTerminal` class can establish connection to a host and then send commands to the device. You can directly connect to a device, or through a jump server. In the latter case, the initial connection is made to the jump server by calling *ConnectoToJumpServer()*, and then the final device with *LogonToHost()*. Generally, you should never call *Connect()* method directly, because connections are handled by *ConnectToJumpServer()* and *LogonToHost()* methods. Both methods checks if a connection has already been established or not. If not, calls *Connect()* internally to establish the network connection to the device specified. If yes, however, uses the JS_TelnetCommand or JS_SSHCommand properties to connect to the specified device over an existing terminal conenction. This is equivalent of typing the command "telnet 10.0.0.1" on a terminal to connect to host 10.0.0.1.

Likewise, if multiple jump servers are chained, *ConnectoToJumpServer()* should be called for each connection. For instance, if one has the following setup :

PGT computer → Jump Server (10.0.0.1) → Jump Server(192.168.1.1) → EndHost (172.16.1.1)

Then the following calls should be made on PGT computer to reach EndHost :

3. *ConnectToJumpServer(10.0.0.1);*
4. *ConnectoToJumpServer(192.168.1.1);*
5. *LogonToHost(172.16.1.1);*

Below you can see the member methods with a description of their function:

```
public sealed class ScriptableTTYTerminal
{
    /// <summary>
    /// Creates a new ScriptableTerminal instance
    /// </summary>
    /// <param name="TerminalScreenLength"></param>
    public ScriptableTTYTerminal(int TerminalScreenLength)
    /// <summary>
    /// The command to be issued on the jump server in order to connect to a device via Telnet
    /// </summary>
    public string JS_TelnetCommand { get; set; }
    /// <summary>
    /// The command to be issued on the jump server in order to connect to a device via SSH
    /// </summary>
    public string JS_SSHCommand { get; set; }
    /// <summary>
    /// Specifies the general timeout value used for wait operations after sending commands
    /// </summary>
    public TimeSpan DefaultCommandTimeout { get; set; }
    /// <summary>
    /// Specifies the general timeout value used for wait operations when connecting via telnet or ssh
    /// </summary>
    public TimeSpan DefaultConnectTimeout { get; set; }
    /// <summary>
    /// Returns the connection state
    /// </summary>
    public bool IsConnected{ get;}
    /// <summary>
    /// Connects to a host as specified by  ConnectionParameters
    /// </summary>
    /// <param name="CD"></param>
    /// <returns>Returns true if connection succeeds</returns>
    public bool Connect(ConnectionParameters CD)
    /// <summary>
    /// Connects to a host as specified by the parameters
    /// </summary>
    /// <param name="CD"></param>
    /// <returns>Returns true if connection succeeds</returns>
    public bool Connect(string Protocol, string DeviceIP, string Username, string Password)
    /// <summary>
    /// Sends the text to the terminal connection.
    /// </summary>
    /// <param name="Text">The text to be sent</param>
    /// <returns>Returns the terminal screen line number in which the text is sent</returns>
    public int SendText(string Text)
    /// <summary>
    /// Sends a NewLine char sequence, based on the terminal's TransmitNL seettings
    /// </summary>
    public void SendNewLine(string Text)
    /// <summary>
    /// Executes a single command. Wait prompt is determined automatically based on the current connection settings.
    /// The prompt is awaited before returning or a CommandTimeoutException is thrown.
    /// </summary>
    /// <param name="Command">The command to execute. Multiple commands must be separated by semicolons. </param>
    /// <returns>Returns the command result text</returns>
    public string ExecCommand(string Command, string waitPrompts)
    /// <summary>
    /// Executes a single command. If a prompt is passed, it also waits for the prompt before returning.
    /// The prompt is awaited before returning or a CommandTimeoutException is thrown.
    /// </summary>
    /// <param name="Command">The command to execute. Multiple commands must be separated by semicolons. </param>
    /// <param name="waitPrompts">If set, waits for any of the prompts before returning. MUST be set to NULL if no
    wait operation is expected </param>
    /// <returns>Returns the command result text</returns>
    public string ExecCommand(string Command, string waitPrompts)
    /// <summary>
    /// Executes multiple commands. Between commands waits for any of the given prompts.
    /// CommandTimeoutException can be thrown in case of timeout error.
    /// </summary>
    /// <param name="commands">An array of commands</param>
    /// <param name="Prompts">An array of acceptable prompt</param>
    /// <returns>Returns the command result text</returns>
    public string ExecCommand(string[] commands, string waitPrompts)
    /// <summary>
    /// Waits for the given prompts. The prompt is always verified only in the last terminal line !
    /// </summary>
    /// <param name="timeout">Waits this much time for prompts at most. Minimum is 2 seconds</param>
    /// <param name="Prompts">The valid prompts to wait for. Multiple prompts must be separated by semicolon</param>
    /// <param name="exitConditionStrings">Aborts waiting for prompts if any of the listed strings detected in the
    response</param>
    /// <returns>Returns the index of the prompt found, or 1000 + the index of the exit condition string. If wait
    operation times out, returns -1</returns>
    public int WaitForPrompt(TimeSpan timeout, string Prompts, string ExitConditionString = null, int SearchFrom = 0)
    /// <summary>
    /// Waits for the given prompt. The prompt is always verified ine the last terminal line !
    /// </summary>
    /// <param name="timeout">Waits this much time for prompts at most</param>
    /// <param name="Prompts">The valid prompts to wait for. Multiple prompts must be separated by semicolon</param>
    /// <param name="exitConditionStrings">Aborts waiting for prompts if any of the listed strings detected in the
    response</param>
    /// <param name="searchFrom">Prompts and exitConditionStrings are searched beyond this line number only</param>
    /// <returns>Returns the index of the prompt found, or 1000 + the index of the exit condition string. If wait
    operation times out, returns -1</returns>
    public int WaitForPrompt(TimeSpan timeout, string[] Prompts, string[] exitConditionStrings = null, int searchFrom
    = 0)
    /// <summary>
    /// Waits for the given text. The text is searched among the terminal lines starting at the specified line
    number.
    /// </summary>
    /// <param name="timeout">Waits this much time before timing out. Minimum is 2 seconds, smaller values will be
    ignored</param>
    /// <param name="Text">The texts to wait for. The text may contain semicolons as a text separator if multiple
    texts may be expected.</param>
    /// <param name="searchFrom">The terminal line number to start the search at </param>
    /// <returns>Returns the index of the text found, or -1 if no text was found or timed out</returns>
    public int WaitForText(TimeSpan timeout, string Text, int searchFrom)
    /// <summary>
    /// Pings a device and returns true if it is reachable
    /// </summary>
    /// <param name="MaxWait">wait no more time for an answer. Minimum is 2 seconds, smaller values will be
    ignored</param>
    /// <param name="commandtemplate">the ping command template containing %DEVICEIP placeholder. If empty, the
    default is to use ping %DEVICEIP</param>
```

```
    /// <param name="deviceIP">what to ping</param>
    /// <param name="waitprompt">waits for this prompt before and after ping</param>
    /// <param name="WaitUntilRechable">if true, waits until a response is received at most for MaxWait</param>
    /// <returns></returns>
    public bool TestHostReachability(TimeSpan MaxWait, string commandtemplate, string deviceIP, string waitprompt,
    bool WaitUntilRechable = false)
    /// <summary>
    /// Retrieves the hostname of the currently connected device. If the prompt contains other stuff, that will be
    returned !
    /// <param name="promptDelimiter">The character terminating the prompt, like $ or #</param>
    /// </summary>
    /// <returns></returns>
    public string GetHostName(string promptDelimiter)
    /// <summary>
    /// Retrieves the current prompt including the hostname, such as R1#
    /// CommandTimeoutException might be thrown.
    /// <param name="promptDelimiter">The character terminating the prompt, like $ or #</param>
    /// </summary>
    /// <returns></returns>
    public string GetFullPrompt(string PromptDelimiter)
    /// <summary>
    /// Logs on to a device via telnet or ssh1/2 protocol. Also enters to priviledged mode if needed.
    /// </summary>
    /// <param name="CD">Parameters required for the connection</param>
    /// <param name="inPrivilegedMode">In case of a cisco device returns true if we are in privileged mode</param>
    /// <returns>Returns the logon result</returns>
    public LogonResultEx LogonToHost(ConnectionParameters CD, out bool inPrivilegedMode)
    /// <summary>
    /// Depending on the current connection status calls ConnectToPrimaryJumpserver or ConnectToSecondaryJumpserver
    /// </summary>
    /// <param name="Protocol">The connection protocol text. Valid values are Telnet, SSH1 or SSH2</param>
    /// <param name="DeviceIP">The IPv4 address of the host to connect to</param>
    /// <param name="Username">The logon username</param>
    /// <param name="Password">The logon password</param>
    /// <param name="ServerPrompt">The jump server prompt</param>
    /// <returns></returns>
    public bool ConnectToJumpServer(string Protocol, string DeviceIP, string Username, string Password, string
    ServerPrompt)
    /// <summary>
    /// Disconnects the current communication channel.
    /// </summary>
    /// <returns>Returns true if it was connected before, false otherwise</returns>
    public bool Disconnect()
}
```

Where ConnectionParameters is :

```
  public class ConnectionParameters
  {
    public ConnectionProtocol Protocol;
    public string DeviceIP;
    public string DeviceVendor;
    public string LogonUserName;
    public string LogonPassword;
    /// <summary>
    /// should be specified for devices having elevated exec mode (like enable mode on cisco)
    /// </summary>
    public string EnablePassword;
    /// <summary>
    /// the prompt expected
    /// </summary>
    public string ServerPrompt;
    /// <summary>
    /// the current prompt (from where we are connecting, mainly for secondary connections)
    /// </summary>
    public string CurrentPrompt;
    /// <summary>
    /// Used to further parametrize telnet command. The format of the final command sent to the device is constructed
    /// as JumpServer.Telnetcommand ConnectionParameters.DeviceIP ConnectionParameters.TelnetParameters
    /// </summary>
    public string TelnetParameters;
    /// <summary>
    /// Used to further parametrize ssh command. The format of the final command sent to the device is constructed
    /// as JumpServer.SSHcommand ConnectionParameters.DeviceIP ConnectionParameters.SSHParameters
    /// </summary>
    public string SSHParameters;
    /// <summary>
    /// Selects reuired authentication type
    /// </summary>
    public TermAuthType AuthType;
    /// <summary>
    /// When public key authentication is required, this is name of the private key file
    /// </summary>
    public string IdentityFile;
  }

public enum TermAuthType { Password, PublicKey };
```

The ultimate goal of TelnetParameters and SSHPArameters is to make it possible to send a formatted command like "telnet 192.168.1.2 /vrf purple" to a host for example. Used by ScriptableTTYTerminal.LogonToHost()

6.8 ACCESSING PGT'S SETTINGS

When executing business login in scripts, it is sometimes necessary to access properties set in Tools/Options, like DeviceUsername or DevicePassword. For this purpose, the static class `PGTSettingsManager` can be used from the PGT.Common namespace. To access this namespace, a reference must be added to Common.dll.

For instance, `PGTSettingsManager`.GetCurrentScriptSettings() returns the settings for the currently active profile.

7 PYTHON DEVELOPMENT SUPPORT

Starting with version 6.0, PGT supports running and debugging Python scripts. PGT can run Python scripts 3 different ways :

- from Python script files loaded on demand in the background
- interactively, using the Python Interactive window
- as Visual Script files (see chapter 8, Visual Script Development)

Python script development is fully covered by the users' manual.

8 VISUAL SCRIPT DEVELOPMENT

In this chapter I want to show you the Visual Scripting capability of Pretty Good Terminal.

Although simple vScripts does not require any programming skills, vScripts provide a general infrastructure to include user defined c# classes, variables and code, even referencing and using external assemblies.

In fact, visual scripts in PGT are nothing else than CustomActionHandlers - described in details in previous chapters – which are built and compiled at runtime by the vScript engine. Under the hood, the vScript execution engine is a c# compiler using CodeDOM and Reflection to provide an intuitive code editor with intelli-sense like controls, where users can enter their own c# code.

I will to show you the object model used and how the script execution engine works. If you understand these topics, you will be able to create sophisticated and complex scripts.

8.1 THE CONCEPT

Most of the time scripts - and humans, too :-) - repeat the following basic steps:
- check a device configuration element by issuing show commands
- analyse the response
- based on the result construct a configuration command

Of course, one step is rarely enough to decide what to do and more configuration checks are required until enough information is collected to build the final configuration change.

The best way to design a script is to represent the required steps visually and organizing them to a flowchart. This is exactly what you can do with PGT's Visual Script Editor. You can add visual script elements and connect them to create a flowchart. Then each element will have its own code executed at runtime, using its own local variables or script global variables to memorize command results. Then connectors again have their own code which is evaluated to decide about the control flow direction, that is, which is the next step to be taken.

A visual script - or vScript - must have a single start element and may have many stop elements. Execution of the vScript starts whenever PGT made a successful connection to a device as specified in a script (not the vScript, but the legacy script which is actually a list a devices along with connection parameters). Then PGT will pass the execution to the vScript.

Let me show you what a vScript is.

8.2 CREATING A SIMPLE SCRIPT

The best way to understand what a vScript is and how it operates is to discuss through a simple example (*you can find the more detailed description of this task in the users's guide*). For this reason let us assume we have a list of routers and we need to update the dialer interface only if :

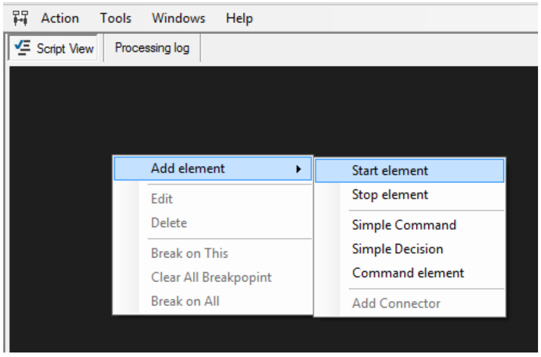
- It is a Cisco router
- Belongs to a specific BGP AS
- The dialer if bandwidth equals to 128

Without vScript, using only the conventional, CLI commands driven simple scripts this would be a challenging task. However, with vScript it is very simple, straightforward and does not even involve any programming.

Now let's see how it works, how to build and use the script.

Just open a new Visual Script Editor from the Actions menu of PGT. If there is any default script presented, select all elements by pressing Ctrl-A - or select with the mouse - and press delete to clear the workspace.

First of all, we need a Start Element. Right click the workspace and from the Add elements menu select the Start element:

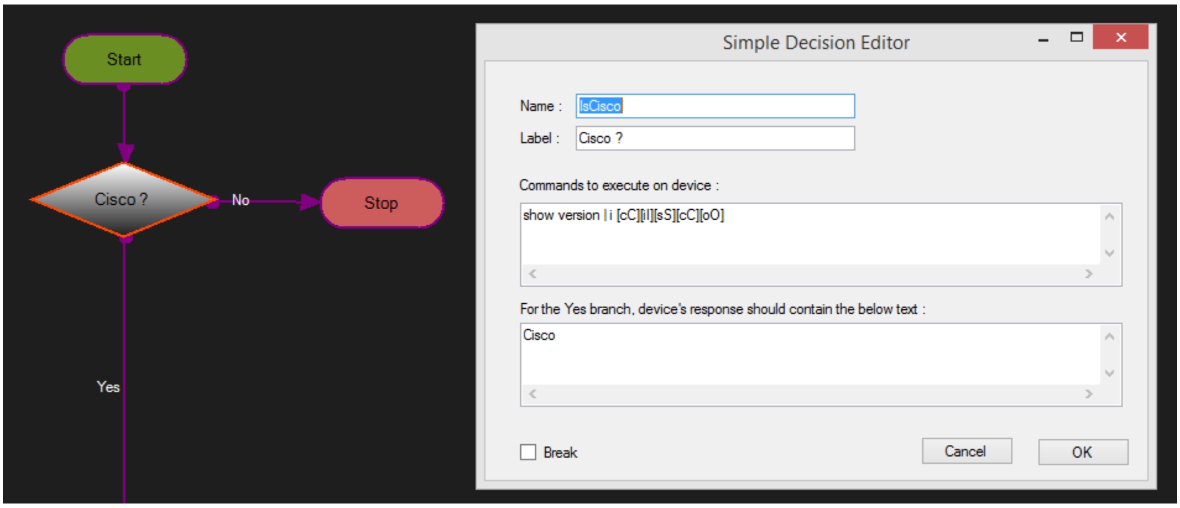


In its simplest form, there is nothing to configure on the Start element, so we can continue with adding a Simple Decision element for checking if we are on a Cisco router.

For this purpose, select Simple Decision element from the context menu shown above.

When the Simple Decision Editor appears, enter the following data to the editor:

- Name : IsCisco
- Label: Is Cisco ?
- Command : show version | i [cC][iI][sS][cC][oO]
- Text to check in answer : Cisco



Name : IsCisco

Label : Cisco ?

Commands to execute on device :

```
show version | i [cC][iI][sS][cC][oO]
```

For the Yes branch, device's response should contain the below text :

```
Cisco
```

☐ Break

This will work exactly as one would expect: sends the command "sh version" to the connected device and checks whether the received answer CONTAINS the word "Cisco". **This is a simple**

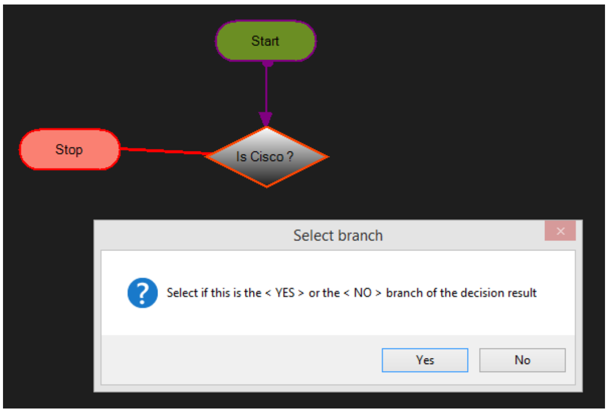
decision: the answer can be "yes" or "no". Please note, that this is a case-sensitive operation both at the router operating system (in case of Cisco IOS and also when parsing the response text. In other words, "cisco" is not equal to "Cisco" when evaluating the response. For the command, we can use the syntax SHOW VERSION | I [CC][II][SS][CC][OO] to match any letter case but the decision element will still evaluate the response in a case sensitive way.

We also need to connect the Start element to this Simple Decision. To do so, go to Start, and select Add Connector from the context menu. The Visual Script designer gets into connection mode and as the mouse is moved over a possible connection target, the element will be highlighted.

From the IsCisco decision element, we have two options: in case the answer was "yes", continue to the next check, if "no", stop the script and report back the results.

Let's start with the "no" branch: first add a Stop element to the script from the context menu. Then go back to the simple decision element, right click on it, and select Add Connector and connect it to the newly added Stop element.

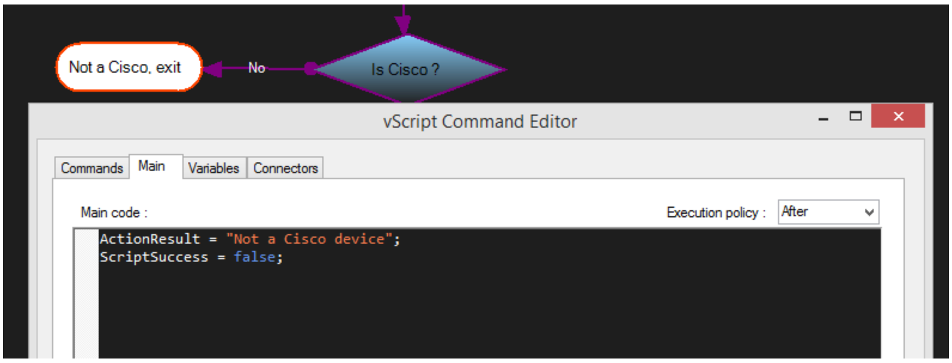
Now it must be decided whether the added connector will represent the "No" or the "Yes" branch of the decision. For this example select "No" as the goal is to stop the script if we are not on a Cisco device.



Select if this is the < YES > or the < NO > branch of the decision result

At this point we may want to report the result with text and also with a logical expression if the configuration was successful or not. To do so, go to the added Stop element and open its editor by double clicking on it.

Then go to the Main tab, and enter the following :



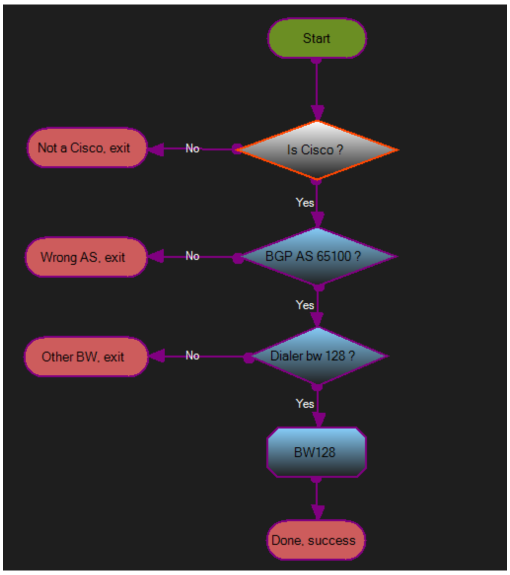
Execution policy : After

```
ActionResult = "Not a Cisco device";
ScriptSuccess = false;
```

As you start typing you will notice that the editor will bring up a popup window for auto completion of the entered text. This way syntax errors can be avoided and you also do not need to remember the correct wording of variables.

The text assigned to the ActionResult variable will be the one appearing in the PGT script window as the Command result. Depending the value of ScriptSuccess, the line in the PGT script window will be coloured green or red.

Following the steps described above, you can build the complete script to get the final one as :



Okay, we have now visually built a simple script, we can run it, debug it, and deploy it. These topics are covered in detail in the user’s manual and I do not want to repeat it, instead, at this point I want to switch to the underlying c# code and the object model used as it is essential to understand the model to effectively develop more complex vScripts.

8.3 THE VSCRIPT OBJECT MODEL

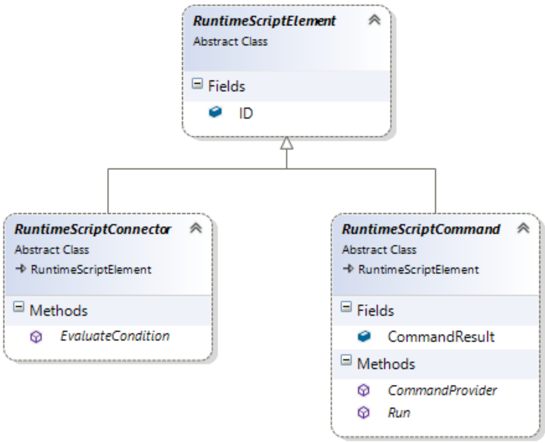
Each visual element of the script represent an individual class. More precisely a nested, public class under a class named as ScriptProxy. The ScriptProxy is the implementation of the IScriptProxy interface which defines all the members required for PGT to interact with the script elements. That is, PGT does not interact directly with the visual script elements, but through the ScriptProxy as it provides a well-defined interface independent of the actual classes inside it.

Let's see how the above simple script appears at code level. Below is the code as generated by PGT internally from the visually constructed script. I know the code seems long to read, but actually very simple and the interesting point is its structure, not what it does:

```
#define DEBUG
using PGT.ExtensionInterfaces;
using PGT.VisualScripts;
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
using System.Text.RegularExpressions;

namespace PGT.VisualScripts.vs_DialerUpdate4SGT
{
    public partial class ScriptProxy:IScriptProxy
    {
        public static string Name = "vs_DialerUpdate4SGT";
        public static bool BreakExecution = false;
        public static string ActionResult = "vScript <vs_DialerUpdate4SGT> processed successfully";
        public static bool ConnectionDropped = false;
        public static bool ScriptSuccess = true;
        public static IScriptExecutorBase Executor;
        public static IscriptableSession Session;
        public static DeviceConnectionInfo ConnectionInfo;
        private List<RuntimeScriptElement> Elements;

        public static TStart Start;
        public static TIsCisco IsCisco;
        public static TStop_0 Stop_0;
        public static TCheckBGP CheckBGP;
        public static TStop_1 Stop_1;
        public static TDialerBandwidth DialerBandwidth;
        public static TStop_2 Stop_2;
        public static TBW128 BW128;
        public static TStop_3 Stop_3;
        public static TStart_IsCisco Start_IsCisco;
        public static TIsCisco_Stop_0 IsCisco_Stop_0;
        public static TIsCisco_CheckBGP IsCisco_CheckBGP;
        public static TCheckBGP_Stop_1 CheckBGP_Stop_1;
        public static TCheckBGP_DialerBandwidth CheckBGP_DialerBandwidth;
        public static TDialerBandwidth_BW128 DialerBandwidth_BW128;
        public static TDialerBandwidth_Stop_2 DialerBandwidth_Stop_2;
        public static TBW128_Stop_3 BW128_Stop_3;

        #region ScriptProxy members
        public ScriptProxy()
        {
            Elements = new List<RuntimeScriptElement>();

            Start = new TStart();
            Elements.Add(Start);
            IsCisco = new TIsCisco();
            Elements.Add(IsCisco);
            Stop_0 = new TStop_0();
            Elements.Add(Stop_0);
            CheckBGP = new TCheckBGP();
            Elements.Add(CheckBGP);
            Stop_1 = new TStop_1();
            Elements.Add(Stop_1);
            DialerBandwidth = new TDialerBandwidth();
            Elements.Add(DialerBandwidth);
            Stop_2 = new TStop_2();
            Elements.Add(Stop_2);
            BW128 = new TBW128();
            Elements.Add(BW128);
            Stop_3 = new TStop_3();
            Elements.Add(Stop_3);
            Start_IsCisco = new TStart_IsCisco();
            Elements.Add(Start_IsCisco);
            IsCisco_Stop_0 = new TIsCisco_Stop_0();
            Elements.Add(IsCisco_Stop_0);
            IsCisco_CheckBGP = new TIsCisco_CheckBGP();
            Elements.Add(IsCisco_CheckBGP);
            CheckBGP_Stop_1 = new TCheckBGP_Stop_1();
            Elements.Add(CheckBGP_Stop_1);
            CheckBGP_DialerBandwidth = new TCheckBGP_DialerBandwidth();
            Elements.Add(CheckBGP_DialerBandwidth);
            DialerBandwidth_BW128 = new TDialerBandwidth_BW128();
            Elements.Add(DialerBandwidth_BW128);
            DialerBandwidth_Stop_2 = new TDialerBandwidth_Stop_2();
            Elements.Add(DialerBandwidth_Stop_2);
            BW128_Stop_3 = new TBW128_Stop_3();
            Elements.Add(BW128_Stop_3);

        }

        public class TStart : RuntimeScriptCommand
        {

            public TStart()
            {
                ID = Guid.Parse("6753d40b-e34d-4108-8b89-d1dcec192fe0");
            }
            public override void Run()
            {

            }
            public override string CommandProvider()
            {
                return "";
            }

        }

        public class TIsCisco : RuntimeScriptCommand
        {

            public TIsCisco()
            {
                ID = Guid.Parse("b40fbd34-bdee-4c92-beb0-540166176ff5");
            }
            public override void Run()
            {

            }
            public override string CommandProvider()
            {
                return "sh version";
            }

        }
        public class TStop_0 : RuntimeScriptCommand
        {

            public TStop_0()
            {
                ID = Guid.Parse("071e0418-fc1d-4a26-8083-0987cef8be42");
            }
            public override void Run()
            {
                ActionResult = "Not a Cisco device";
                ScriptSuccess = false;
            }
            public override string CommandProvider()
            {
                return "";
            }

        }

        public class TCheckBGP : RuntimeScriptCommand
        {

            public TCheckBGP()
            {
                ID = Guid.Parse("045319ba-65ea-4c5e-8c23-4e323281e9a2");
            }
            public override void Run()
            {

            }
            public override string CommandProvider()
            {
                return "sh run | in router bgp";
            }

        }

        public class TStop_1 : RuntimeScriptCommand
        {

            public TStop_1()
            {
                ID = Guid.Parse("7042f2f5-3087-4fcf-8c35-d96c398f23ea");
            }
            public override void Run()
            {
                ActionResult = "Wrong AS number";
                ScriptSuccess = false;
            }
            public override string CommandProvider()
            {
                return "";
            }

        }

        public class TDialerBandwidth : RuntimeScriptCommand
        {

            public TDialerBandwidth()
            {
                ID = Guid.Parse("38022f47-a25e-4756-b466-98a2eae7ca1b");
            }
            public override void Run()
            {

            }
            public override string CommandProvider()
            {
                return "sh run int dialer 1 | inc bandwidth";
            }

        }

        public class TStop_2 : RuntimeScriptCommand
        {

            public TStop_2()
            {
                ID = Guid.Parse("98b449df-1131-4874-902f-e1fc86e85202");
            }
            public override void Run()
            {
                ActionResult = "Other BW";
                ScriptSuccess = false;
            }
            public override string CommandProvider()
            {
                return "";
            }

        }

        public class TBW128 : RuntimeScriptCommand
        {
            public TBW128()
            {
                ID = Guid.Parse("e83bb686-125f-4f34-b5eb-529d09f62be0");
            }
            public override void Run()
            {

            }
            public override string CommandProvider()
            {
                return "dialer load-threshold 100 either";
            }

        }

        public class TStop_3 : RuntimeScriptCommand
        {

            public TStop_3()
            {
                ID = Guid.Parse("761fbbc6-c029-482a-b0c8-5614ca1e6830");
            }
            public override void Run()
            {
                ActionResult = "Dialer IF updated successfully";
                ScriptSuccess = true;
            }
            public override string CommandProvider()
            {
                return "";
            }

        }

        public class TStart_IsCisco : RuntimeScriptConnector
        {

            public TStart_IsCisco()
            {
                ID = Guid.Parse("56f137bc-3b69-4c9d-ba04-8c2d7c76f934");
            }
            public override bool EvaluateCondition()
            {
                 return true;;
            }
        }

        public class TIsCisco_Stop_0 : RuntimeScriptConnector
        {

            public TIsCisco_Stop_0()
            {
                ID = Guid.Parse("75c0be8d-cebe-421c-b008-429291d03118");
            }
            public override bool EvaluateCondition()
            {
                 return IsCisco.CommandResult.IndexOf("Cisco") < 0;;
            }
        }

        public class TIsCisco_CheckBGP : RuntimeScriptConnector
        {

            public TIsCisco_CheckBGP()
            {
                ID = Guid.Parse("b4938aec-f5cc-4006-8fe2-5ef85e0a179f");
            }
            public override bool EvaluateCondition()
            {
                 return IsCisco.CommandResult.IndexOf("Cisco") >= 0;;
            }
        }

        public class TCheckBGP_Stop_1 : RuntimeScriptConnector
        {

            public TCheckBGP_Stop_1()
            {
                ID = Guid.Parse("1cb0c89d-5ddc-4420-ba88-3b7ee645e828");
            }
            public override bool EvaluateCondition()
            {
                 return CheckBGP.CommandResult.IndexOf("65100") < 0;;
            }
        }

        public class TCheckBGP_DialerBandwidth : RuntimeScriptConnector
        {

            public TCheckBGP_DialerBandwidth()
            {
                ID = Guid.Parse("d9948aaf-d569-4b76-a291-a821a25b7671");
            }
            public override bool EvaluateCondition()
            {
                 return CheckBGP.CommandResult.IndexOf("65100") >= 0;;
            }
        }

        public class TDialerBandwidth_BW128 : RuntimeScriptConnector
        {

            public TDialerBandwidth_BW128()
            {
                ID = Guid.Parse("5a71de56-3f39-42fa-82b6-565b8d10dc0d");
            }
            public override bool EvaluateCondition()
            {
                 return DialerBandwidth.CommandResult.IndexOf("128") >= 0;;
            }
        }

        public class TDialerBandwidth_Stop_2 : RuntimeScriptConnector
        {

            public TDialerBandwidth_Stop_2()
            {
                ID = Guid.Parse("f992b6a5-7a78-496b-9666-3dad51bd4b61");
            }
            public override bool EvaluateCondition()
            {
                 return DialerBandwidth.CommandResult.IndexOf("128") < 0;;
            }
        }

        public class TBW128_Stop_3 : RuntimeScriptConnector
        {

            public TBW128_Stop_3()
            {
                ID = Guid.Parse("a0c54fb8-94d3-45f3-9c30-e18ccd9de7c3");
            }
            public override bool EvaluateCondition()
            {
                 return true;;
            }
        }
       #endregion

    }
}
```

It can be seen that each visual script element corresponds to a runtime class nested into ScriptProxy class. ScriptProxy implements the IScriptProxy interface (no need to go into details of it) and provides a means of interaction between the vScript execution engine and the runtime generated script elements. The important point is that each script element have its own unique which is the link between the runtime generated class and the visually designed element. All of them are instantiated in the ScriptProxy constructor and are added to the list of known elements which list is used internally by ScriptProxy to provide interaction.

The runtime generated classes descend either from RuntimeScriptCommand or RuntimeScriptConnector, while their common ancestor is RuntimeScriptElement as shown below:



The key to script execution is that both RuntimeScriptCommand and RuntimeScriptConnector classes have abstract methods which PGT can call as the script executes.

For example, look at TStart:

```
public class TStart : RuntimeScriptCommand
{
    public TStart()
    {
        ID = Guid.Parse("6753d40b-e34d-4108-8b89-d1dcec192fe0");
    }
    public override void Run()
    {
    }
    public override string CommandProvider()
    {
        return "";
    }
}
```

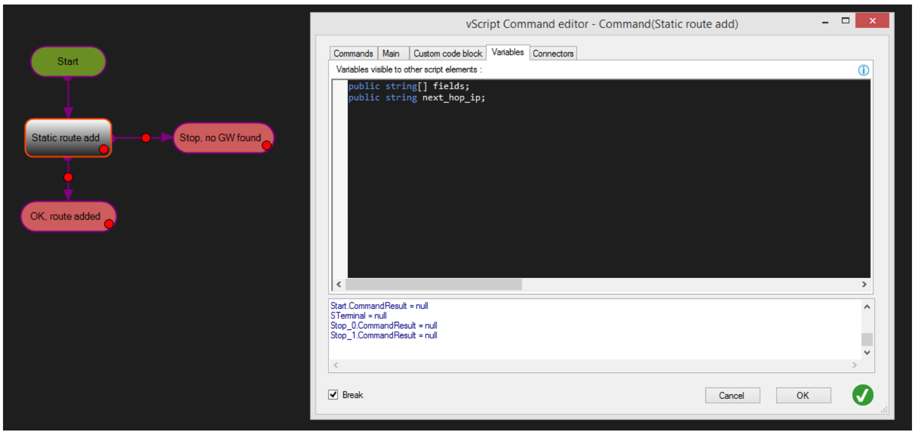
CommandProvider() is used to get the CLI comand that will be sent to a device, and Run() contains the code entered in the Main block of a Command element. PGT will call these two methods when the Start element is executed.

Connectors, on the other hand, are more simple stuff. They have just one method, the EvaluateCondition() which returns a boolean value. If the returned value is True, PGT assumes execution should flow to the connected element of this connector. For instance the connector pointing from Check BGP AS 6500? to Dialer BW 128? element evaluates the CommandResult variable of CheckBGP if the answer contained the string 65000:

```
public class TCheckBGP_DialerBandwidth : RuntimeScriptConnector
{
    public TCheckBGP_DialerBandwidth()
    {
        ID = Guid.Parse("d9948aaf-d569-4b76-a291-a821a25b7671");
    }
    public override bool EvaluateCondition()
    {
        return CheckBGP.CommandResult.IndexOf("65100") >= 0;;
    }
}
```

As Start actually does not do anything interesting in this example, let’s see another script which is about adding a new static route to a host.

First see the script in the Visual Script Editor, focusing on the "Static route add" command element first:

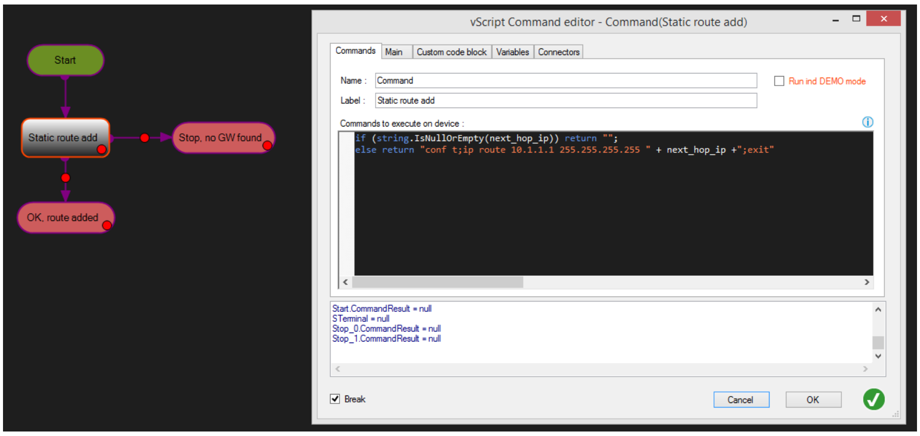


Variables visible to other script elements :

```
public string[] fields;
public string next_hop_ip;
```

```
Start.CommandResult = null
STerminal = null
Stop_0.CommandResult = null
Stop_1.CommandResult = null
```

☑ Break



Name : Command

Label : Static route add

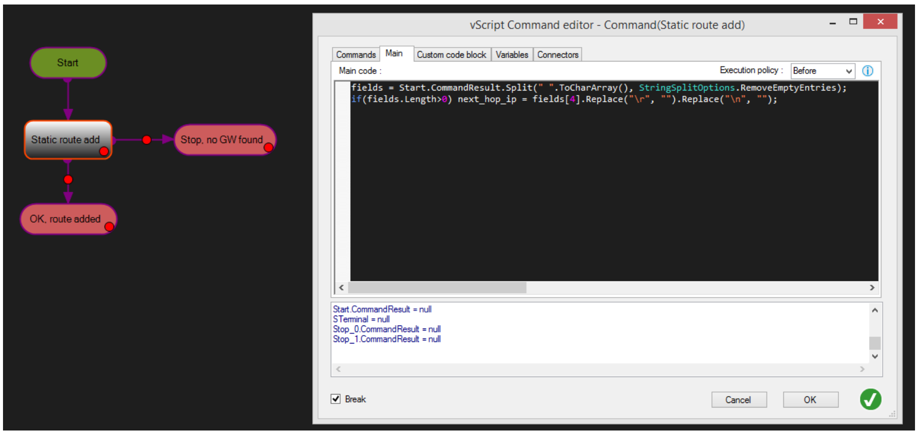
☐ Run ind DEMO mode

Commands to execute on device :

```
if (string.IsNullOrEmpty(next_hop_ip)) return "";
else return "conf t;ip route 10.1.1.1 255.255.255.255 " + next_hop_ip +";exit"
```

```
Start.CommandResult = null
STerminal = null
Stop_0.CommandResult = null
Stop_1.CommandResult = null
```

☑ Break



Main code :

Execution policy : Before

```
fields = Start.CommandResult.Split(" ".ToCharArray(), StringSplitOptions.RemoveEmptyEntries);
if(fields.Length>0) next_hop_ip = fields[4].Replace("\r", "").Replace("\n", "");
```

```
Start.CommandResult = null
STerminal = null
Stop_0.CommandResult = null
Stop_1.CommandResult = null
```

☑ Break

As you can see, this command element has some variables and a Main code block to process the CommandResult variable of the start element - will show you what it is - and extract the next hop address of a route to store it in its local variable next_hop_ip. Then in the Commands block uses this variable to construct the CLI command to be sent to the router.

Let's see the generated code of the vScript:

```
using PGT.ExtensionInterfaces;
using PGT.VisualScripts;
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
using System.Text.RegularExpressions;
```

```
namespace PGT.VisualScripts.new_vScript
{
  public partial class ScriptProxy:IScriptProxy
  {
    public static string Name = "new_vScript";
    public static bool BreakExecution = false;
    public static string ActionResult = "vScript <new_vScript> processed successfully";
    public static bool ConnectionDropped = false;
    public static bool ScriptSuccess = true;
    public static IScriptExecutorBase Executor;
    public static IScriptableSession Session;
    public static DeviceConnectionInfo ConnectionInfo;
    private List<RuntimeScriptElement> Elements;

    public static TStart Start;
    public static TCommand Command;
    public static TStop_0 Stop_0;
    public static TStop_1 Stop_1;
    public static TKnownGW KnownGW;
    public static TStart_CommandOnNo Start_CommandOnNo;
    public static TNoGW NoGW;


    #region ScriptProxy members
    public ScriptProxy()
    {
      Elements = new List<RuntimeScriptElement>();

      Start = new TStart();
      Elements.Add(Start);
      Command = new TCommand();
      Elements.Add(Command);
      Stop_0 = new TStop_0();
      Elements.Add(Stop_0);
      Stop_1 = new TStop_1();
      Elements.Add(Stop_1);
      KnownGW = new TKnownGW();
      Elements.Add(KnownGW);
      Start_CommandOnNo = new TStart_CommandOnNo();
      Elements.Add(Start_CommandOnNo);
      NoGW = new TNoGW();
      Elements.Add(NoGW);

    }

    public class TStart : RuntimeScriptCommand
    {

      public TStart()
      {
        ID = Guid.Parse("6442f81e-1ce9-4148-9bdc-9707f475ae8c");
      }
      public override void Run()
      {

      }
      public override string CommandProvider()
      {
        return "sh run | i ip route 192.168.10.0";
      }

    }

    public class TCommand : RuntimeScriptCommand
    {
      public string[] fields;
      public string next_hop_ip;
      public TCommand()
      {
        ID = Guid.Parse("5e054322-8eb4-40dd-b754-2949933bc6de");
      }
      public override void Run()
      {
        fields = Start.CommandResult.Split(" ".ToCharArray(), StringSplitOptions.RemoveEmptyEntries);
        if(fields.Length>0) next_hop_ip = fields[4].Replace("\r", "").Replace("\n", "");
      }
      public override string CommandProvider()
      {
        if (string.IsNullOrEmpty(next_hop_ip)) return "";
      else return "conf t;ip route 10.1.1.1 255.255.255.255 " + next_hop_ip +";exit";
      }
    }

    public class TStop_0 : RuntimeScriptCommand
    {

      public TStop_0()
      {
        ID = Guid.Parse("dade9f45-0a91-4daa-b1e1-4c422e814b51");
      }
      public override void Run()
      {
        ActionResult = "Static route configured";
        ScriptSuccess = true;
      }
      public override string CommandProvider()
      {
        return "";
      }

    }

    public class TStop_1 : RuntimeScriptCommand
    {

      public TStop_1()
      {
        ID = Guid.Parse("b0d144b8-56d2-4e7a-8672-f8c904d7c4f0");
      }
      public override void Run()
      {
        ActionResult = "Could find gateway";
        ScriptSuccess = false;
      }
      public override string CommandProvider()
      {
        return "";
      }

    }

    public class TKnownGW : RuntimeScriptConnector
    {

      public TKnownGW()
      {
        ID = Guid.Parse("22537898-4633-4b25-8940-791cff434f45");
      }
      public override bool EvaluateCondition()
      {
         return true;;
      }
    }

    public class TStart_CommandOnNo : RuntimeScriptConnector
    {

      public TStart_CommandOnNo()
      {
        ID = Guid.Parse("7a781dbf-4e46-4448-8f5f-b770472a1be0");
      }
      public override bool EvaluateCondition()
      {
         return true;;
      }
    }

    public class TNoGW : RuntimeScriptConnector
    {

      public TNoGW()
      {
        ID = Guid.Parse("e209a3c1-80d1-4c68-986d-732c4eb80642");
      }
      public override bool EvaluateCondition()
      {
        return  string.IsNullOrEmpty(Command.next_hop_ip);
      }
    }


    #endregion
  }
}
```

First check the Start element code. The CommandProvider() method will now return a valid command to check for an existing route, so it will be sent to the connected router. All runtime script elements has a variable named CommandResult which will store the response received from the connected device. As the CommandResult are usually a multiline text, PGT also provides the aCommandResult variable which is a string array and contains CommandResult lines. Using aCommandResult can eliminate the need of splitting CommandResult to lines. We will use this in TCommand's Run() method to extract the next hop ip from the response - which is the fourth word - and store it in the local variable next_hop_ip:

```
public class TCommand : RuntimeScriptCommand
{
    public string[] fields;
    public string next_hop_ip;
    public TCommand()
    {
        ID = Guid.Parse("5e054322-8eb4-40dd-b754-2949933bc6de");
    }
    public override void Run()
    {
        fields = Start.CommandResult.Split(" ".ToCharArray(), StringSplitOptions.RemoveEmptyEntries);
        if(fields.Length>4) next_hop_ip = fields[4].Replace("\r", "").Replace("\n", "");
    }
    public override string CommandProvider()
    {
        if (string.IsNullOrEmpty(next_hop_ip)) return "";
        else return "conf t;ip route 10.1.1.1 255.255.255.255 " + next_hop_ip +";exit";
    }
}
```

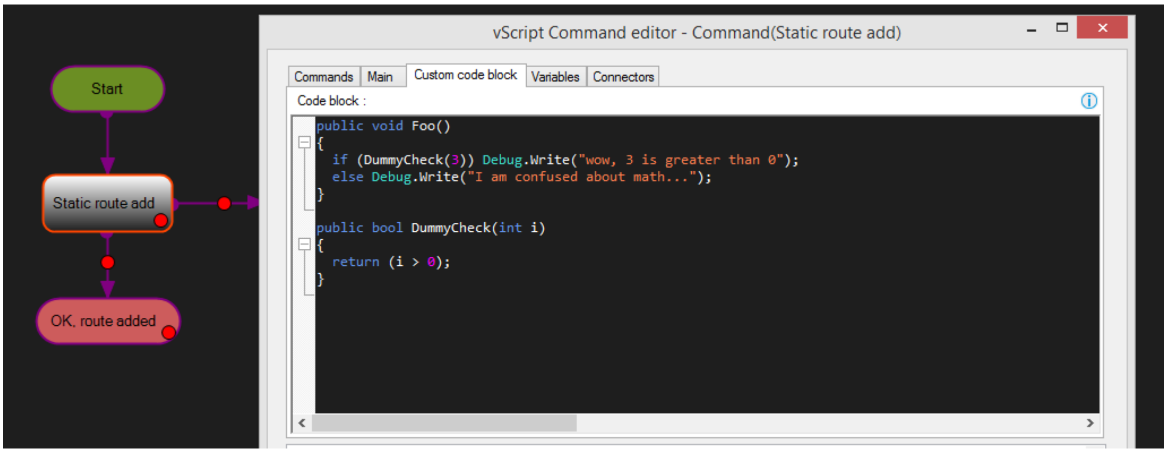
Then in CommandProvider() method we check the variable, and if it is not empty, we build the configuration commands we want to send to the router.

8.4 A STEP FURTHER

For the vScript to successfully execute, PGT requires that a runtime command element has the Run() and CommandProvider() methods. But why limit the class to only these two methods? To make a runtime command element more versatile, you can define a custom code block of the element, which code is injected to the class as is. That is, the code block should be formatted to syntactically fit into a class. Let's see an example how to extend our TCommand class from the above example:

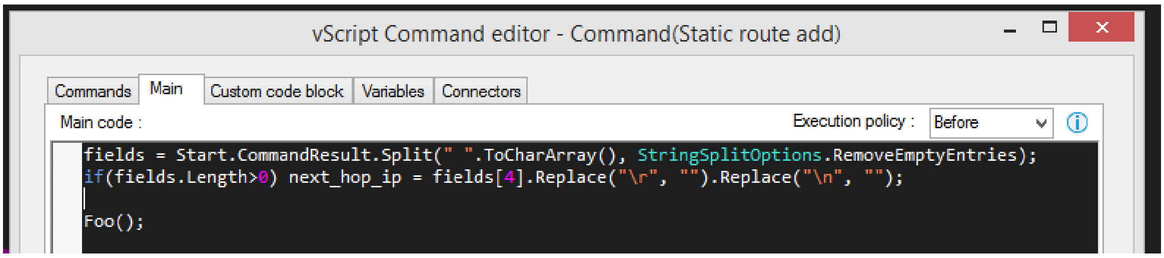
```
public class TCommand : RuntimeScriptCommand
{
    public string[] fields;
    public string next_hop_ip;
    public TCommand()
    {
        ID = Guid.Parse("5e054322-8eb4-40dd-b754-2949933bc6de");
    }
    public override void Run()
    {
        fields = Start.CommandResult.Split(" ".ToCharArray(), StringSplitOptions.RemoveEmptyEntries);
        if(fields.Length>4) next_hop_ip = fields[4].Replace("\r", "").Replace("\n", "");
    }
    public override string CommandProvider()
    {
        if (string.IsNullOrEmpty(next_hop_ip)) return "";
        else return "conf t;ip route 10.1.1.1 255.255.255.255 " + next_hop_ip +";exit";
    }
    //<This is the placeholder of a custom code block>
}
```

As you see, the custom code block can contain any number of methods or variables. Although PGT will not directly call these methods, you can then can do so in either the Main code block or the Commands code block. Consider the below example, where we add two class methods:



```
public void Foo()
{
  if (DummyCheck(3)) Debug.Write("wow, 3 is greater than 0");
  else Debug.Write("I am confused about math...");
}

public bool DummyCheck(int i)
{
  return (i > 0);
}
```

Then we can make use of them from Main:



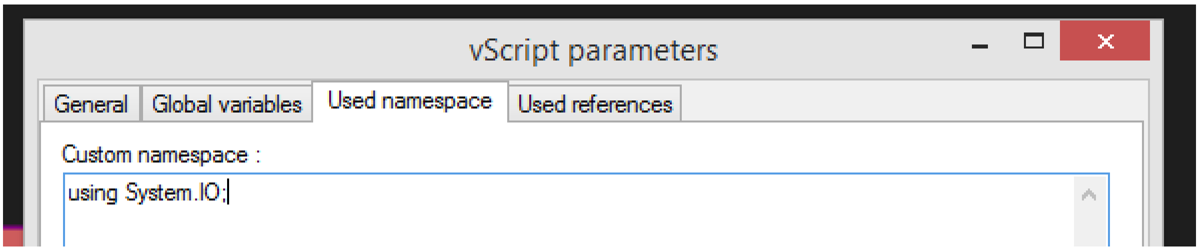
Execution policy : Before

```
fields = Start.CommandResult.Split(" ".ToCharArray(), StringSplitOptions.RemoveEmptyEntries);
if(fields.Length>0) next_hop_ip = fields[4].Replace("\r", "").Replace("\n", "");

Foo();
```

And the code generated in the background is:

```
public class TCommand : RuntimeScriptCommand
{
    public string[] fields;
    public string next_hop_ip;
    public TCommand()
    {
        ID = Guid.Parse("5e054322-8eb4-40dd-b754-2949933bc6de");
    }
    public override void Run()
    {
        fields = Start.CommandResult.Split(" ".ToCharArray(), StringSplitOptions.RemoveEmptyEntries);
        if(fields.Length>0) next_hop_ip = fields[4].Replace("\r", "").Replace("\n", "");
        Foo();
    }

    public override string CommandProvider()
    {
        if (string.IsNullOrEmpty(next_hop_ip)) return "";
    else return "conf t;ip route 10.1.1.1 255.255.255.255 " + next_hop_ip +";exit";
    }

    public void Foo()
    {
        if (DummyCheck(3)) Debug.Write("wow, 3 is greater than 0");
        else Debug.Write("I am confused about math...");
    }
    public bool DummyCheck(int i)
    {
        return (i > 0);
    }
}
```

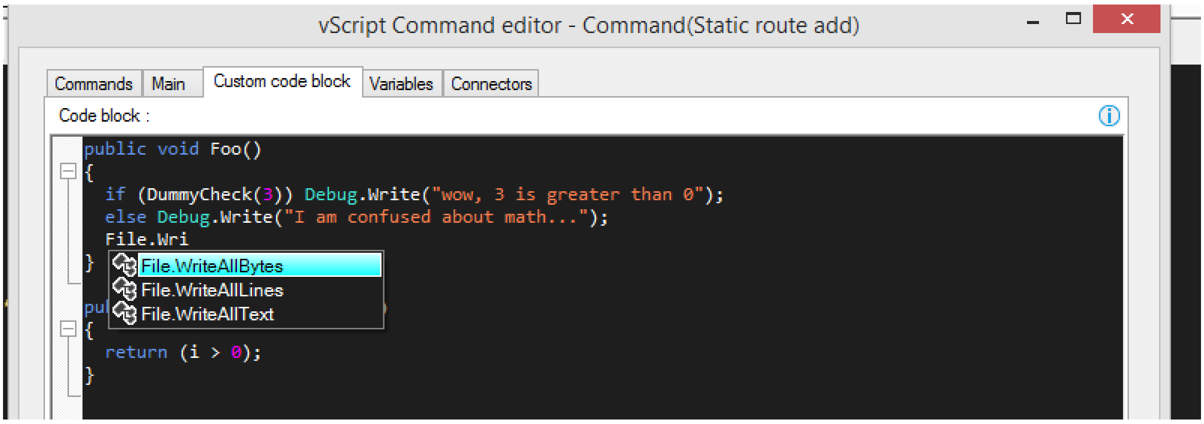
To make things even more elastic, you can add custom usings and external assembly references to the vScript. For instance, if you need to do some file operation from within the script, you need to add System.IO to the usings, and afterwards you can use the File class in your script:



Custom namespace :

```
using System.IO;
```

And so the File class gets known to the code editor:



Code block :

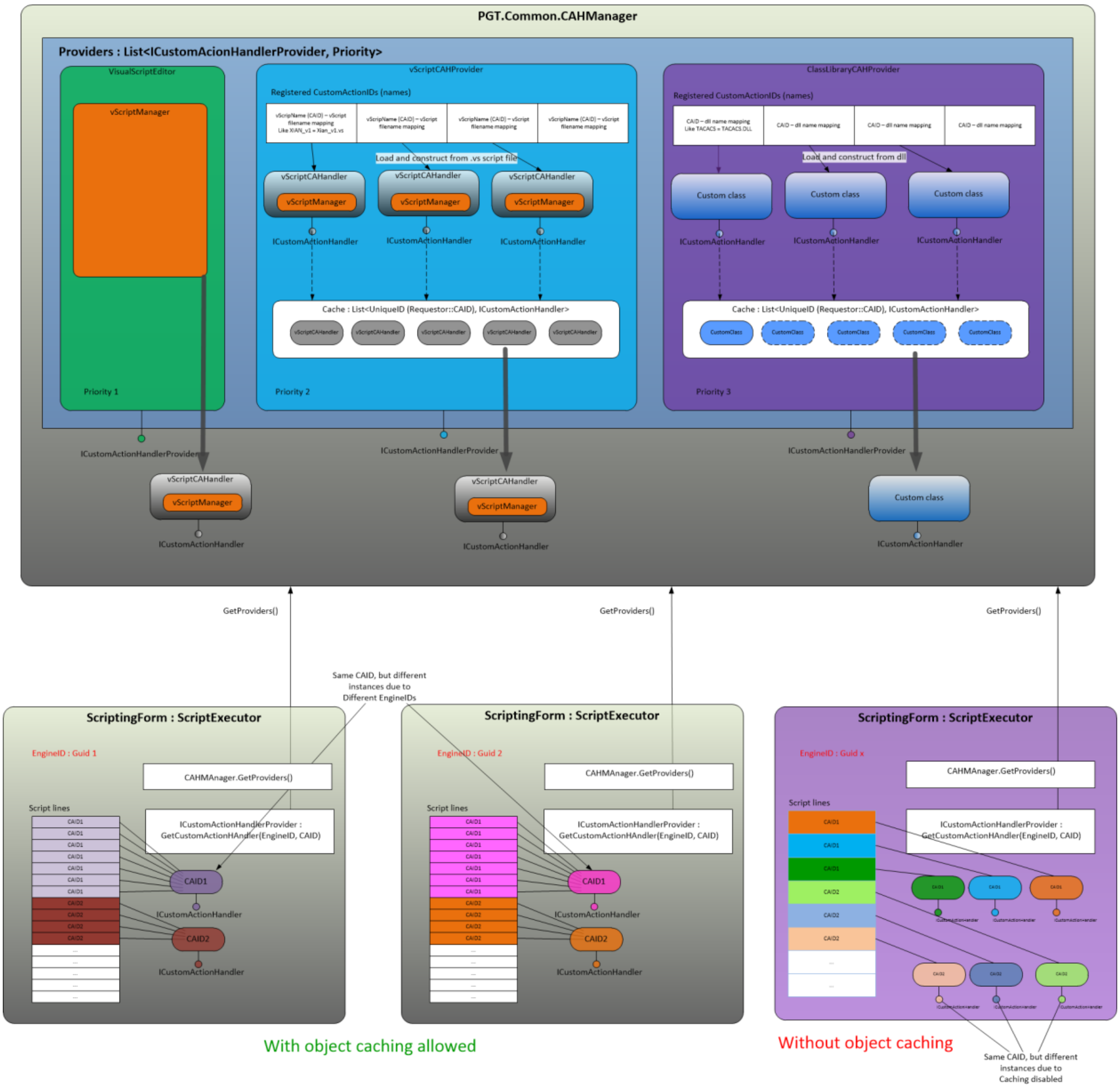
```
public void Foo()
{
  if (DummyCheck(3)) Debug.Write("wow, 3 is greater than 0");
  else Debug.Write("I am confused about math...");
  File.Wri
}

pul
{
  return (i > 0);
}
```

9 CUSTOMACTIONHANDLER ARCHITECTURE

The following diagram show PGT internal architecture for CustomActionHandlers. The aim of this diagram to make it clear how objects are created and maintained:

CAID = CustomActionID – the name of the Custom Action Handler

CustomActionHandlerProvider priority is important when the same CAID is served by different providers. This happens for instance when the VisualScriptEditor is opened with a vScript that is also registered via Tools/Options. In this case the VisualScriptEditor will produce the CustomActionHAndler as it has the lowest priority


PGT.Common.CAHManager
Providers : List<ICustomAcionHandlerProvider, Priority>
VisualScriptEditor
vScriptManager
Priority 1
vScriptCAHProvider
Registered CustomActionIDs (names)
vScripName (CAID) – vScript filename mapping Like XIAN_v1 = Xian_v1.vs
vScripName (CAID) – vScript filename mapping
Load and construct from .vs script file
vScriptCAHandler
vScriptManager
ICustomActionHandler
Cache : List<UniqueID (Requestor::CAID), ICustomActionHandler>
vScriptCAHandler
Priority 2
ClassLibraryCAHProvider
Registered CustomActionIDs (names)
CAID – dll name mapping Like TACACS = TACACS.DLL
CAID – dll name mapping
Load and construct from dll
Custom class
ICustomActionHandler
Cache : List<UniqueID (Requestor::CAID), ICustomActionHandler>
CustomClass
Priority 3
ICustomActionHandlerProvider
vScriptCAHandler
vScriptManager
ICustomActionHandler
Custom class
GetProviders()
Same CAID, but different instances due to Different EngineIDs
ScriptingForm : ScriptExecutor
EngineID : Guid 1
EngineID : Guid 2
EngineID : Guid x
CAHMAnager.GetProviders()
ICustomActionHandlerProvider : GetCustomActionHAndler(EngineID, CAID)
Script lines
CAID1
CAID2
ICustomActionHandler
With object caching allowed
Without object caching
Same CAID, but different instances due to Caching disabled

Object caching can be disabled for vScripts only in Tools/Options/vScript settings.

10 LICENSE

Portions of the software are licensed under Apache License v2.0

11 LIMITATION OF LIABILITY

UNDER NO LEGAL THEORY, INCLUDING, BUT NOT LIMITED TO, NEGLIGENCE, TORT, CONTRACT, STRICT LIABILITY, OR OTHERWISE, SHALL THE AUTHOR OF THE PROGRAM CODE BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, EXEMPLARY, RELIANCE OR CONSEQUENTIAL DAMAGES INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOST PROFITS, LOSS OF GOODWILL, WORK STOPPAGE, ACCURACY OF RESULTS, COMPUTER FAILURE OR MALFUNCTION, OR DAMAGES RESULTING FROM USE. THE AUTHOR OF THE PROGRAM CODE LIABILITY FOR DAMAGES OF ANY KIND WHATSOEVER ARISING OUT OF THIS AGREEMENT SHALL BE LIMITED TO THE FEES PAID BY LICENSEE FOR THE SOFTWARE.

WARRANTY DISCLAIMER. THE AUTHOR OF THE PROGRAM CODE PROVIDES THE SOFTWARE "AS IS" AND WITHOUT WARRANTY OF ANY KIND, AND HEREBY DISCLAIMS ALL EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, PERFORMANCE, ACCURACY, RELIABILITY, QUIET ENJOYMENT, INTEGRATION, TITLE, NON-INTERFERENCE AND NON-INFRINGEMENT. FURTHER, THE AUTHOR OF THE PROGRAM CODE DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS THAT THE SOFTWARE WILL BE FREE FROM BUGS OR THAT ITS USE WILL BE UNINTERRUPTED OR THAT THE SOFTWARE OR WRITTEN MATERIALS WILL BE CORRECT, ACCURATE, OR RELIABLE. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS AGREEMENT..