## PowerShape and PowerMill API

# User Guide - PowerMill

Contents

[PowerShape and PowerMill API 1](#_Toc411586045)

[User Guide - PowerMill 1](#_Toc411586046)

[Introduction 3](#_Toc411586047)

[What is PowerShape and PowerMill API? 3](#_Toc411586048)

[Installing the API 4](#_Toc411586049)

[Learning to code with the API 5](#_Toc411586050)

[Examples 16](#_Toc411586051)

## Introduction

### What is PowerShape and PowerMill API?

PowerShape and PowerMill API, hereafter referred to as the API, is an easy to use coding alternative to direct use of the macro language supplied by PowerShape and PowerMill.

The power of the API can be harnessed from within the Microsoft Visual Studio programming environment and used for standalone executables. The programmer can choose to develop in either C# or VB.net.

Advantages of the the API approach to application development are as follows:

* Changes made to the macro language by PowerShape and PowerMill development teams are hidden from the developer, with version discrepancies being handled seamlessly by the API.
* The developer is able to leverage all of the power provided by the VB.net and C# programming languages.
* Significantly fewer lines of code are required to perform like-for-like operations using the API over direct macro language scripting, resulting in reduced complexity and shorter development times.

Throughout this document:

* Notes have been written in both C# and VB.net and have been colour coded as follows:

|  |  |
| --- | --- |
|  | C# |
|  |  |
|  | VB.net |

* You will need a licensed copy of Microsoft Visual Studio 2010 or later to make use of PowerShape and PowerMill API
* We recommend that, if running a 32Bit version of PowerMill, you compile your applications as 32Bit; likewise, if you are running a 64Bit version of PowerMill, compile in 64Bit.

### Where to find the API

The API is now open source at GitHub here:

<https://github.com/Autodesk/PowerShapeAndPowerMillAPI>

You can get the lastest alpha, beta or release version on NuGet here:

<https://www.nuget.org/profiles/Autodesk>

### Learning to code with the API

To begin with, we will look at some of the more common tasks performed with the API such as starting a new project, connecting to PowerMill, creating simple models and so forth. After this we will look at some examples where macro code can be replaced with the API.

#### Creating a Project

1. Open up Microsoft Visual Studio and start a new project.
2. Select ‘Visual C#’ or ‘Visual Basic’ (based on your preference) from the options on the left, then select ‘Console Application’ (Figure 1).
3. Give the project a name and a save location (Figure 1).

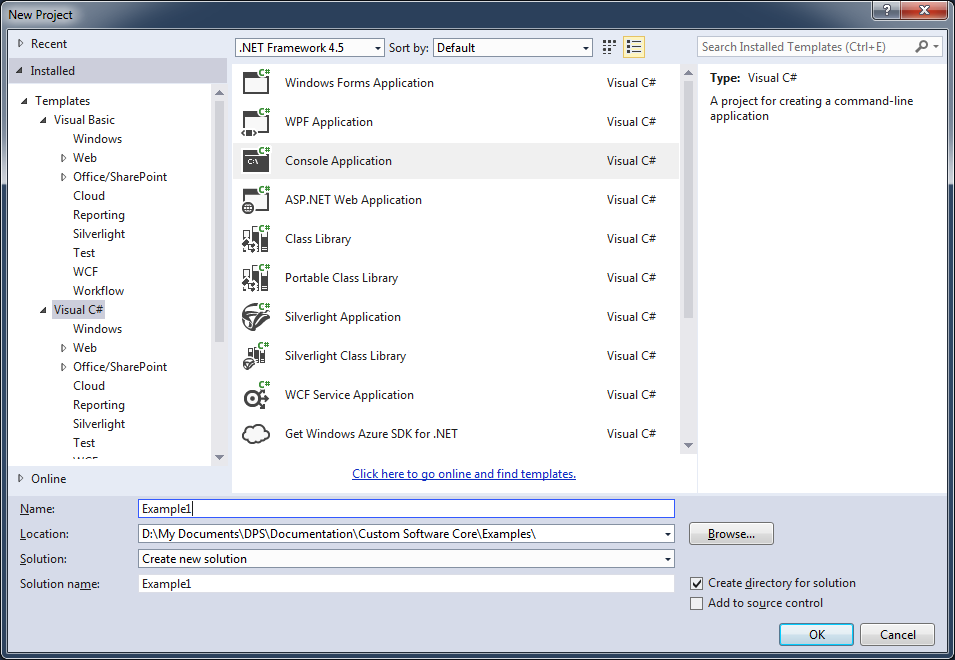


Figure – New Project

1. Your main project source file will contain the following code:

|  |  |
| --- | --- |
|  | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace Example1  {  class Program  {  static void Main(string[] args)  {  }  }  } |
|  | Module Module1  Sub Main()  End Sub  End Module |

1. The next step is to set references to PowerShape and PowerMill API. Figure 2 shows a reference diagram detailing the dependencies for each reference.

Autodesk.Utilities

Autodesk.Geometry

Autodesk.ProductInterface

Autodesk.ProductInterface.PowerMill

Autodesk.ProductInterface.PowerShape

Figure – Reference Diagram

1. To add the required references there are now 2 options. The preferred method is to use NuGet. If you are using an older version that came with an installer then go to step 9 to 11.
2. In your project, right click on “References” and choose “Manage NuGet Packages…”
3. Find available packages at nuget.org by searching for “Autodesk”. You will find (amongst others) the following packages:

* Autodesk.ProductInterface.PowerMill – Use this one to create an application that will interface with PowerMill
* Autodesk.WPFControls.ProductControls.PowerMill – Use this one to embed PowerMill into your WPF application.
* You will have to choose show prerelease versions to add alpha or beta versions.

1. If using an installed verison of the API, right click on ‘References’ within your solution in the Solution Explorer on the left and then select ‘Add’ then ‘Reference’ (Figure 3).

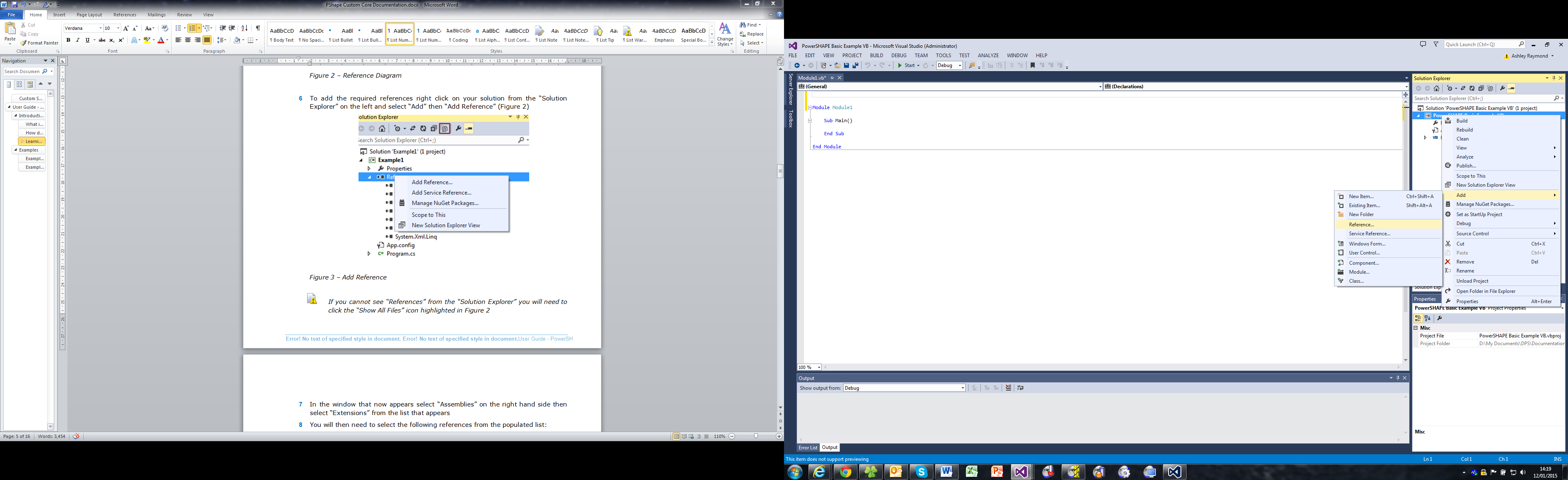


Figure – Add Reference

1. From within the window that now appears, select ‘Assemblies’ and then ‘Extensions’ from the list.
2. Navigate the list of references and select the following:

* Autodesk.Geometry
* Autodesk.ProductInterface
* Autodesk.ProductInterface.PowerMill
* Autodesk.Utilities

1. With the references added (manuall or via NuGet), insert the following using directive at the top of your source file:

|  |  |
| --- | --- |
|  | using Autodesk.ProductInterface.PowerMill; |
|  |  |
|  | Imports Autodesk.ProductInterface.PowerMill |

#### Connecting to PowerMill

The following section demonstrates how the API can be used to communicate with a running instance of PowerMill.

1. Start an instance of PowerMill.
2. To access the PowerMill instance, add the following line to the main method of your source file:

|  |  |
| --- | --- |
|  | PMAutomation powerMill = new PMAutomation (Autodesk.ProductInterface.InstanceReuse.UseExistingInstance); |
|  |  |
|  | Dim powerMill As New PMAutomation(Autodesk.ProductInterface.  InstanceReuse.UseExistingInstance) |

* While typing this line IntelliSense gives a number of options. For example, after InstanceReuse we have options to: create a new instance (in addition to any existing instance of PowerMill); Create a single instance (close any PowerMill instances already open and create a new one) or UseExistingInstance (attach to a running instance of PowerMill) see Figure 4.

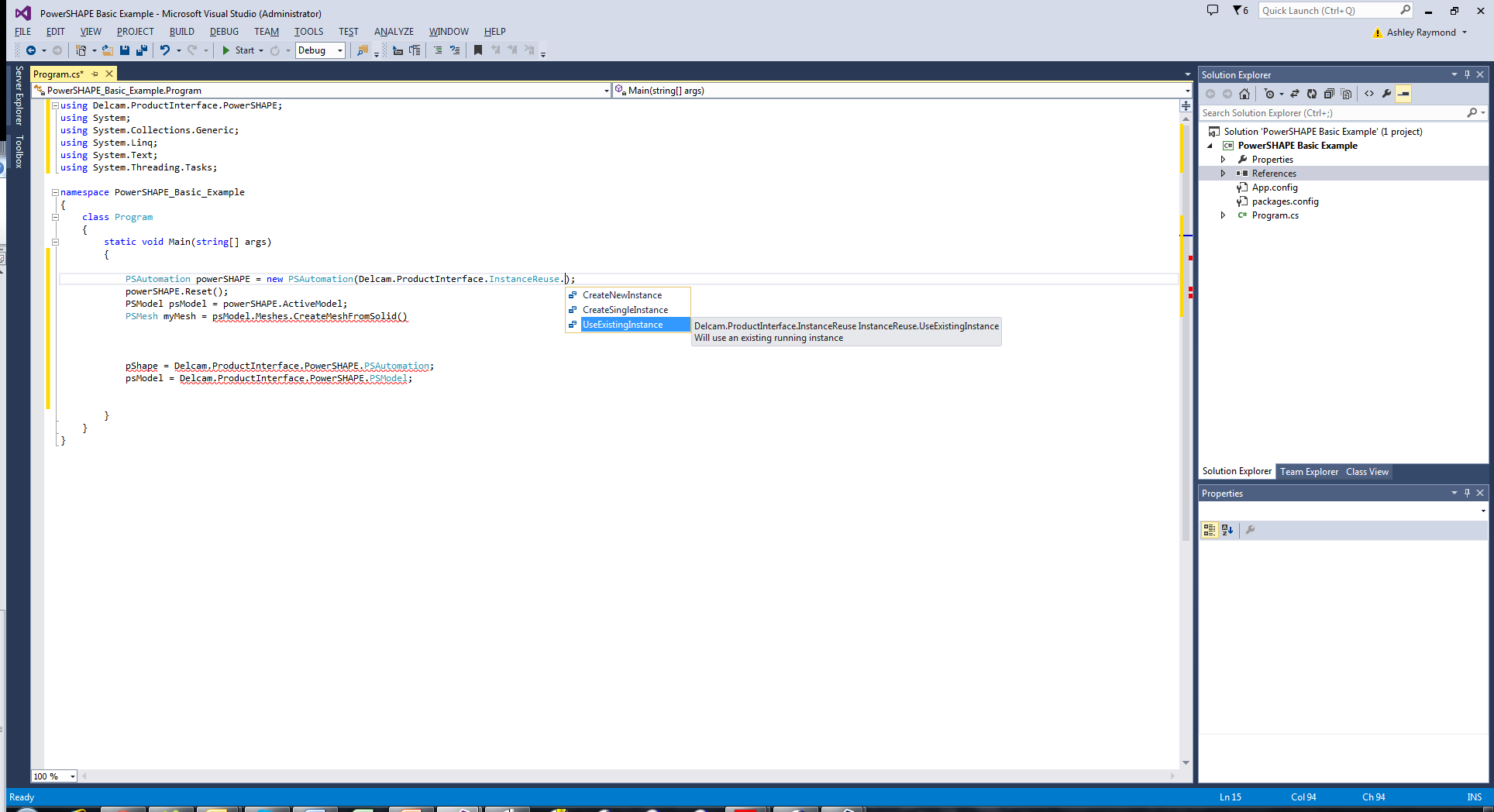


Figure – Using IntelliSense

* If you continue the line by typing a ‘.’ after your selection of InstanceReuse, you are presented with an additional option to display or not the PowerMill GUI.

1. Now attach to the PowerMill active project by adding the following code to the main method of your source file:

|  |  |
| --- | --- |
|  | PMProject session = powerMill.ActiveProject; |
|  |  |
|  | Dim session As PMProject = powerMill.ActiveProject |

1. The PMProject object (session) can now be used to perform a number of operations on PowerMill (Figure 5).

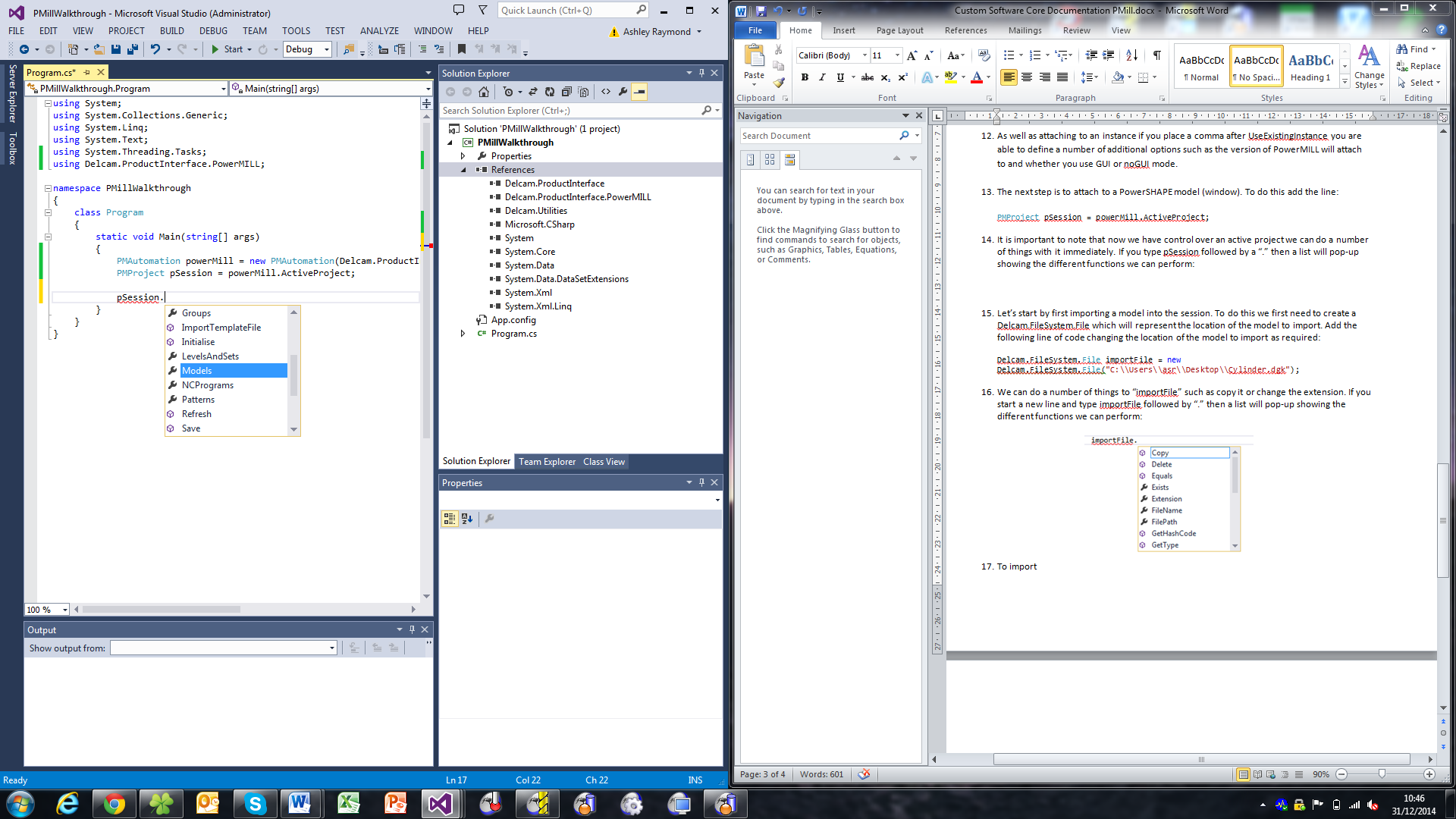


Figure – IntelliSense with session

1. At this point it may be desirable to disable PowerMill dialogs so that code is free to run without user prompts.

|  |  |
| --- | --- |
|  | powerMill.DialogsOff();  powerMill.DialogsOn(); |
|  |  |
|  | powerMill.DialogsOff()  powerMill.DialogsOn() |

#### Importing/Exporting a Model

To import a solid/surface, create a Autodesk.FileSystem.File object and add this to the session.

1. Create an instance of a Autodesk.FileSystem.File object as follows:

|  |  |
| --- | --- |
|  | Autodesk.FileSystem.File importFile = new Autodesk.FileSystem.File(@"C:\Users\asr\Desktop\Cylinder.dgk"); |
|  |  |
|  | Dim importFile As New Autodesk.FileSystem.File("C:\Users\asr\Desktop\Cylinder.dgk") |

* Cylinder.dgk will be located in the path to which you extracted your example files earlier and may be different to that specified above.

1. Import the model into the session:

|  |  |
| --- | --- |
|  | PMModel myModel = session.Models.CreateModel(importFile); |
|  |  |
|  | Dim myModel As PMModel = session.Models.CreateModel(importFile) |

1. Operations can now be performed upon the model. For example, should you wish to find the maximum size of the model in X, use the bounding box method shown here:

|  |  |
| --- | --- |
|  | Autodesk.Geometry.MM sizeX = myModel.BoundingBox.MaxX; |
|  |  |
|  | Dim sizeX As Autodesk.Geometry.MM = myModel.BoundingBox.MaxX |

* Lengths are stored in a Autodesk.Geometry.MM as it facilitates length-specific operations such as converting to inches:

|  |  |
| --- | --- |
|  | Inch inchX = (Inch)sizeX; |
|  |  |
|  | Dim inchX As Inch = CType(sizeX, Inch) |

* PowerShape and PowerMill API always assumes the PowerMill session to be running in mm; thus the return value will always be treated as mm. If you attempted the following: Autodesk.Geometry.Inch sizeX = myModel.BoundingBox.MaxX with a session running in inches, the result would be read in Inches but then multiplied by 25.4 as the API assumes it to be in mm.

1. If we wished to select the model (to perform an operation only on selected items, for example) we can use the code:

|  |  |
| --- | --- |
|  | myModel.AddToSelection(true); |
|  |  |
|  | myModel.AddToSelection(True) |

* Adding true as an argument to AddToSelection will clear the current selection in PowerMill before selecting the model.

1. This method can also be applied to importing a DMT file as a block to machine using:

|  |  |
| --- | --- |
|  | session.CreateBlock(blockFile); |
|  |  |
|  | session.CreateBlock(blockFile) |

* In this case ‘blockFile’ must be a Autodesk.FileSystem.File pointing to a DMT file available to you.

#### Tools and Toolpaths

There are also a number of operations that can be performed on toolpaths and tools within the session. In order for the following code segments to return a result, you require access to a session that has tools and toolpaths defined.

1. Assume there exists within the session a single tool about which you require information. Firstly you must instantiate a PMTool object within your code and assign to it the tool about which information is required. This is achieved with the following code:

|  |  |
| --- | --- |
|  | PMTool tool = session.Tools[0]; |
|  |  |
|  | Dim tool As PMTool = session.Tools(0) |

* You will need to know the index of the tool to create an instance of it or, alternatively, the name of the tool, which can be used to achieve the same result thus: session.Tools.get\_Item(<name>).

1. We are now able to perform a number of operations on the tool; for example, we can change the diameter of the tool as follows:

|  |  |
| --- | --- |
|  | tool.Diameter = 20.0; |
|  |  |
|  | tool.Diameter = 20.0 |

* If there exists a pre-calculated toolpath that uses this tool, it will be invalidated by the above operation and PowerMill will issue a warning to that effect. It is, therefore, necessary to invalidate all toolpaths before changing the tool. Code to manage this is presented in the accompanying samples.

1. Alternatively, if you want to know the diameter of the tool, use the following:

|  |  |
| --- | --- |
|  | Autodesk.Geometry.MM toolDiam = tool.Diameter; |
|  |  |
|  | Dim toolDiam As Autodesk.Geometry.MM = tool.Diameter |

1. The same methodology can be applied to instantiating an instance of an existing toolpath. To create the instance use:

|  |  |
| --- | --- |
|  | PMToolpath toolpath = session.Toolpaths[0]; |
|  |  |
|  | Dim toolpath As PMToolpath = session.Toolpaths(0) |

* You will need to know the index of the toolpath to create an instance of it or, alternatively, the name of the toolpath, which can be used to achieve the same result thus: session.Toolpaths.get\_Item(<name>).

1. We are now able to perform a number of operations on the toolpath; for example, we can set the name of the toolpath using:

|  |  |
| --- | --- |
|  | toolpath.Name = "Key Toolpath"; |
|  |  |
|  | toolpath.Name = "Key Toolpath" |

To check if the toolpath has been calculated and to calculate it if it has not, use the following:

|  |  |
| --- | --- |
|  | if (toolpath.IsCalculated == false)  {  toolpath.Calculate();  } |
|  |  |
|  | If toolpath.IsCalculated = False Then  toolpath.Calculate()  End If |

#### Boundaries and Patterns

Consider the case of a PowerMill session containing an empty boundary and a set of toolpaths to which this boundary is applied. You may wish to add a DGK file to the empty boundary and recalculate the toolpaths, an operation that can be performed in the following manner:

1. Modify your main method so that it resembles the following.

|  |  |
| --- | --- |
|  | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using Autodesk.ProductInterface.PowerMill;  namespace PMillWalkthrough  {  class Program  {  static void Main(string[] args)  {  PMAutomation powerMill = new PMAutomation(Autodesk.ProductInterface.InstanceReuse.UseExistingInstance);  }  }  } |
|  |  |
|  | Imports Autodesk.ProductInterface.PowerMill  Module Module1  Sub Main()  Dim powerMill As New PMAutomation(Autodesk.ProductInterface.InstanceReuse.UseExistingInstance)  End Sub  End Module |

1. Now load a PowerMill project and create a PMProject object with which to manipulate it.

|  |  |
| --- | --- |
|  | Autodesk.FileSystem.Directory projectDirectory = new Autodesk.FileSystem.Directory(@"C:\PowerShape and PowerMill API Data\the API PowerMill Examples\Boundary Example");  powerMill.LoadProject(projectDirectory);  PMProject session = powerMill.ActiveProject; |
|  |  |
|  | Dim projectDirectory As New Autodesk.FileSystem.Directory("C:\PowerShape and PowerMill API Data\the API PowerMill Examples\Boundary Example")  powerMill.LoadProject(projectDirectory)  Dim session As PMProject = powerMill.ActiveProject |

* The Boundary Example folder will be located in a path relative to that which you extracted your example files to earlier and may be different to that specified above.

1. As before, turn off dialogs before proceeding futher.

|  |  |
| --- | --- |
|  | powerMill.DialogsOff(); |
|  |  |
|  | powerMill.DialogsOff() |

1. Now initialise a the API boundary object with the single boundary available in the PowerMill project.

|  |  |
| --- | --- |
|  | PMBoundary myBoundary = session.Boundaries[0]; |
|  |  |
|  | Dim myBoundary As PMBoundary = session.Boundaries(0) |

* You will need to know the index of the boundary to create an instance of it or, alternatively, the name of the boundary, which can be used to achieve the same result thus: session.Boundaries.get\_Item(<name>).

1. Now import from file the boundary that is to be used. This may be accomplished using the following code:

|  |  |
| --- | --- |
|  | Autodesk.FileSystem.File boundaryPath = new Autodesk.FileSystem.File(@"C:\Users\asr\Desktop\PowerShape and PowerMill API Data\the API PowerMill Examples\BoundaryFile.dgk");  myBoundary.InsertFile(boundaryPath); |
|  |  |
|  | Dim boundaryPath As New Autodesk.FileSystem.File("C:\Users\asr\Desktop\PowerShape and PowerMill API Data\the API PowerMill Examples\BoundaryFile.dgk")  myBoundary.InsertFile(boundaryPath) |

* BoundaryFile.dgk will be located in the path to which you extracted your example files earlier and may be different to that specified above.

1. With the new boundary applied, create an instance of the toolpath and calculate it:

|  |  |
| --- | --- |
|  | PMToolpath myToolpath = session.Toolpaths[0];  myToolpath.Calculate(); |
|  |  |
|  | Dim myToolpath As PMToolpath = session.Toolpaths(0)  myToolpath.Calculate() |

#### Executing Macro Commands

In some instances there may be a requirement for functionality that does not exist within the current version of the API; in these cases macro commands can be issued from within your code. Methods for achieving this are detailed below.

1. To issue a macro command to create a blank pattern, for example, use the following:

|  |  |
| --- | --- |
|  | powerMill.Execute("CREATE PATTERN ;"); |
|  |  |
|  | powerMill.Execute("CREATE PATTERN ;") |

1. Alternatively you may wish to acquire information from PowerMill to use within your program. If, for example, you wished to retrieve the units PowerMill is currently running in, issue the following:

|  |  |
| --- | --- |
|  | powerMill.ExecuteEx("PRINT PAR TERSE ‘Units’"); |
|  |  |
|  | powerMill.ExecuteEx("PRINT PAR TERSE ‘Units’") |

1. It is important to note that entities created within PowerMill using macro commands will not appear within the the API cache until it is updated. To update the cache, issue the following command:

|  |  |
| --- | --- |
|  | session.Refresh(); |
|  |  |
|  | session.Refresh() |

1. It is now possible to create in code an instance of the PowerMill object that was created with the macro command. If, for example, it was required to create a tool within PowerMill using a macro command and then instantiate the new tool in code, the following could be used:

|  |  |
| --- | --- |
|  | powerMill.Execute("CREATE TOOL ; ENDMILL FORM TOOL");  powerMill.Execute("EDIT TOOL '1' DIAMETER '12.0'");  powerMill.Execute("TOOL ACCEPT");  session.Refresh();  PMTool tool = session.Tools[0]; |
|  |  |
|  | powerMill.Execute("CREATE TOOL ; ENDMILL FORM TOOL")  powerMill.Execute("EDIT TOOL '1' DIAMETER '12.0'")  powerMill.Execute("TOOL ACCEPT")  session.Refresh()  Dim tool As PMTool = session.Tools(0) |

* It is strongly recommended that you always use available the API functionality in preference to issuing raw macro commands. If you feel that the API is lacking the functionality you require, please contact the forum at:

<http://forum.delcam.com/viewforum.php?f=53>

#### Deploying Solutions

Once your development is complete, you will need a way of deploying your application to end users. This section will guide you through the process of creating an installer.

1. Begin by ensuring that the *Microsoft Visual Studio Installer Projects* extension is installed within your copy of Microsoft Visual Studio. If not, select ‘Tools’ followed by ‘Extensions and Updates’; now locate the extension from within the resulting window and install it.
2. Next, right click on your solution and select ‘Add’ then ‘New Project’ (Figure 6).

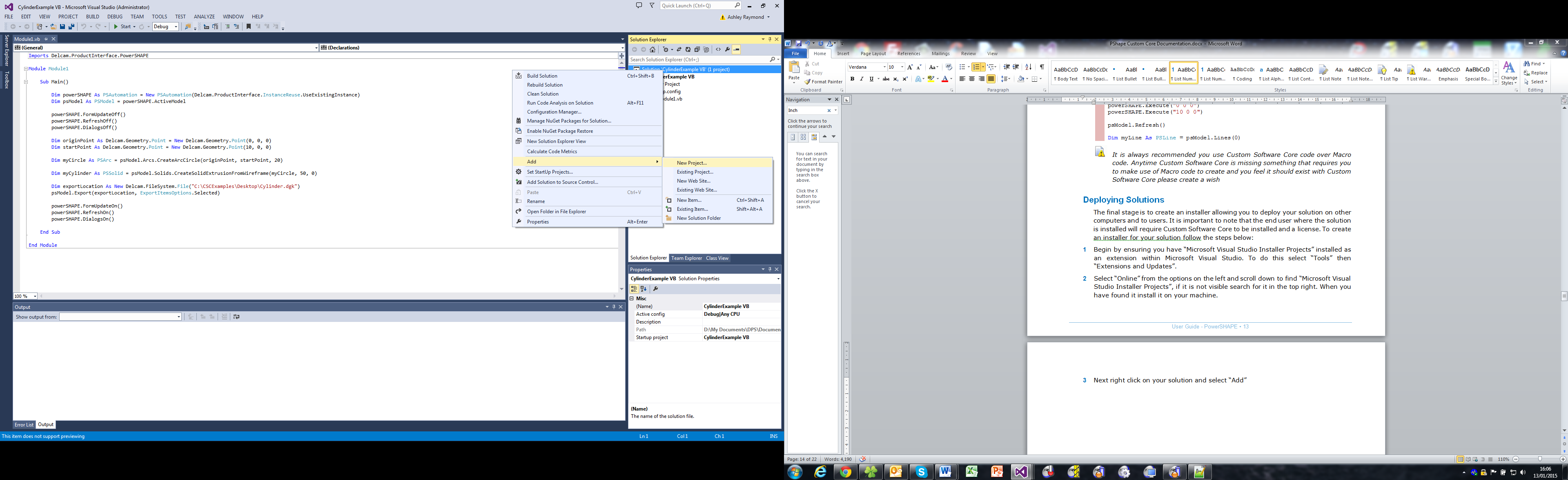


Figure – Adding a new project

1. Select ‘Other Project Types’ followed by ‘Visual Studio Installer’ from the left-hand-side of the new window; then select ‘Setup Project’.
2. Now configure a Manufacturer and Product Name. Select ‘Setup1’ from within Solution Explorer and edit the Manufacturer and ProductName fields in the Properties tab as demonstrated in figure 7. Note: If the Properties tab is not visible, select it from the View menu.

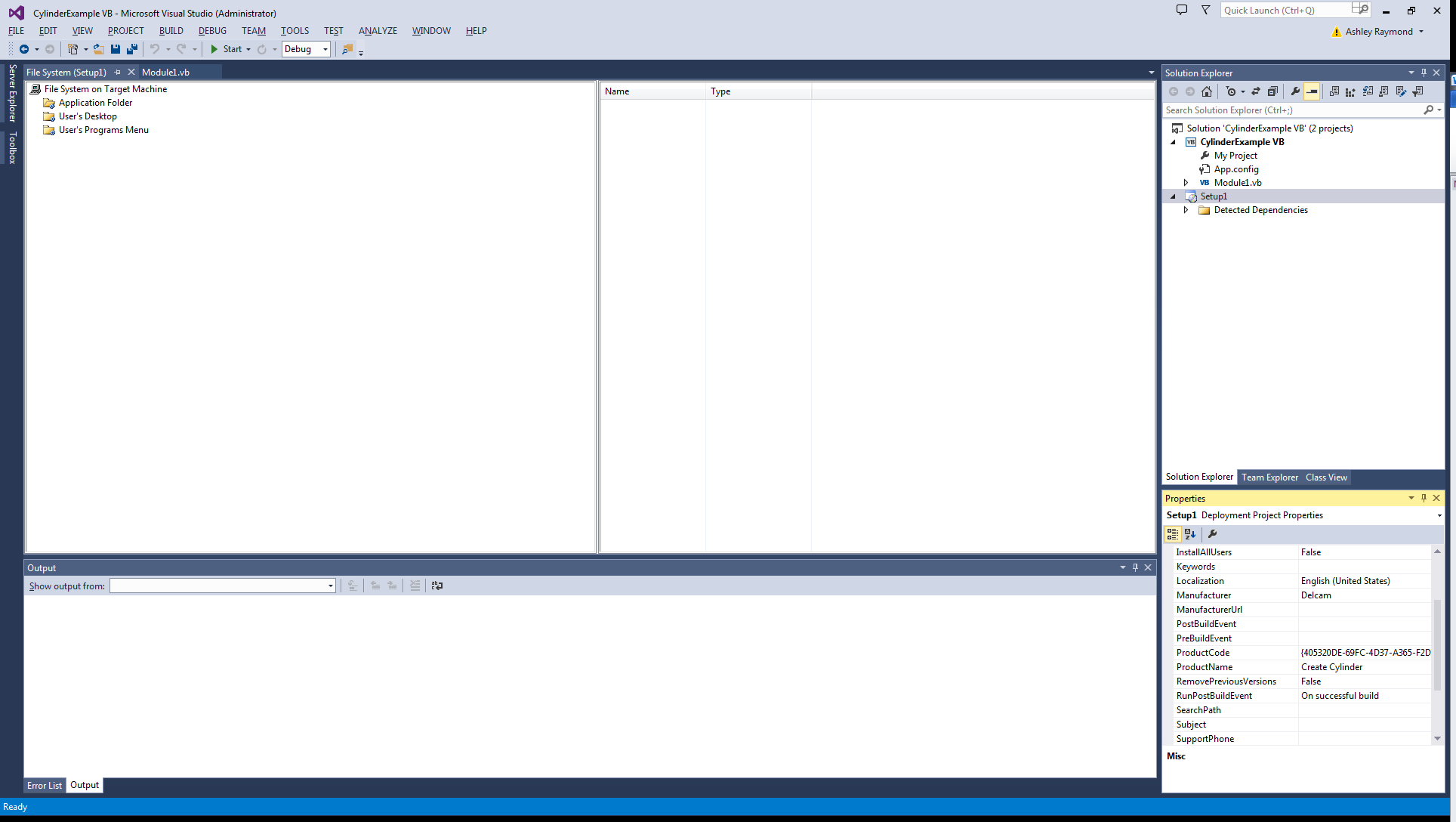


Figure – Setting Manufacturer, ProductName

1. Right click the ‘Application Folder’ from within the File System window to the right of Solution Explorer. Select ‘Add’ and then select ‘Project Output’ from the resultant context menus (Figure 8).

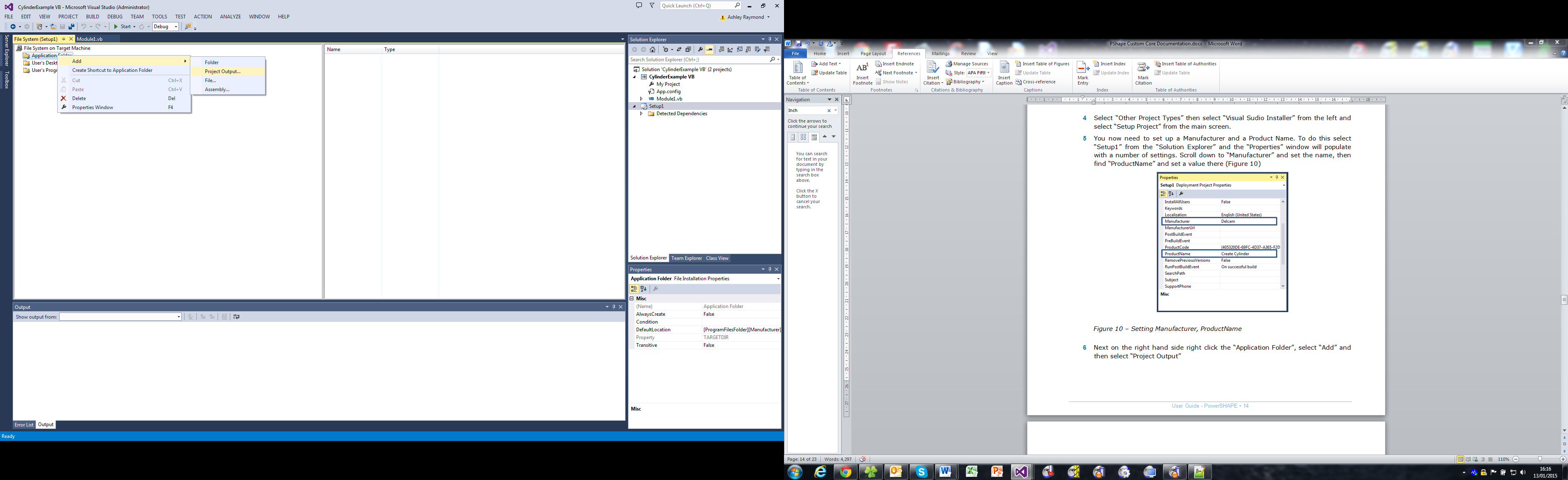


Figure – Adding a project output

* If you cannot see the File System window, select ‘File System Editor’ from the Solution Explorer toolbar (Figure 9).

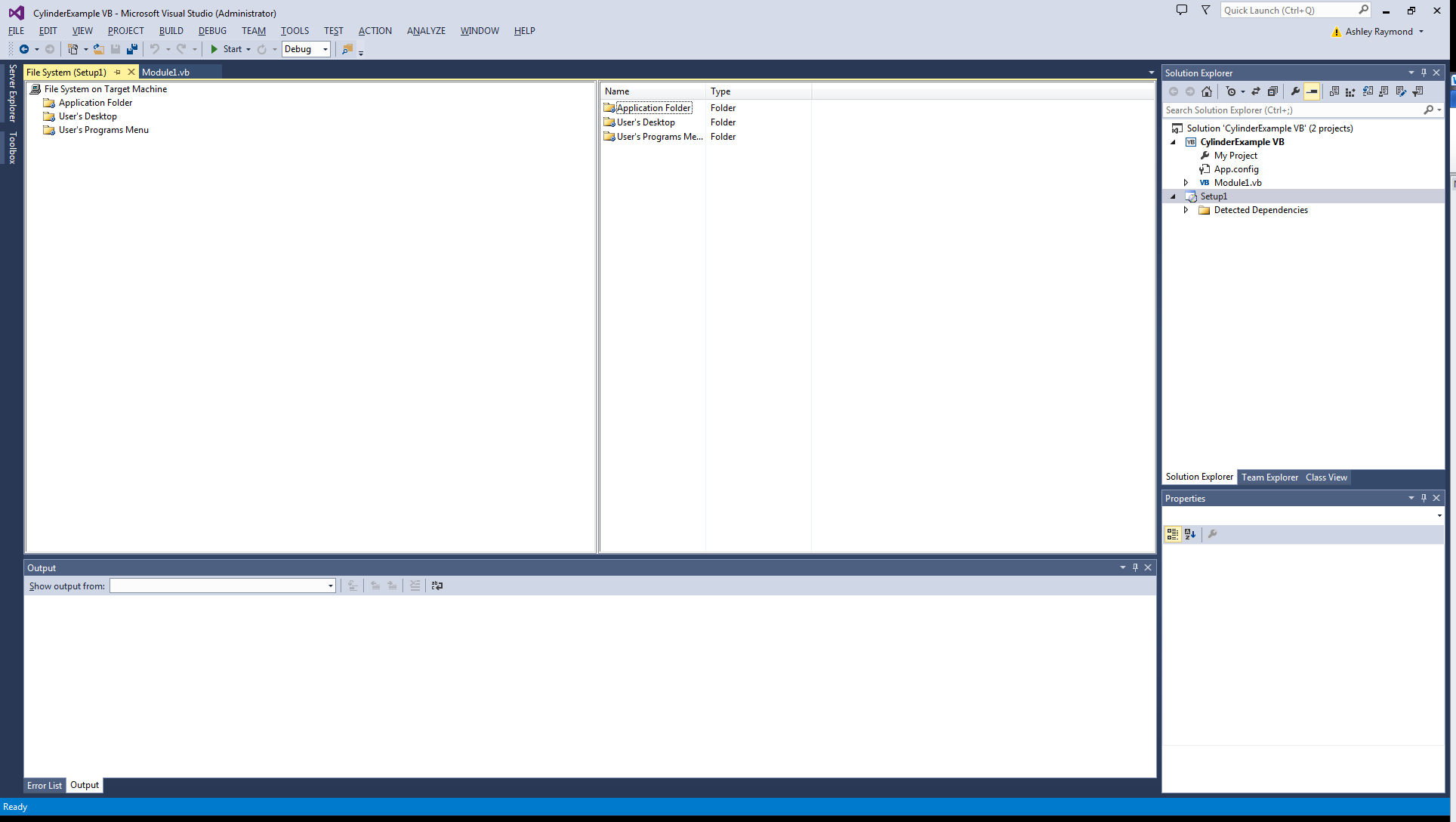


Figure – File System Editor

1. From the dialog box, select ‘Primary Output’ and click ‘Ok’.
2. You will notice references appear in the right hand box (Figure 10).

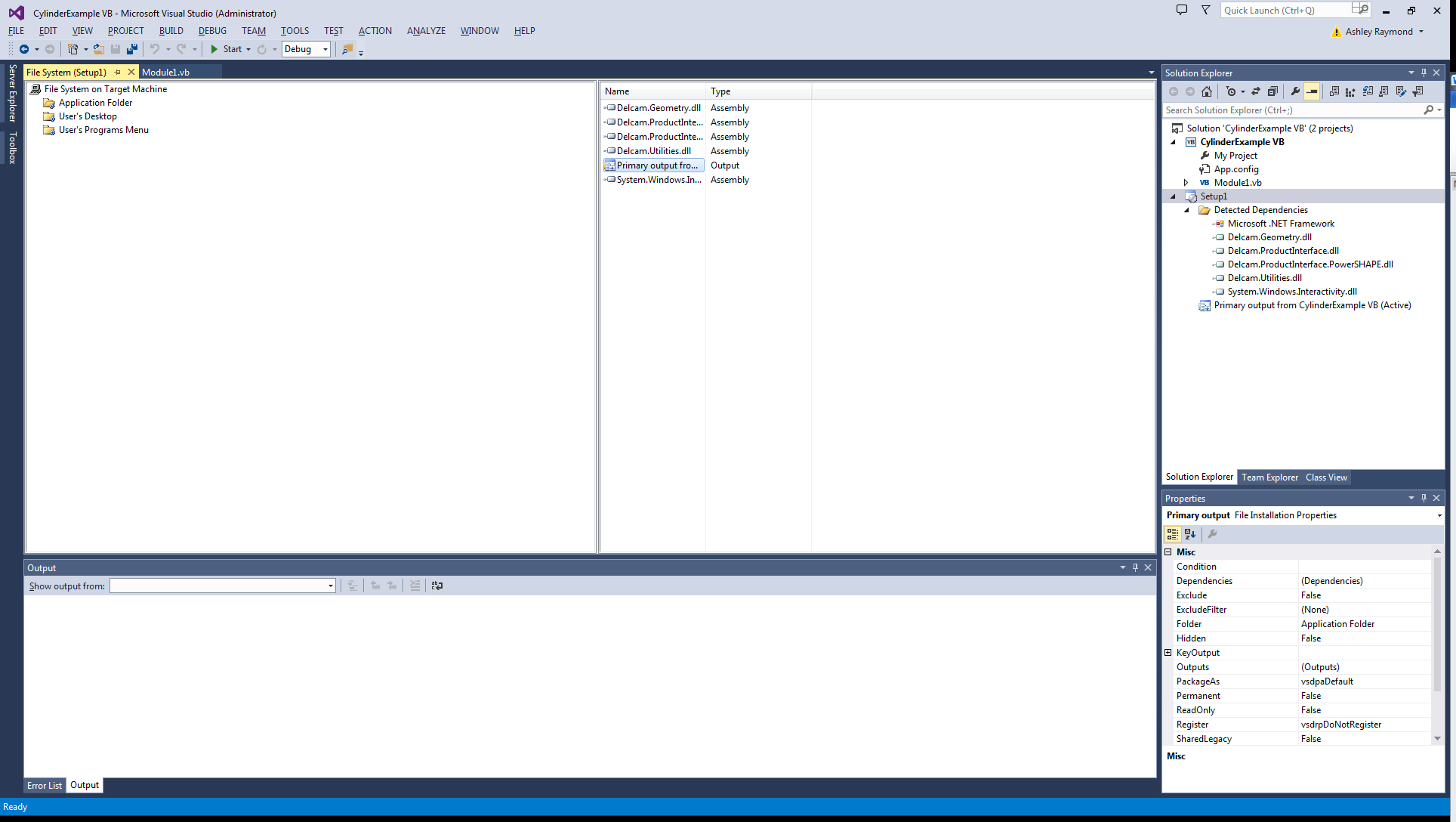


Figure – Adding References

1. As it is a requirement that the end user install the API, the Delcam references are not required and should be excluded from the project. Select each in turn and set ‘Exclude’ to True in the Properties tab.
2. If it is desired to add a shortcut to the start menu then right click ‘User’s Program’s Menu’ and select ‘Add’ followed by ‘Folder’; name the folder appropriately (Figure 11).

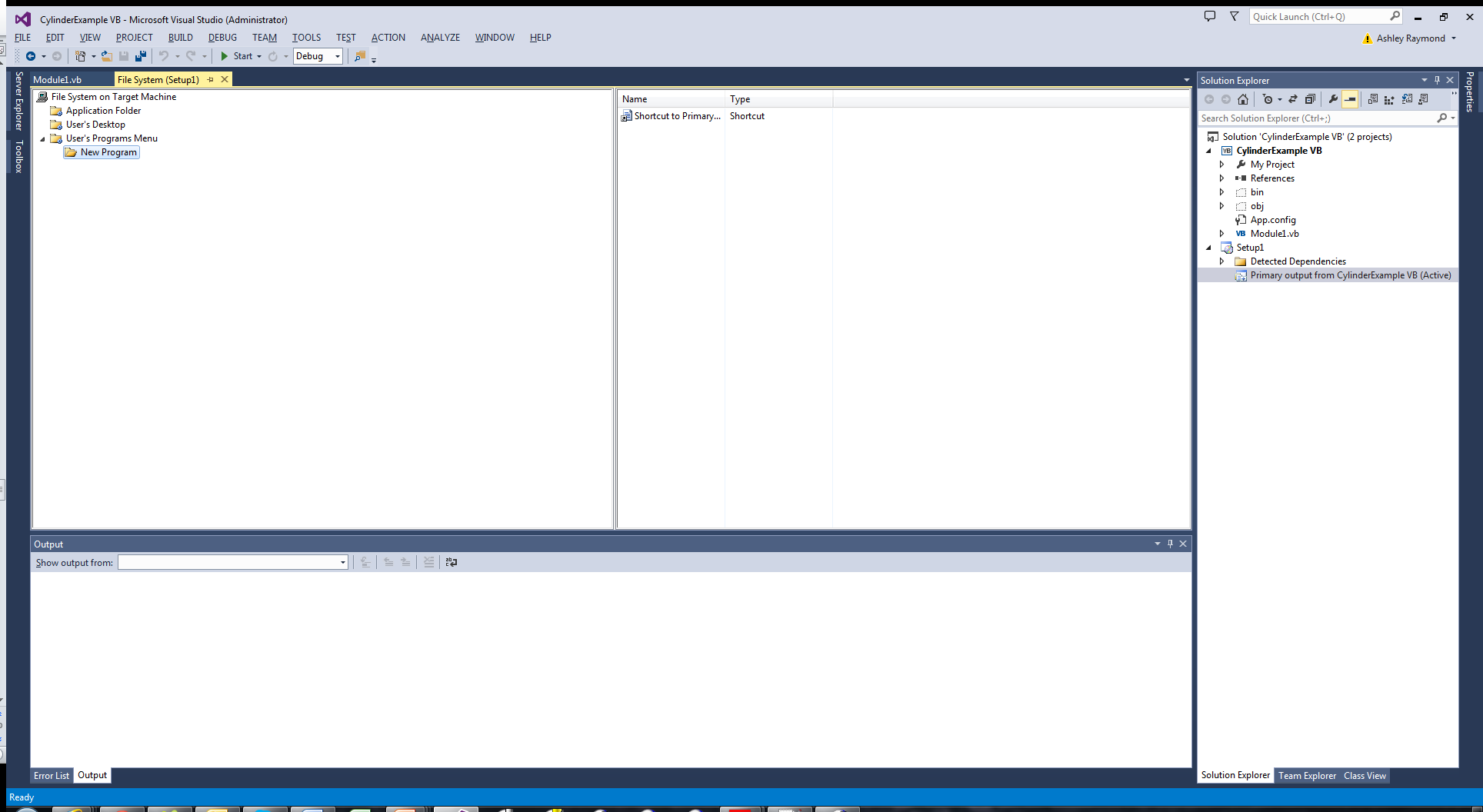


Figure – Set an install folder

1. Right click the newly created folder and select ‘Create New Shortcut’.
2. Select the new folder and, in the empty panel to the right, right click and select ‘Create new shortcut’.

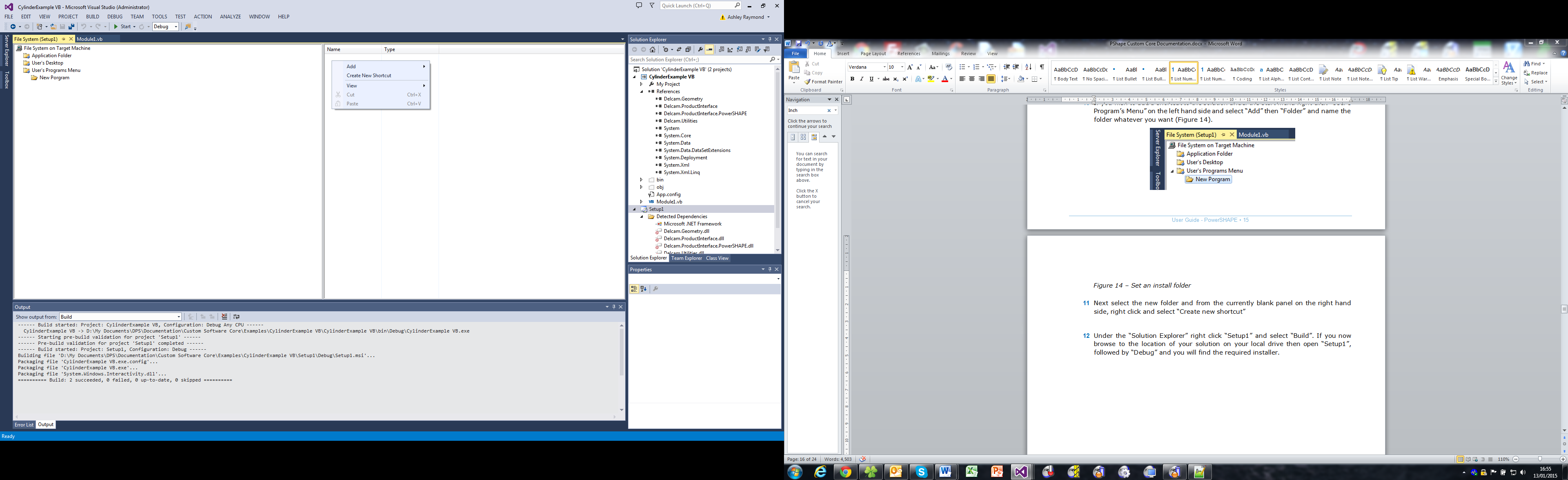


Figure – Adding shortcut to program menu

1. In the resulting dialog, double click the ‘Application Folder’ and select the item starting with ‘Primary output from…’ Rename it to whatever you wish.
2. To create the installer, right click ‘Setup1’ from within Solution Explorer and select ‘Build’. Once the operation has completed, browse to the location of the solution on the local drive, open Setup1 and then browse to either the Release or Debug folder - depending upon your solution settings – and the installer will be available therein.

## Examples

#### Example 1 - Using Template Files

You will often find it is easier to work with PowerMill templates containing predefined tools, patterns and toolpaths rather than creating these items from macro code. Using this method will mean you can make simple changes to your template such as changing a tool diameter and this new information will be applied every time you run your macro rather than having to edit the code behind. As an example we will look at starting a new session, opening a template that contains a number of toolpaths with settings previously determined and a tool, importing a model and re-calculating toolpaths and exporting the NC code.

1. Close all PowerMill instances and launch a new one.
2. Select ‘Insert’ from the PowerMill main menu and ‘Template Object’ from the drop-down. From within the resulting dialog, browse the location to which was extracted the supplied sample data and select ‘Template\_Example.ptf’. Click ‘Open’.
3. From within the PowerMill explorer tree, open the preferences form for ‘NC programs’ and set the ‘Output Folder’ to a location on your desktop. All other preferences will be set from within code.
4. Save the template by selecting ‘File’ from the PowerMill main menu and ‘Save Template Objects…’ from the drop-down. Specify a filename and location to which to save the template file. This can be the same as the original if so desired.
5. Launch Microsoft Visual Studio.
6. Create a new console application project and save it to a location of your choice.
7. Add the following references:

* Autodesk.Geometry
* Autodesk.ProductInterface
* Autodesk.ProductInterface.PowerMill
* Autodesk.Utilities

1. Add the following using directive to the top of the source file containing the main method.

|  |  |
| --- | --- |
|  | using Autodesk.ProductInterface.PowerMill; |
|  |  |
|  | Imports Autodesk.ProductInterface.PowerMill |

1. Add the following line to the main method:

|  |  |
| --- | --- |
|  | PMAutomation powerMILL = new PMAutomation (Autodesk.ProductInterface.InstanceReuse.UseExistingInstance); |
|  |  |
|  | Dim powerMILL As New PMAutomation(Autodesk.ProductInterface.  InstanceReuse.UseExistingInstance) |

* When you run your code make sure PowerMill is open as we have chosen to use an existing instance rather than open a new one.

1. Attach to the PowerMill active project by adding the following code to the main method of the source file:

|  |  |
| --- | --- |
|  | PMProject session = powerMILL.ActiveProject; |
|  |  |
|  | Dim session As PMProject = powerMILL.ActiveProject |

1. Turn off dialogs.

|  |  |
| --- | --- |
|  | powerMill.DialogsOff(); |
|  |  |
|  | powerMill.DialogsOff() |

* Dialogs can be reactivated with PowerMill.DialogsOn().

Create a Autodesk.FileSystem.File representing the location of the template file:

|  |  |
| --- | --- |
|  | Autodesk.FileSystem.File pTemplate = new Autodesk.FileSystem.File(@"C:\the API PowerMill Examples\Template\_Example.ptf"); |
|  |  |
|  | Dim pTemplate As New Autodesk.FileSystem.File("C:\the API PowerMill Examples\Template\_Example.ptf") |

* Template\_Example.ptf will be located in the path to which you extracted your example files earlier and may be different to that specified above.

1. The template can now be imported into the PowerMill session:

|  |  |
| --- | --- |
|  | session.ImportTemplateFile(pTemplate); |
|  |  |
|  | session.ImportTemplateFile(pTemplate) |

1. Add a model to the PowerMill session:

|  |  |
| --- | --- |
|  | Autodesk.FileSystem.File modelPath = new Autodesk.FileSystem.File(@"C:\the API PowerMill Examples\Pocket-Feature.dgk");  PMModel myModel = session.Models.CreateModel(modelPath); |
|  |  |
|  | Dim modelPath As New Autodesk.FileSystem.File("C:\the API PowerMill Examples\Pocket-Feature.dgk")  Dim myModel As PMModel = session.Models.CreateModel(modelPath) |

* Pocket-Feature.dgk will be located in the path to which you extracted your example files earlier and may be different to that specified above.

1. Select the model from the session and create a block. This requires a mixture of the API and macro code:

|  |  |
| --- | --- |
|  | myModel.AddToSelection(true);  powerMILL.Execute("FORM BLOCK");  powerMILL.Execute("EDIT BLOCK RESET");  powerMILL.Execute("BLOCK ACCEPT"); |
|  |  |
|  | myModel.AddToSelection(True)  powerMILL.Execute("FORM BLOCK")  powerMILL.Execute("EDIT BLOCK RESET")  powerMILL.Execute("BLOCK ACCEPT") |

1. Now calculate each toolpath in the template and set the correct rapid heights by cycling through each toolpath in turn:

|  |  |
| --- | --- |
|  | foreach (PMToolpath toolpath in session.Toolpaths)  {  toolpath.IsActive = true;  toolpath.Calculate();  powerMILL.Execute("FORM TOOLZHEIGHTS");  powerMILL.Execute("EDIT TOOLPATH SAFEAREA RESET");  powerMILL.Execute("RESET TOOLPATH RAPID");  powerMILL.Execute("TOOLZHEIGHTS ACCEPT");  toolpath.IsActive = false;  } |
|  |  |
|  | For Each toolpath As PMToolpath In session.Toolpaths  toolpath.IsActive = True  toolpath.Calculate()  powerMILL.Execute("FORM TOOLZHEIGHTS")  powerMILL.Execute("EDIT TOOLPATH SAFEAREA RESET")  powerMILL.Execute("RESET TOOLPATH RAPID")  powerMILL.Execute("TOOLZHEIGHTS ACCEPT")  toolpath.IsActive = False  Next |

1. Each toolpath can now be posted. This will be accomplished by assigning sequential names through use of a counter:

|  |  |
| --- | --- |
|  | Int64 count = 1; |
|  |  |
|  | Dim count As Integer = 1 |

1. Since the NC programs already exist within the session, we need only give them an option file, an output workplane, a name and then write them:

|  |  |
| --- | --- |
|  | foreach (PMNCProgram sNCProgram in session.NCPrograms)  {  sNCProgram.MachineOptionFileName = @"C:\the API PowerMill Examples\DMU160P\_S840D\_JM\_Rev7.pmoptz";  sNCProgram.OutputWorkplaneName = "OWP";  sNCProgram.Name = "O00" + count.ToString();  sNCProgram.Write();  count = count + 1;  } |
|  |  |
|  | For Each sNCProgram As PMNCProgram In session.NCPrograms  sNCProgram.MachineOptionFileName = "C:\the API PowerMill Examples\DMU160P\_S840D\_JM\_Rev7.pmoptz"  sNCProgram.OutputWorkplaneName = "OWP"  sNCProgram.Name = "O00" + count.ToString()  sNCProgram.Write()  count = count + 1  Next |

* You will need to set a location of an available Post Processor.

1. Dialogs can now be re-enabled if so desired and the project executed:

|  |  |
| --- | --- |
|  | powerMill.DialogsOn(); |
|  |  |
|  | powerMill.DialogsOn() |

The complete code listing is as given below:

|  |  |
| --- | --- |
|  | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using Autodesk.ProductInterface.PowerMill;  namespace UsingTemplateFiles  {  class Program  {  static void Main(string[] args)  {  PMAutomation powerMILL = new PMAutomation(Autodesk.ProductInterface.InstanceReuse.UseExistingInstance);  PMProject session = powerMILL.ActiveProject;  powerMILL.DialogsOff();  Autodesk.FileSystem.Directory savePath = new Autodesk.FileSystem.Directory(@"C:\Users\asr\Desktop\the API PowerMill Examples\TemplateSession");  Autodesk.FileSystem.File pTemplate = new Autodesk.FileSystem.File(@"C:\Users\asr\Desktop\the API PowerMill Examples\Template\_Example.ptf");  session.ImportTemplateFile(pTemplate);  session.Initialise();  Autodesk.FileSystem.File modelPath = new Autodesk.FileSystem.File(@"C:\Users\asr\Desktop\the API PowerMill Examples\Pocket-Feature.dgk");  PMModel myModel = session.Models.CreateModel(modelPath);  myModel.AddToSelection(true);  powerMILL.Execute("FORM BLOCK");  powerMILL.Execute("EDIT BLOCK RESET");  powerMILL.Execute("BLOCK ACCEPT");  foreach (PMToolpath toolpath in session.Toolpaths)  {  toolpath.IsActive = true;  toolpath.Calculate();  powerMILL.Execute("FORM TOOLZHEIGHTS");  powerMILL.Execute("EDIT TOOLPATH SAFEAREA RESET");  powerMILL.Execute("RESET TOOLPATH RAPID");  powerMILL.Execute("TOOLZHEIGHTS ACCEPT");  toolpath.IsActive = false;  }  Int64 count = 1;  foreach (PMNCProgram sNCProgram in session.NCPrograms)  {  sNCProgram.MachineOptionFileName = @"C:\Users\asr\Desktop\the API PowerMill Examples\DMU160P\_S840D\_JM\_Rev7.pmoptz";  sNCProgram.OutputWorkplaneName = "OWP";  sNCProgram.Name = "O00" + count.ToString();  sNCProgram.Write();  count = count + 1;  }  powerMILL.DialogsOn();  }  }  } |
|  |  |
|  | Imports Autodesk.ProductInterface.PowerMill  Module Module1  Sub Main()  Dim powerMILL As New PMAutomation(Autodesk.ProductInterface.InstanceReuse.UseExistingInstance)  Dim session As PMProject = powerMILL.ActiveProject  powerMILL.DialogsOff()  Dim pTemplate As New Autodesk.FileSystem.File("C:\the API PowerMill Examples\Template\_Example.ptf")  session.ImportTemplateFile(pTemplate)  Dim modelPath As New Autodesk.FileSystem.File("C:\the API PowerMill Examples\Pocket-Feature.dgk")  Dim myModel As PMModel = session.Models.CreateModel(modelPath)  myModel.AddToSelection(True)  powerMILL.Execute("FORM BLOCK")  powerMILL.Execute("EDIT BLOCK RESET")  powerMILL.Execute("BLOCK ACCEPT")  For Each toolpath As PMToolpath In session.Toolpaths  toolpath.IsActive = True  toolpath.Calculate()  powerMILL.Execute("FORM TOOLZHEIGHTS")  powerMILL.Execute("EDIT TOOLPATH SAFEAREA RESET")  powerMILL.Execute("RESET TOOLPATH RAPID")  powerMILL.Execute("TOOLZHEIGHTS ACCEPT")  toolpath.IsActive = False  Next  Dim count As Integer = 1  For Each sNCProgram As PMNCProgram In session.NCPrograms  sNCProgram.MachineOptionFileName = "C:\the API PowerMill Examples\DMU160P\_S840D\_JM\_Rev7.pmoptz"  sNCProgram.OutputWorkplaneName = "OWP"  sNCProgram.Name = "O00" + count.ToString()  sNCProgram.Write()  count = count + 1  Next  End Sub  End Module |