LUSAS

LUSAS Programmable Interface (LPI)

Developer Guide

LUSAS Programmable Interface (LPI) Developer Guide

LUSAS Version 24.0 : Issue 1

LUSAS
Forge House, 66 High Street, Kingston upon Thames,
Surrey, KT1 1HN, United Kingdom

Tel: +44 (0)20 8541 1999
Fax +44 (0)20 8549 9399
Email: info@lusas.com
http://www.lusas.com

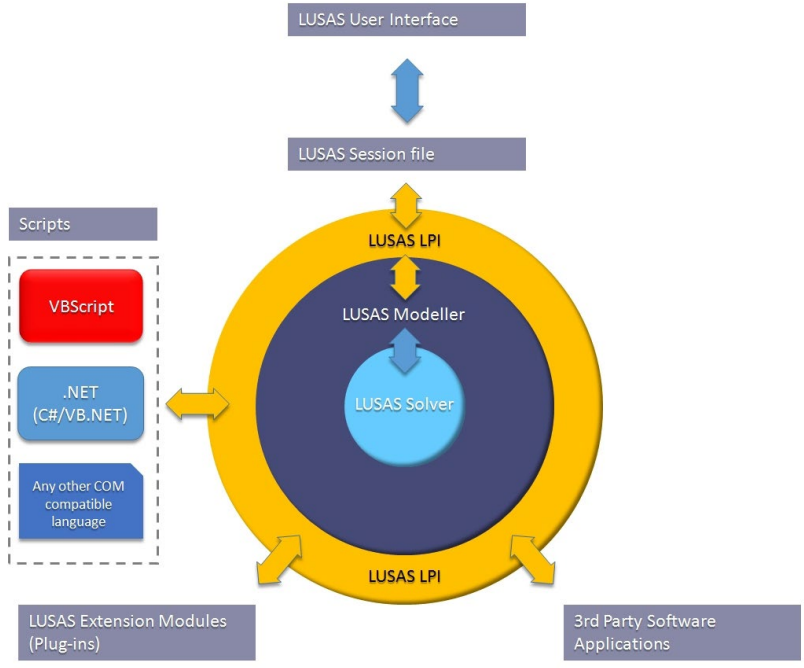
Distributors Worldwide

Table of Contents

Introduction

Introduction

LUSAS software is highly customisable. The built-in LUSAS Programmable Interface (LPI) allows the customisation and automation of modelling and results processing tasks and creation of user-defined menu items, dialogs and toolbars as a means to access those user-defined resources. It can also be used for transferring data between LUSAS and other software applications, and to control other programs from within LUSAS Modeller, or control LUSAS Modeller from other programs.



With LPI, any user can automate the creation of complete structures, either in LUSAS or from third-party software, carrying out design checks, optimising members and outputting graphs, spreadsheets of results and custom reports. Because everything carried out by a user is recorded in a LUSAS Modeller session file, anything that LUSAS can do, can also be controlled by another application via the LUSAS

Programmable Interface. This means that you can view and edit a recorded session, parameterise those commands, turn them into sub-routines, add loops and other functions to the scripts and create a totally different application or program - using the proven core technology of LUSAS.

In addition to the accessing and customising LUSAS Modeller via the LUSAS Programmable Interface, user-defined material models (written in Fortran) can be compiled and built into a customised LUSAS Solver executable by using the LUSAS Material Model Interface (LUSAS MMI).

Topics covered in this guide

The aim of this guide is to help you use the more advanced facilities available to customise LUSAS and interface with other applications. The guide covers:

- Creating dialogs using VB.NET
- LUSAS via COM
- LUSAS Material Model Interface

LUSAS Programmable Interface (LPI) Customisation and Automation Guide

A separate LUSAS Programmable Interface (LPI) Customisation and Automation Guide is also available covering more basic topics:

- Getting started with the LUSAS Programmable Interface (LPI)
- Identifying LPI Functions
- Customising the interface
- Getting started with VBS
- A simple example script
- Creating your own menus

Creating dialogs using VB.NET

A dialog is a window that appears on the display screen to neatly present information or request input from the user. Creating dialogs, and some other advanced features, such as object oriented programming, is a more advanced topic, and some programming skills are required.

VBScript belongs to the group of interpreted languages, meaning that there is no need for compilers to be used. This is because the implementations execute directly, line by line. Examples of other interpreted languages are JavaScript, Python and Perl.

To create a dialog you will need to use a compiled programming language. Examples of compiled languages are Visual Basic .NET (VB.NET), C#, Java, C++. For this example VB.NET will be used, because although it is a different language to VBScript, it uses the same syntax, and you should be already familiar with it. Also, and more importantly, you do not need to be familiar with COM programming, as you would if C++ was used instead.

Advantages of VB.NET over VBScript include

- Better user-interface creation
- Debugging with break points
- More extensible. Allows Object Oriented Programming (OOP)
- Multiple functions saved into one .dll file. Easier to share across the company than multiple .vbs script files
- Easier to do testing

Choosing a development environment

To use VB.NET a compiler needs to be installed. The simplest way to achieve this is to install Visual Studio Community edition. Alternatively, the dotnet SDK (Software Development Kit) can be installed and used with other editors such as VSCode. This guide is written around the use of Visual Studio Community edition.

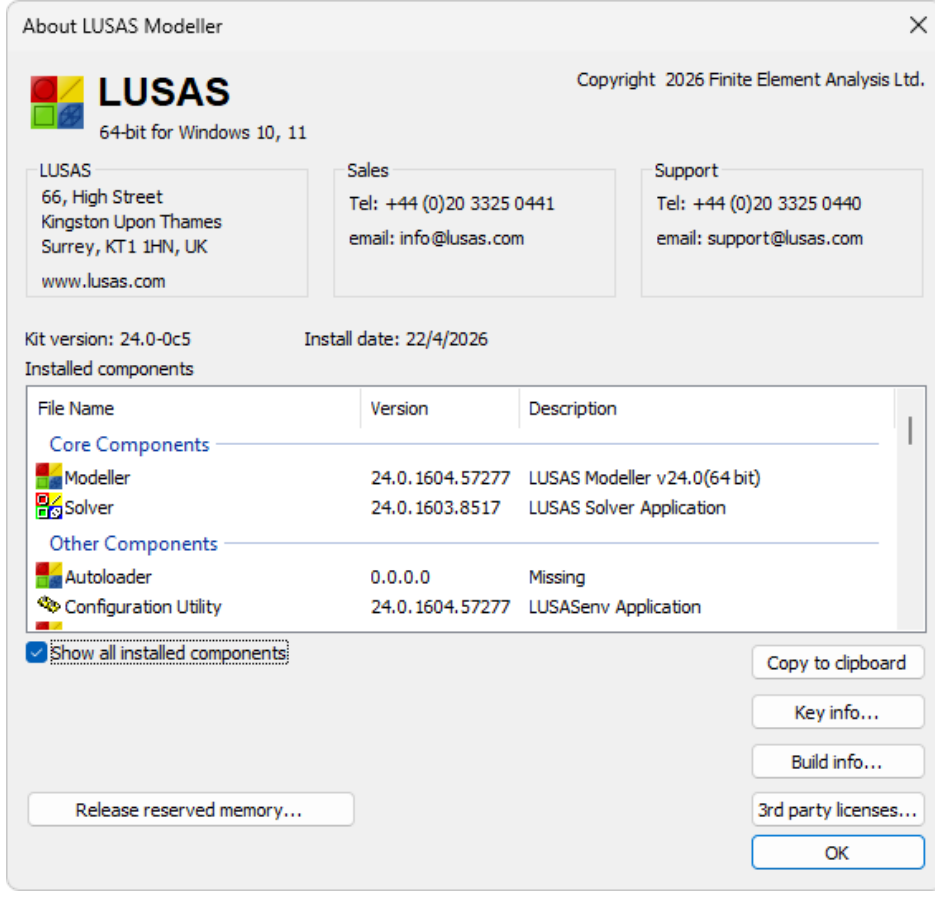
Visual Studio Community is free for individual developers, open-source projects, academic research, education, and small professional teams. Users should be aware of the licensing terms under which it is supplied.

Visual Studio Community be downloaded from:

https://www.visualstudio.com/downloads

Creating a LUSAS dialog

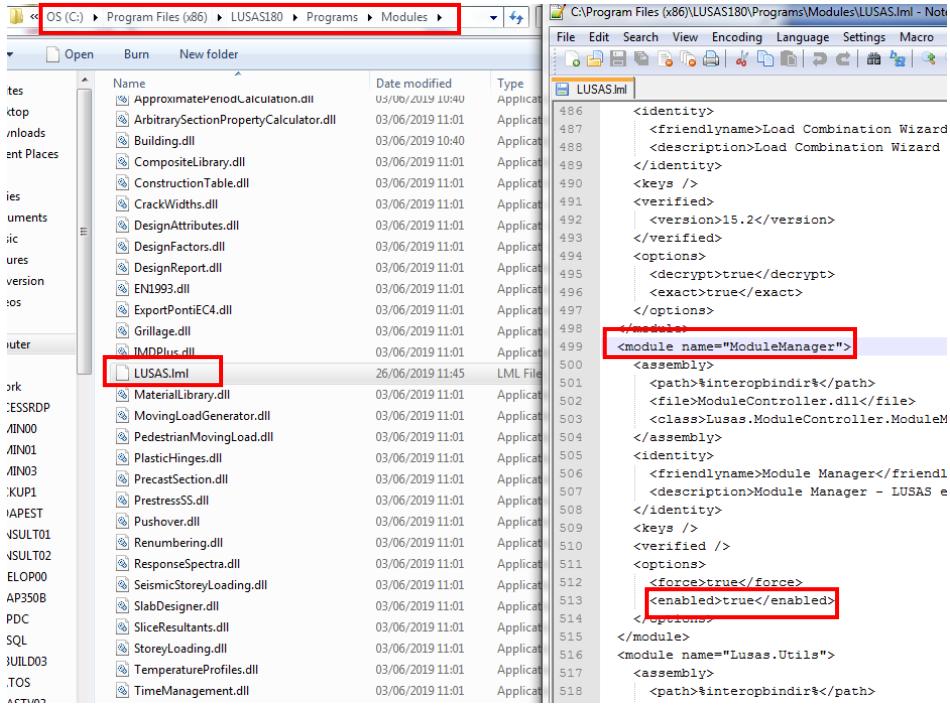
To create a LUSAS dialog, you will need to add a new Module to LUSAS. Modules are like plugins. All the installed modules can be seen by selecting the menu item **Help > About LUSAS Modeller…** and checking **Show all installed components**



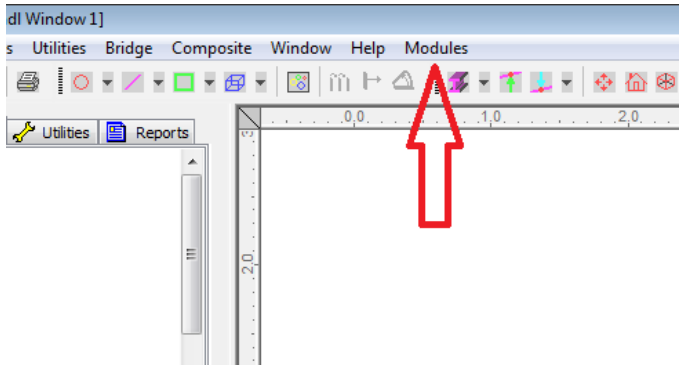
Module Manager

LUSAS Modeller's Modules are controller by the Module Manager and the Module Manager dialog can be displayed by following the steps below.

1. Open the file C:\Program Files\LUSAS<version_number>\Programs\Modules\LUSAS.lml into a text file editor.
2. Search for the text ModuleManager
3. Enable it by changing false to true as shown below
4. Save the file.

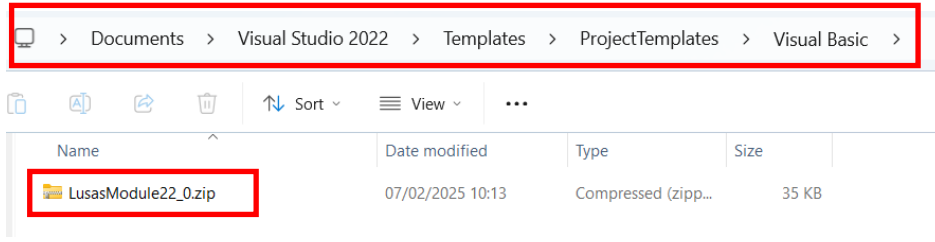


From now on, when you run LUSAS Modeller you will see the Modules menu:



Creating a new module

- Copy the LusasModule<version_number>.zip file from the LUSAS installation directory **C:\Programs Files (x86)\LUSAS220\Programs (x86)** to the Visual Studio project template folder. Typically: **Documents\Visual Studio 2019\Templates\ProjectTemplates\Visual Basic**

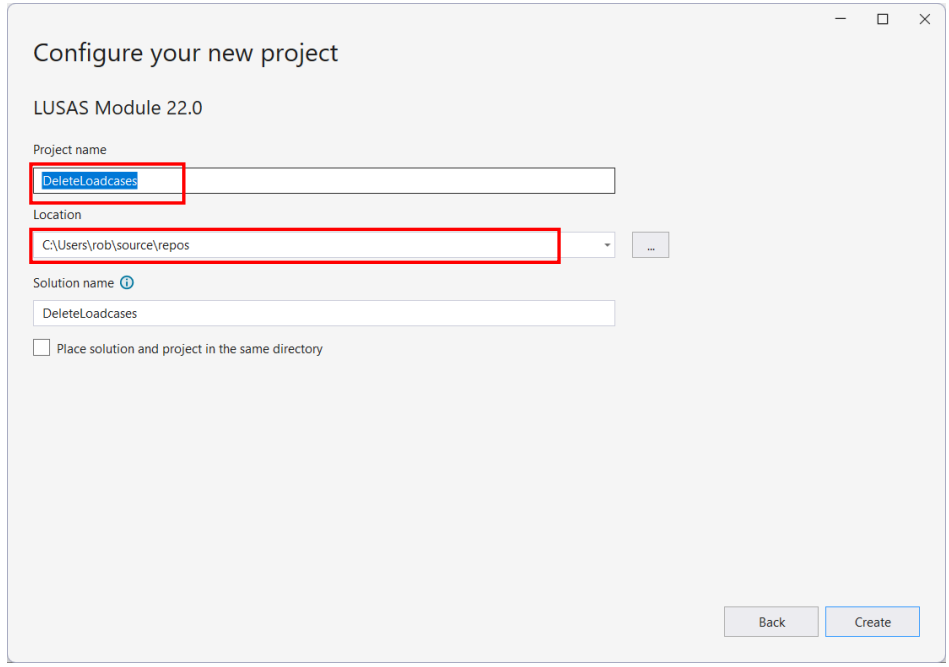


Note. If the project template does not show in Visual Studio when creating a project a repair of the Visual Studio installation may be required.

Running Visual Studio

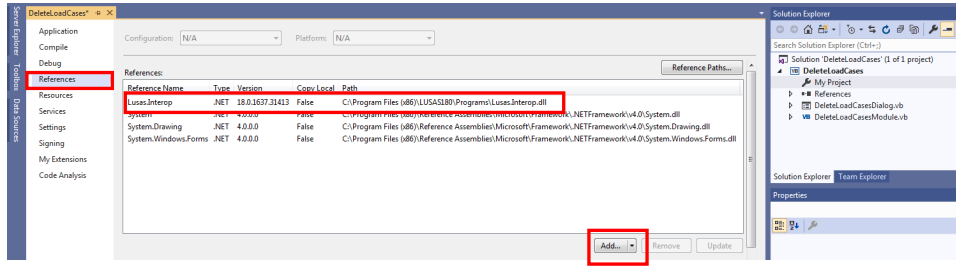
- Open Visual Studio and select **Create a new Project**
- In Visual Studio search “LUSAS” in the project templates and select **LUSAS Module<version number>** template for your version of LUSAS from the available list. Then click Next.
- The dialog example that will be covered in this guide creates a new module to delete loadcases, therefore enter the name **DeleteLoadcases** (without any space) and click OK.

Note. It is important to use a sensible name as this propagates throughout the automatically generated code.

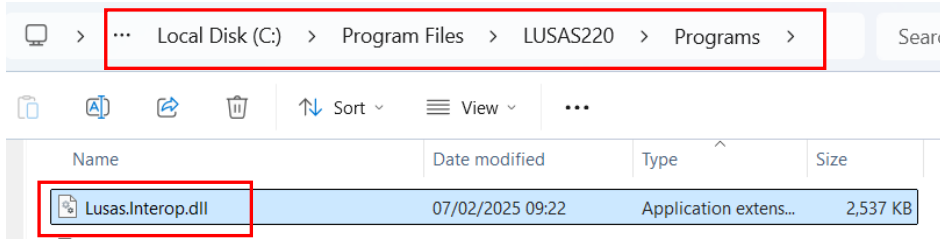


Note. If more than one version of LUSAS is installed on your machine you should check the version number of LUSAS that is being referenced by Visual Studio and ensure that this number is what is required. This can be done by checking the path to the LusasInterop dll.

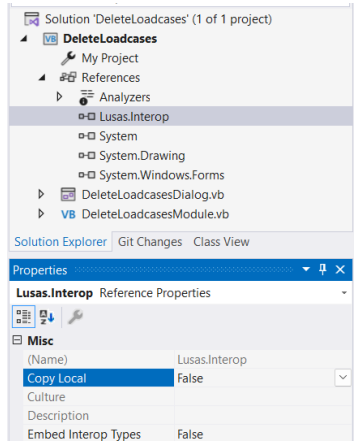
- On the Solution Explorer double-click on **My Project** and visually check the path for LusasInterop.dll



- If the wrong version of LUSAS is being referenced, remove the existing reference, then click **Add** and locate the **Lusas.Interop.dll** file in the right version path

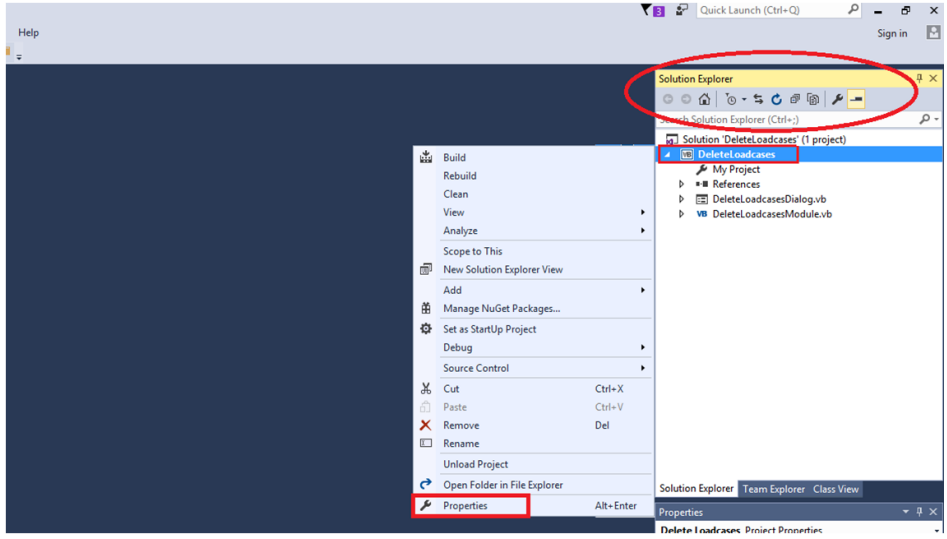


- Right click on the reference to **Lusas.Interop**, select properties and in the properties window ensure Copy Local=False and Embed Interop Types = False.

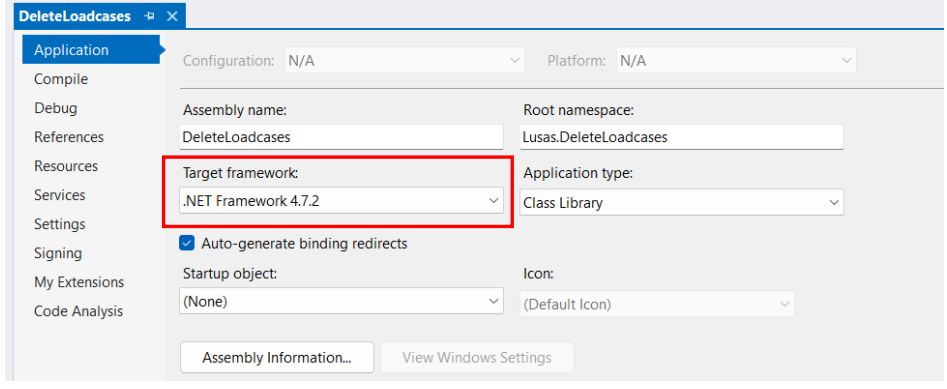


Carrying on:

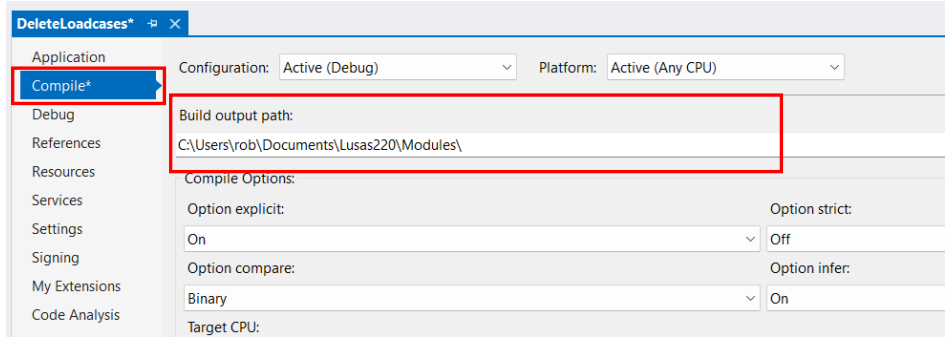
- In the Solution Explorer right-click on the project icon **DeleteLoadcases** and choose **Properties**



- On the Application page verify the Target framework is set to .NET Framework 4.7.2

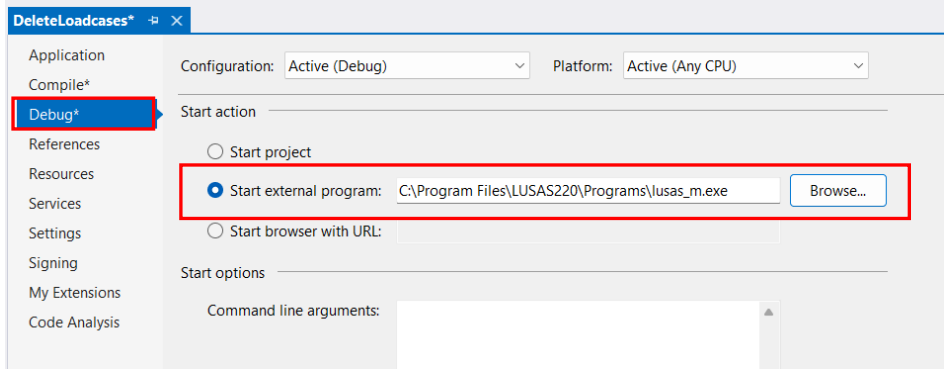


- On the Compile tab, browse and change the build output path to **Documents\LUSAS<version_number>\Modules.** This is the recommended folder for user modules, but any folder with write access can be used.



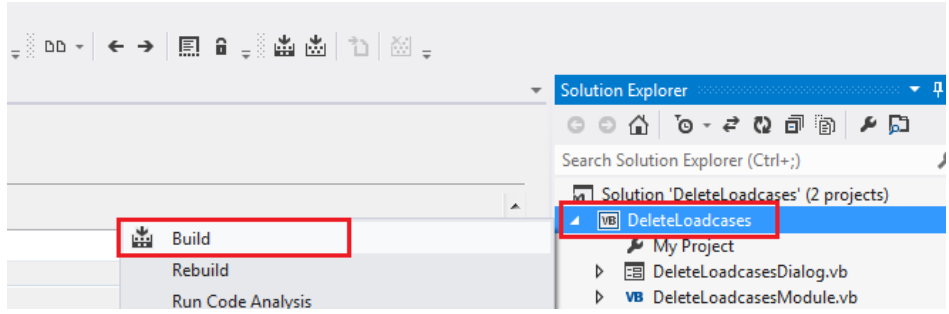
Note. If both 64Bit and 32Bit applications are to be supported, then it is better to build the modules into the "per machine folders" typically - **C:\ProgramData\Lusas<version_number>\x64\Modules** or **C:\ProgramData\Lusas<version_number>\x86\Modules**

- On the Debug page, browse and change the external program path to **C:\<LUSAS Installation Folder>\Programs\lusas_m.exe**



Build the project

- In the Solution Explorer right-click on the project icon **DeleteLoadcases** entry and select **Build**.



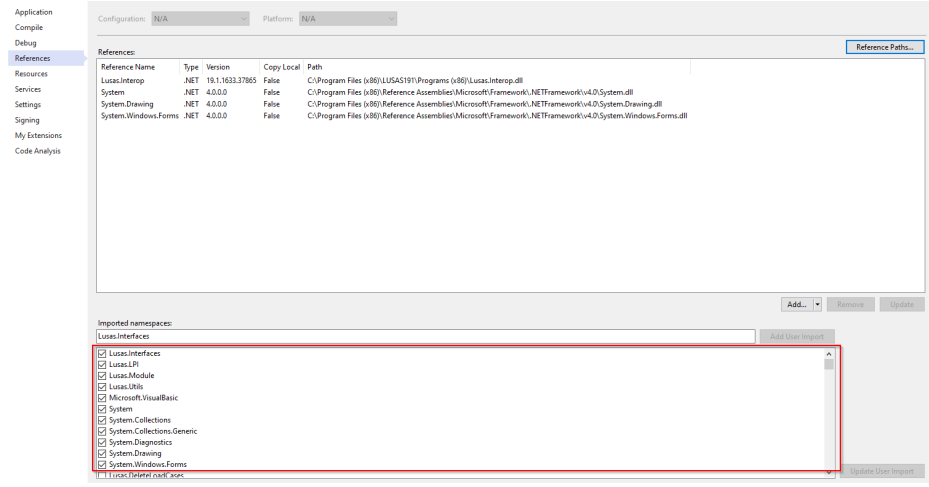
Note. Details of the build will appear in the output window.

```
1>  DeleteLoadCases -> C:\Users\matth\source\repos\DeleteLoadCases\DeleteLoadCases\DeleteLoadCases.dll
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

If issues are encountered...

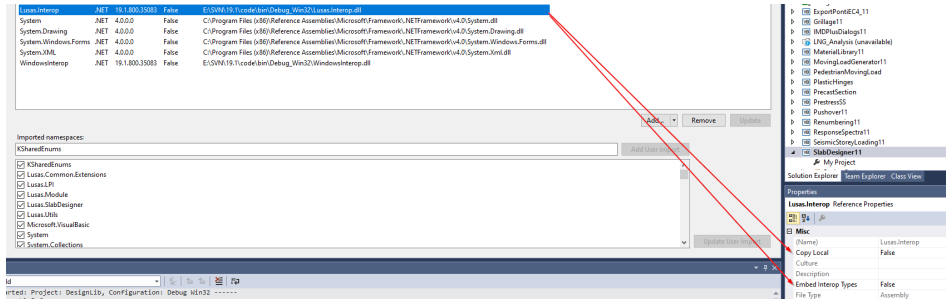
With Visual Studio 2019 Professional an issue maybe encountered. If this is the case, proceed as follows:

1. Make sure the following namespaces are checked in the References tab (specifically the LUSAS ones):



This can be found by right-clicking on the project name and selecting **Properties** (as explained in the figure at the bottom of page 9.)

Specifically, for the LUSAS.Interop set “Copy Local” and “Embed Interop Types” to false as shown below (if not already done so above):



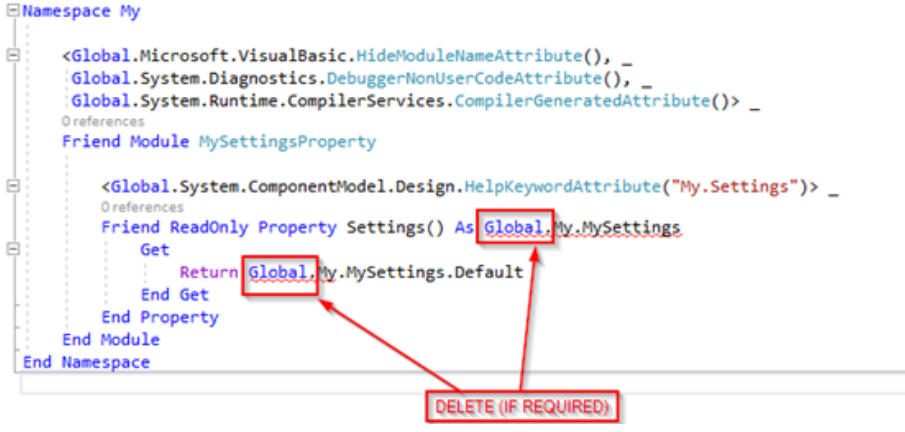
2. Depending on the version of Visual Studio in use you may get the following build errors:

   Type 'Global.My.MySettings' is not defined

   or

   My is not a member of 'Global'

These can be solved by double clicking on the error message (which will take you to the line of code where the error occurs) and deleting the word "Global."



Specifically, for LUSAS version 19.1, in order to add a reference to LUSAS Modeller ActiveX Script Language 19.1

3. Open a **cmd** prompt as **Admin.**

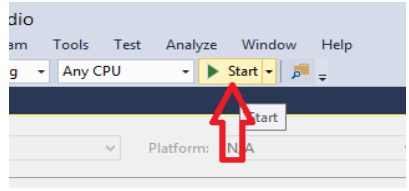
To do this type "**cmd**" into the start menu search box when the cmd icon appears right click on it and select "**run as admin**".

4. Change directory (**cd**) to **C:\Program Files (x86)\LUSAS191\Programs (x86)**
5. Type **lusas_m.exe /RegServer**

If this does not overcome this issue, please contact LUSAS technical support.

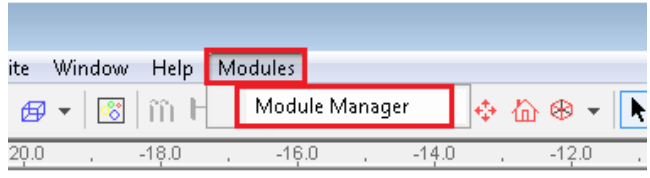
Run the project

- Press the **F5** Key in Visual Studio to run (or click on start button)

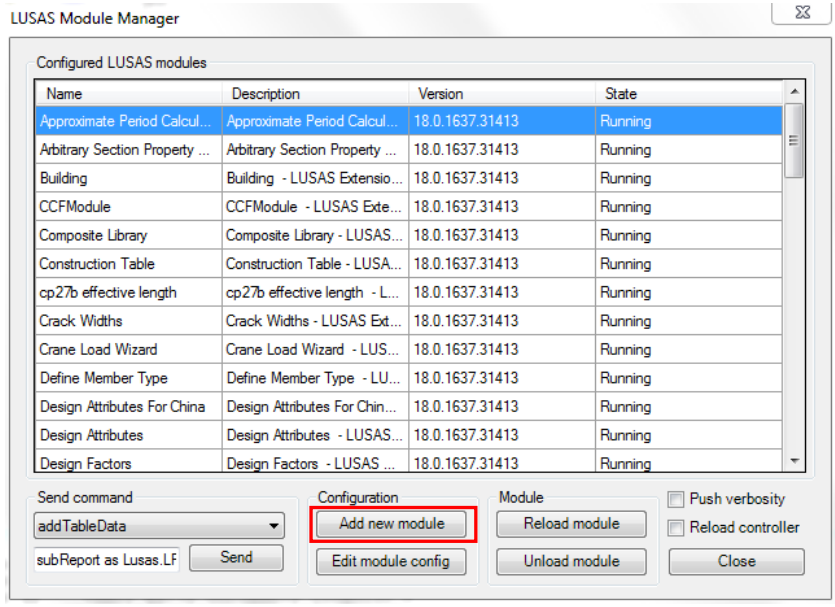


LUSAS Modeller should start.

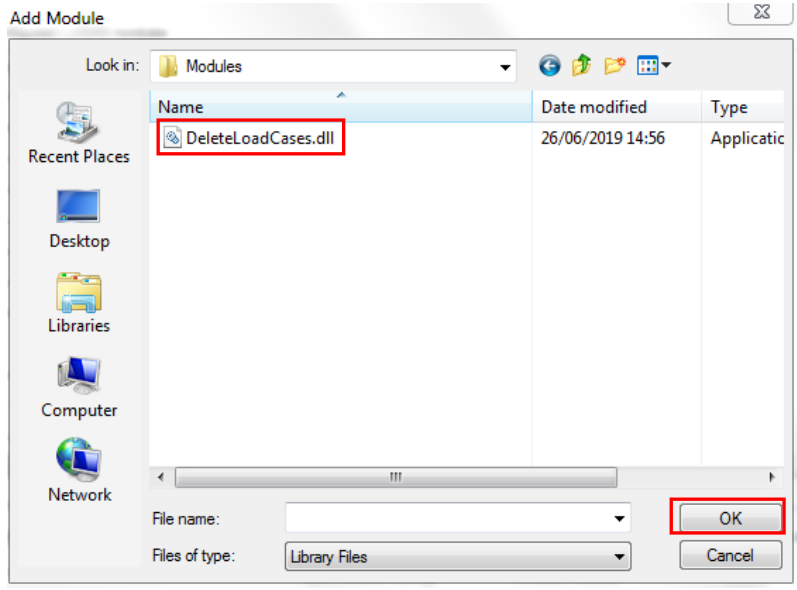
- Open a LUSAS model or create a new one.
- Open the **Module Manager**



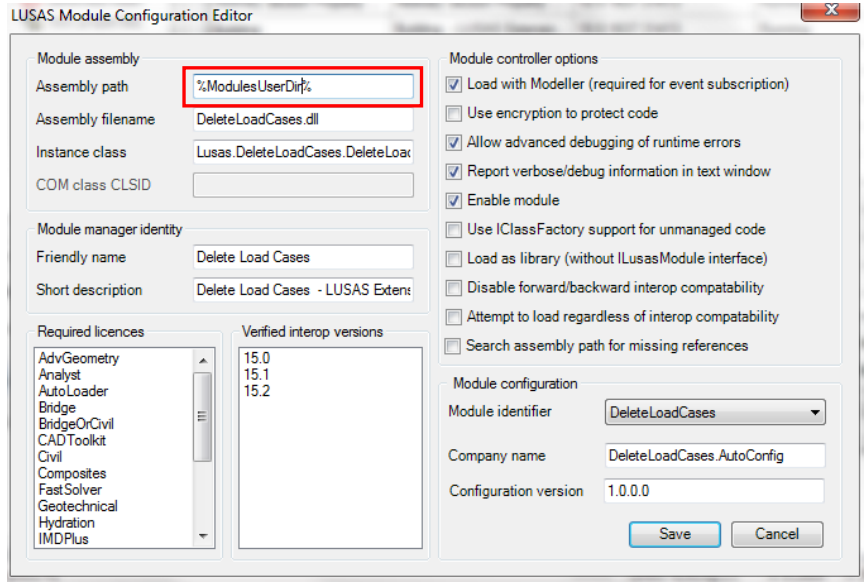
- Click **Add new module**



- Browse on the Add Module dialog for **Documents\LUSAS<version_number>\220\Modules\DeleteLoadcases.dll** and click **OK**

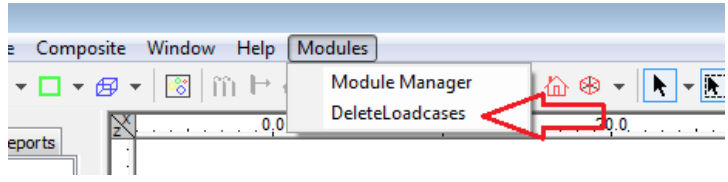


The LUSAS Module Configuration Editor dialog will appear. Change the assembly path to match the build output path in the user directory by typing in **%ModulesUserDir%.** Alternatively, the full folder path can be entered.

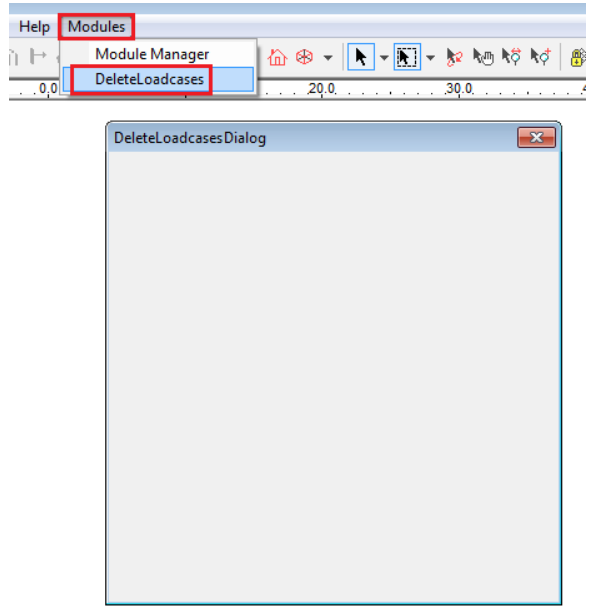


- Press the **Save** button and then **Close**

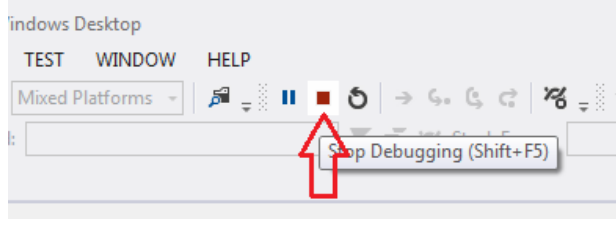
From now on, you will see the DeleteLoadcases item in the Modules menu.



- Choose the **DeleteLoadcases** menu item. A blank window is displayed, because you have not yet added any controls or written any code.

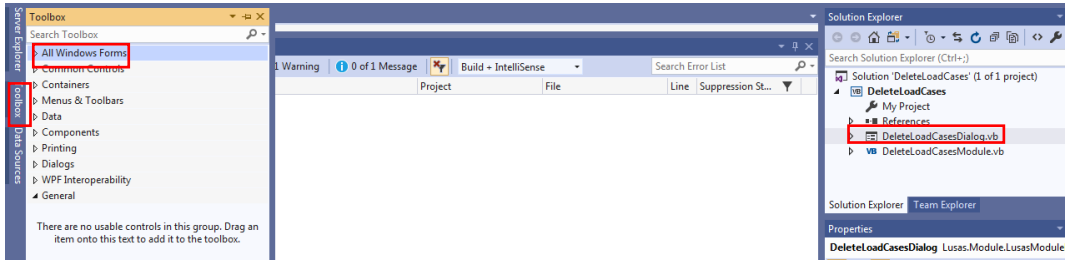


- Click on **Stop Debugging** in Visual Studio to close LUSAS Modeller and amend the project.

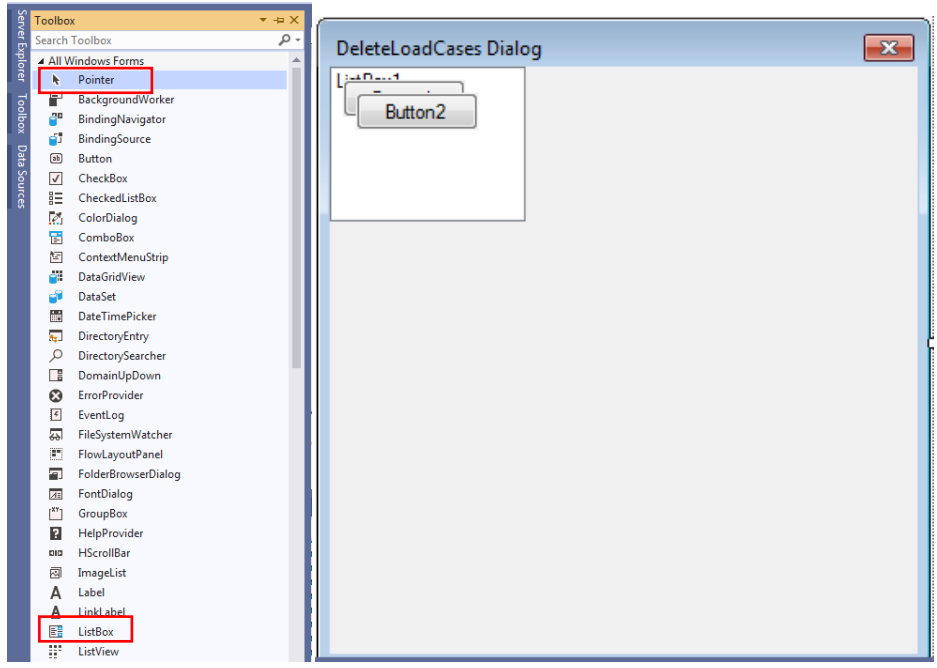


Adding dialog controls

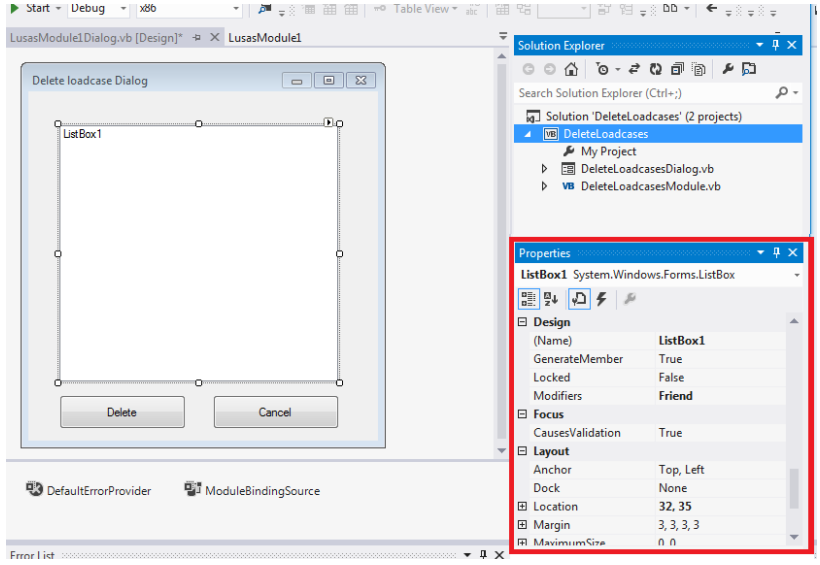
- In Visual Studio double-click on **DeleteLoadcasesDialog.vb** to open a blank dialog.



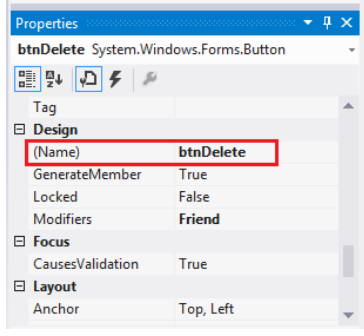
- Click on **Toolbox**, **All Windows Forms** and add a **ListBox** and two Buttons by double clicking them

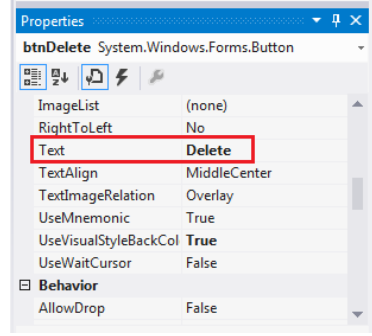


- Select each of the controls to modify its properties (Name, location, size...)

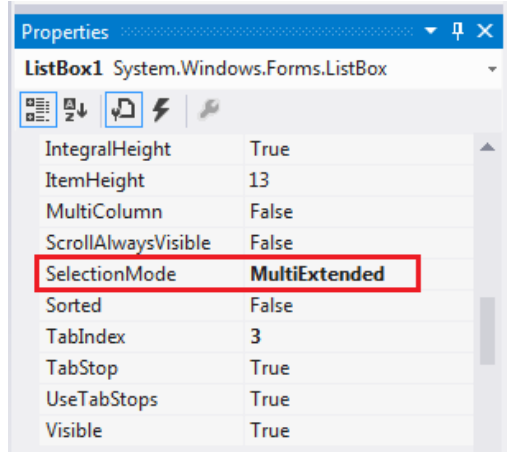


- Click on **Button1** in the dialog and define its Name to be **btnDelete** with Text: **Delete**





- For Button 2 define its Name to be **btnCancel** with Text: **Cancel**
- For the ListBox define the Selection Mode to be **MultiExtended** (This will allow selection of multiple items in the listbox).



Defining a ListBox

All the loadsets (Loadcases, Combinations and Envelopes) are to be listed in the ListBox

- Double-click on the form (i.e the dialog itself) to create an event handler for the form load event

```
Private Sub DeleteLoadCasesDialog_Load(sender As Object, e As EventArgs) Handles MyBase.Load

End Sub
```

Modify the code as shown below:

Tip. Open the PDF file for this guide and copy and paste the code. Take care to ensure that any unwanted line breaks are removed.

```
Private Sub Delete_LoadcasesDialog_Load(sender As Object, e As
EventArgs) Handles MyBase.Load

Call PopulateListBox()

End Sub

Private Sub PopulateListBox()

        'Delete previous items from listbox

        ListBox1.Items.Clear()

        'Add loadsets to the listbox
```

```
        Dim LoadsetsArray =
moduleObject.Modeller.db.getLoadsets("All", "All")

        For i = 0 To UBound(LoadsetsArray)

            ListBox1.Items.Add(LoadsetsArray(i).getName())

        Next

    End Sub
```

Defining a Delete Button

The selected loadsets of the list box need to be deleted when pressing this button, so:

- Double-click on the button **Delete** to create an event handler for this button's click event.
- Modify it by typing the following:

Tip. Rather than type the lines of VB required, open the PDF file for this guide and for the remainder of this section of the Guide copy and paste the code where needed. Take care to ensure that any unwanted line breaks are removed.

```
Private Sub btnDelete_Click(sender As Object, e As EventArgs)  Handles
btnDelete.Click


       'Delete loadsets that are selected in the listbox

        For Each Item In ListBox1.SelectedItems

           Call moduleObject.Modeller.database.deleteLoadset(Item)

        Next


        Call PopulateListBox()

    End Sub
```

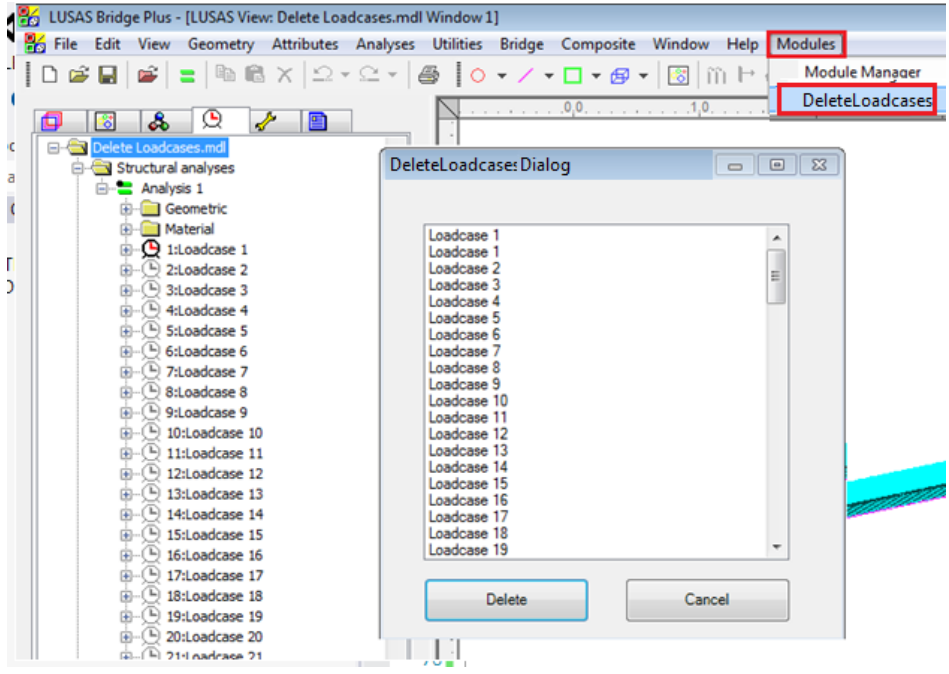
Defining a Cancel Button

To cause the dialog to close:

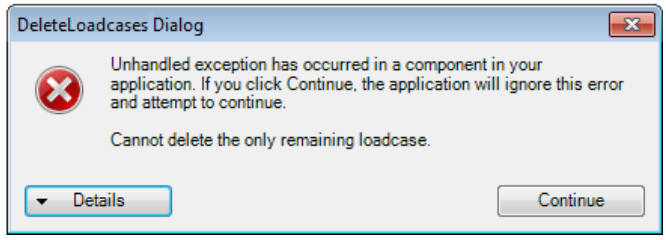
- Double-click on the button **Cancel** to create an event handler for this button click event and modify it to:

```
    Private Sub btnCancel_Click(sender As Object, e As EventArgs) Handles
btnCancel.Click

        Close()

End Sub
```

- Now build and run the project by pressing the **F5** key in Visual Studio.
- In LUSAS Modeller select the menu item **Modules> DeleteLoadcases**, then select which loadcase to delete and press the **Delete** button.



You should notice that if you try to delete all loadcases you get the following error message:



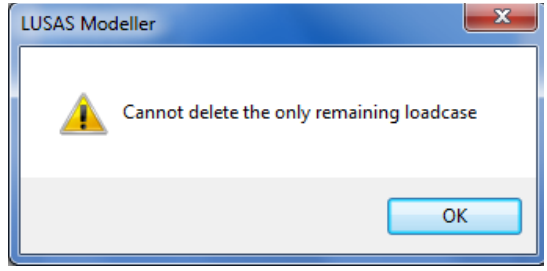
This is because Modeller has raised an exception as there must always be at least one loadcase. You can handle or catch the exception so you get a meaningful message saying “Cannot delete the only remaining loadcase”

Handling errors

The code will be modified to handle this situation. Add the following code at the beginning of the btnDelete_Click function.

```
    Private Sub btnDelete_Click(sender As Object, e As EventArgs)
Handles btnDelete.Click

        Dim loadcaseArray =
moduleObject.Modeller.database.getLoadsets("Loadcase", "All")

        Try

            'Delete loadsets that are selected in the listbox

            For Each Item In ListBox1.SelectedItems

                Call moduleObject.Modeller.database.deleteLoadset(Item)

            Next

        Catch ex As Exception

            Call moduleObject.Modeller.AfxMsgBox(ex.Message())

        End Try

        Call PopulateListBox()

    End Sub
```

Now, if you try to delete all loadcases you get the following ‘cleaner’ message:



General considerations

Basic dialog design

- Use the ToolBox to create controls
- Double-click on the control in the toolbox to create the control at standard size
- Drag and drop to desired position using grid lines to line up with other controls
- Name the controls in Properties using standard naming convention (See basic dialog controls prefixes below e.g. txt, btn, opt, chk)
- Set FormBorderStyle = FixedDialog
- Set Localizable = True

Basic dialog controls

- TextBox (txt) – string of input or output
- Button (btn) – activate an event
- ComboBox (cbo) – choice of preset input
- CheckBox (chk) – true or false
- RadioButton (opt) – choice of a number of options
- NumericUpDown (spn) – Integer or decimal within specified range
- PictureBox (img) – display images on dialog
- GroupBox (grp) – groups radio buttons
- Label (lbl) – add text to dialog
- Panel (pnl)– invisible group, enable/disable multiple controls

Code design considerations

- Only code relating to the dialog should be contained in the dialog class <projectname>Dialog.vb – e.g. Events, data check functions, change label text etc. (Access to the module code is achieved by using moduleObject.<function>)
- Place all worker code in module class <projectnameModule.vb> (access to Modeller LPI functions is achieved using Modeller.<LPIfunction>")
- All variables must be allocated a Type. Modeller has a number of predefined data types e.g. IFPoint, IFLine. All Modeller data types start with IF and a full list is automatically displayed in Visual Studio when the type is being declared.
- Use access modifiers to restrict the scope of functions as much as possible. Only make the function public if it is required outside the module.
- Private only available to routines in this module
- Protected only available within class and derived classes
- Friend only available within the assembly (Dialog to module)
- Public visible globally and outside of the assembly
- It is a good idea to comment all functions, classes, modules etc using the standard XML comment blocks. Note: Typing three quotes (''') on the line immediately above the function name will automatically present the standard html template with the parameters included.

Multiple Dialogs in a single module

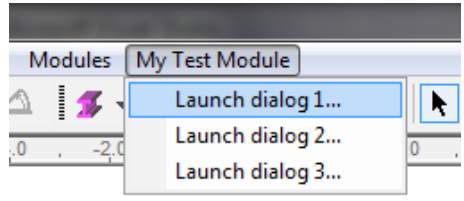
By default the LUSAS module template is set up to handle a single dialog. It is possible for a module to provide multiple dialogs and provide multiple menu entries. It is good practice to keep all related functionality in a single module.

When a module creates a new menu item, Modeller will return a unique id for that menu item, these id's should be stored in "member" variables within the module. Modules can create new menu items in the onRefreshMainMenu function as shown below:

```
Private m_dialog1_ID As Integer

Private m_dialog2_ID As Integer
```

```
Private m_dialog3_ID As Integer


''' <summary>

''' Called when Modeller is redrawing the Main Menu.

''' </summary>

''' <remarks>

''' Allows custom modules to append and maintain their own menus.

''' </remarks>

Protected Overrides Sub onRefreshMainMenu()

    Dim myModuleMenu As IFMenu = rootMenu.appendMenu("My Test Module")

    m_dialog1_ID = myModuleMenu.appendItem("Launch dialog 1...",
"textwin.writeLine(""Test Dialog 1"")")

    m_dialog2_ID = myModuleMenu.appendItem("Launch dialog 2...",
"textwin.writeLine(""Test Dialog 2"")")

    m_dialog3_ID = myModuleMenu.appendItem("Launch dialog 3...",
"textwin.writeLine(""Test Dialog 3"")")

End Sub
```

When the user clicks these menu items all modules will be called with the id of the menu item. It is the responsibility of the module to listen for any menu ids it creates and respond accordingly.



Notes:

- The modules are called via the onMenuClick function
- The module should respond when it is called with the correct menu ID.

- For each menu id the moduleDialog should be set to a new instance of the correct dialog before the runModule function is called. This way the correct dialog will be shown, as below.

```
''' <summary>

''' Called when the user clicks on a menu entry.


''' </summary>

''' <param name="menuID">ID of the menu that has been clicked.</param>

''' <param name="edittingObj">Object that is being edited (nothing when
creating a new object).</param>

''' <param name="clientData">Data that was provided to Modeller when
defining edittingObj.</param>

''' <returns>true if the click event was handled by this
Module.</returns>

''' <remarks>

''' LUSAS expects the a Module handling the event to execute itself
(typically using runModule()).

''' </remarks>

Function onMenuClick(ByVal menuID As Integer, ByVal edittingObj As
Object, Optional ByVal clientData As Object = Nothing) As Boolean



    If (m_dialog1_ID = menuID) Then

        moduleDialog = New myDialog1(Me)

        runModule()

        Return True

    End If

    If (m_dialog2_ID = menuID) Then

        moduleDialog = New myDialog2(Me)

        runModule()
```

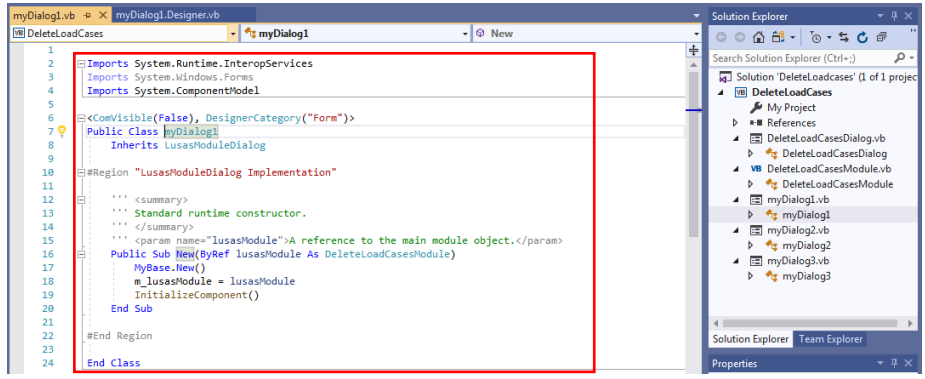
```
        Return True

    End If

    If (m_dialog3_ID = menuID) Then

        moduleDialog = New myDialog3(Me)

        runModule()

        Return True

    End If

    Return False

End Function
```

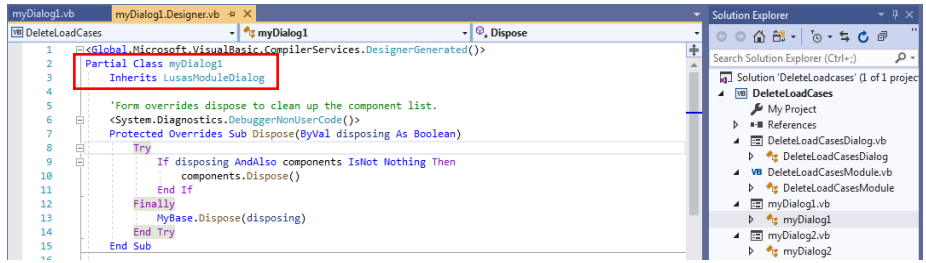
- Menu items maybe enabled or disabled in the onMenuUpdate event

```
''' <summary>

''' Called when a menu entry needs to be drawn.

''' Allows the Module to specify whether the menu item should be
disabled or checked.

''' </summary>

''' <param name="menuID">ID of the menu that has been clicked.</param>

''' <param name="edittingObj">Object that is being edited (nothing when
creating a new object).</param>

''' <param name="enable">Set to true to enable the menu item.</param>

''' <param name="checked">

''' Set to 0 to show an 'off' tickbox next to the menu.

''' Set to 1 to show an 'on' tick mark by the side of the menu.

''' Set to 2 to show an indeterminate check.

''' Set to 3 to show no tick at all.

''' </param>

''' <param name="clientData">Data that was provided to Modeller when
defining edittingObj.</param>
```

```
''' <returns>true if the update event was handled by this
Module.</returns>

''' <remarks>

''' Only when a Module handles an menu update event are the changed
values of enable/checked respected.

''' </remarks>

Function onMenuUpdate(ByVal menuID As Integer, ByVal edittingObj As
Object, ByRef enable As Boolean, ByRef checked As Integer, Optional
ByVal clientData As Object = Nothing) As Boolean



    If (m_dialog1_ID = menuID) Then

        enable = (Modeller.db.countSurfaces() > 0)

        Return True

    End If

    If (m_dialog2_ID = menuID) Then

        enable = True

        Return True

    End If

    Return False

End Function
```

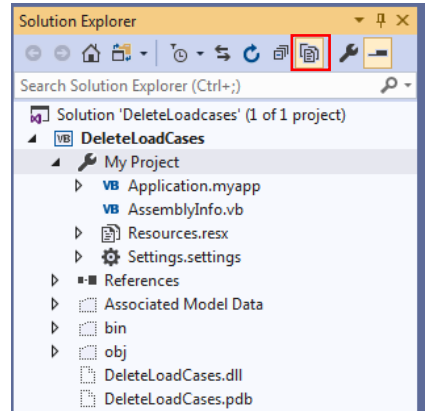
Note. All dialogs must inherit from LusasModuleDialog. When adding a new dialog you should change the code in the dialog designer to inherit from LusasModuleDialog rather than System.Windows.Forms.Form



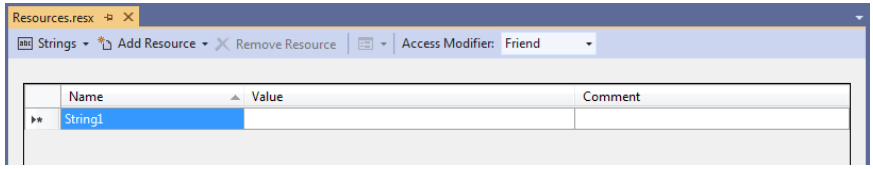


Translation considerations

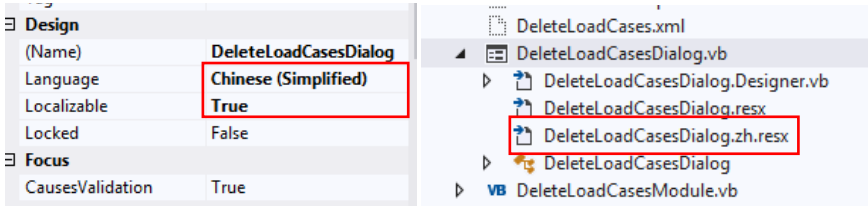
- The default language should always be English. All strings should be defined in the Resources.resx file. To access the Resources.resx file pick the **Show All Files** button in the Solution Explorer.



Double clicking on the **Resources.resx** will display a window to name and define the strings.



Note. If languages other than English are to be supported the dialog property Localizable property should be set to be **True** and the Language should be changed to the translation language, as for example for Chinese (Simplified) This will automatically create a new resource file for the dialog where the translated string should be defined and allow the labels to be translated and the size and position of the controls to be customised. Changing the Language back to Default will display the English labels with the controls set to in their original size and position.



By using this approach any strings which do not have a translation will be displayed in English and a Language for which translation is not supported will show English labels and strings.

VB.NET online tutorials

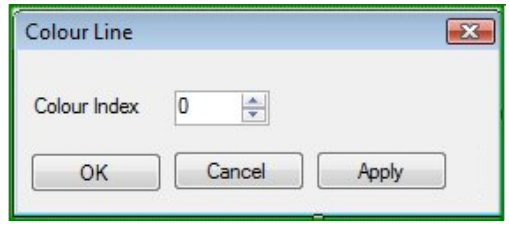
VB.NET online tutorials are widely available. Here are some examples:

English:
https://www.youtube.com/watch?v=hkcO_M9gcNw&index=1&list=PL42055376AE25291E

English:
https://www.youtube.com/watch?v=AJpTbPasJqI&list=PLS1QulWo1RIYLpgVN_CpXbkOQoYJTItzg

Chinese: https://channel9.msdn.com/Series/Visual-Basic-Fundamentals-for-Absolute-Beginners/01

VB.NET dialog exercise



The preceding dialog is required to allow a user to change the colour of all Lines in a model. The dialog should be activated from the menu item My Menu> Colour Line

Write the code to enable this to take place.

The solution is shown on the next page.

VB.NET dialog solution

1. In Dialog Class:

```
    Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnOK.Click

        Call btnApply_Click(sender, e)

        Call btnCancel_Click(sender, e)

    End Sub



    Private Sub btnApply_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnApply.Click

        moduleObject.ColourLines(spnColourIndex.Value)

    End Sub



    Private Sub btnCancel_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnCancel.Click

        Me.Close()

    End Sub
```

2. In Module Class, in existing function onRefreshMainMenu change:

```
If rootMenu.exists("Modules") Then

    modMenu = rootMenu.getSubMenu("Modules")

Else

    modMenu = rootMenu.appendMenu("Modules")

End If
```

to:

```

```

```
Dim menuName As String = "My Menu"

If rootMenu.exists(menuName) Then

    modMenu = rootMenu.getSubMenu(menuName)

Else

    modMenu = rootMenu.appendMenu(menuName)

End If
```

3. Add function

```
    ''' <summary>

    ''' Routine to colour lines

    ''' </summary>

    ''' <param name="colour">colour index</param>

    ''' <remarks></remarks>

    Public Sub ColourLines(ByVal colour)

        Dim lines As Object = Modeller.database.getObjects("Lines",
"All")

        For Each line As IFLine In lines

            line.setPen(colour)

        Next

End Sub
```

LUSAS via COM

Component Technology

The LUSAS Programmable Interface allows interfacing with other compatible Windows programs through a Component Object Model (COM) interface. This technology defines a set of rules by which two programs can communicate and allows controlling those programs as if they were part of LUSAS Modeller. LUSAS can also be used as a component of another system (running transparently if required) providing modelling capabilities, analysis solutions and results viewing and processing options for that application.

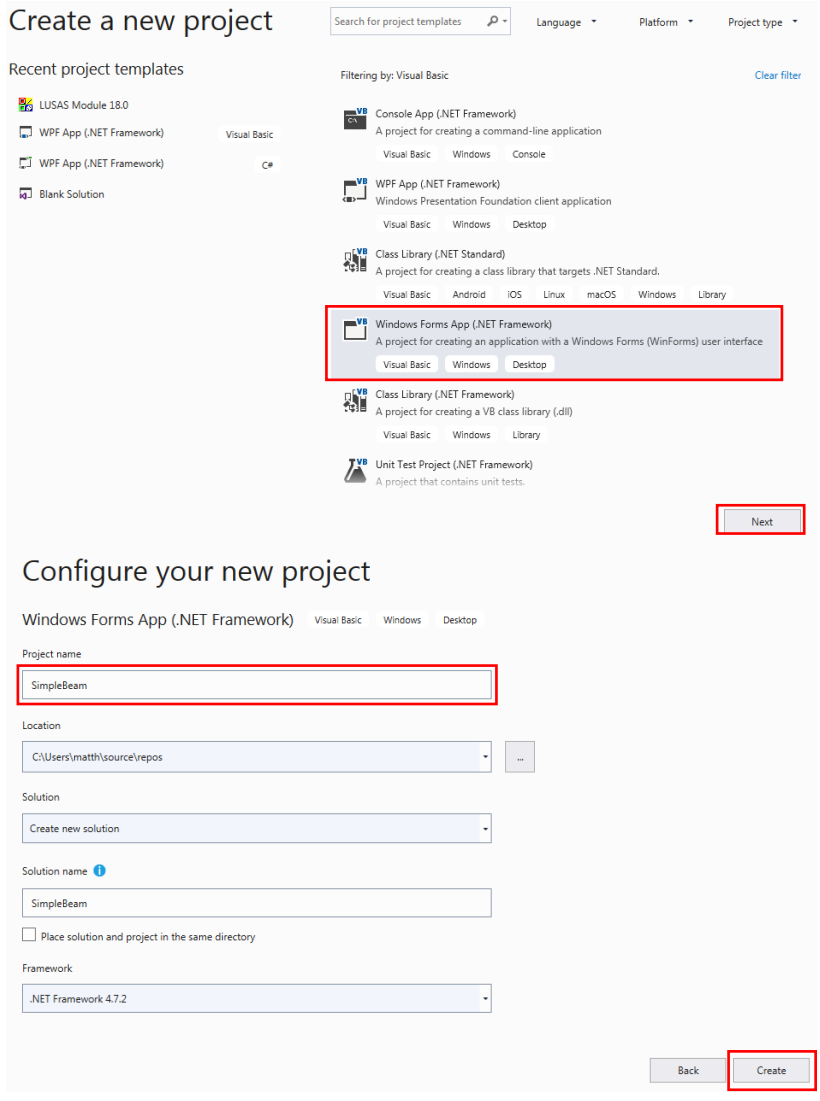
In order to drive LUSAS from a standalone application via COM (Component Object Model), LUSAS must be installed and licenced. When creating a COM instance of LUSAS a licence will be used. The licence will be in use for the lifetime of the instance and must be properly disposed of to release the licence.

VB.NET Application Example

To illustrate the process involved, a stand-alone application called SimpleBeam will be created. The application will accept two parameters, length and load. The application will use LUSAS to analyse the beam and return the results for the maximum bending moment.

Create a new project

- In Visual Studio create a new Windows Form App (.NET Framework) in Visual Basic called SimpleBeam.



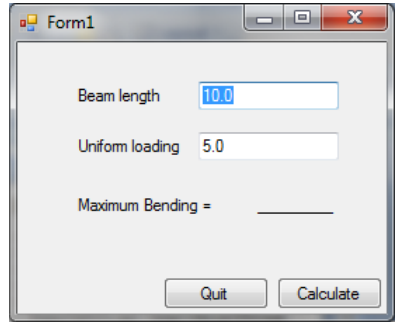
Note that LUSAS interop is based on the .NET framework version 4.7.2 shown above. This may not appear in the recent versions of Visual Studio which defaults to dotnet 8 or even dotnet 9. The naming of the dotnet framework can be confusing, later versions drop the term “framework”.

To target the required dotnet framework 4.7.2 click on the project icon “Simple Beam” in the Solution Explorer which should open an xml project file.

Here you can change the name of the target framework in the xml tags "TargetFramework" to be net472

With that done we need to carry out the following steps.

- Add a reference to LUSAS Modeller.
- In Solution Explorer, click Show All Files
- Right-click on "Dependencies" and select "Add COM Reference"
- In the COM tab select LUSAS Modeller ActiveX Script Language 22.0
- Create the following dialog using the Toolbox to add textbox controls and labels as shown in the previous example:



For simplicity all code is placed in the dialog as follows, by double clicking the form window.

```
Imports LusasM22_0

Public Class Form1

    Private m_lusas As LusasM22_0.LusasWinApp ' Reference to Lusas
Modeller

    Private Sub btnCalculate_Click(sender As System.Object, e As
System.EventArgs) Handles btnCalculate.Click
        analyseBeam()
    End Sub
```

```
    Private Sub analyseBeam()
        ' Get the params
        Dim length As Double = Double.Parse(txtLength.Text)
        Dim loading As Double = Double.Parse(txtLoading.Text)

        ' Create an instance of modeller
        m_lusas = New LusasM22_0.LusasWinApp

        ' Create a new model
        m_lusas.newDatabase()
        ' Set the vertical axis
        m_lusas.db.setLogicalUpAxis("Z")
        ' Set the unit system
        m_lusas.db.setModelUnits("kN,m,t,s,C")


        ' *** Create a line ***
        ' Get the geometry data object
        Dim geomData As IFGeometryData = m_lusas.geometryData()
        ' Set the defaults
        geomData.setAllDefaults()
        ' Set the coordinates of the first point
        geomData.addCoords(0, 0, 0)
        ' Set the coordinates of the second point
        geomData.addCoords(length, 0, 0)
        ' Create the line object
        Dim linesDBop As IFObjectSet = m_lusas.db.createLine(geomData)
        ' Get the lines
        Dim lines() As Object = linesDBop.getObjects("Lines", "All")
        ' Get a reference to the created line
        Dim beamLine As IFLine = lines(0)


        ' *** Create a mesh attribute ***
        Dim meshAttr As IFMeshLine = m_lusas.db.createMeshLine("Beam
Mesh")
        ' Set the element type and number of elements (1m elements here)
        meshAttr.setNumber("BMS3", length)


        ' *** Create a geometric attribute ***
        Dim geomAttr As IFGeometricLine =
m_lusas.db.createGeometricLine("Beam Geometry")
        ' Set the element type
        geomAttr.setValue("elementType", "3D Thick Beam")
        ' Set the beam properties
        geomAttr.setBeam(0.0125, 0.0004573, 0.00002347, 0.0, 0.00000121,
0.00532608, 0.00755776, 0.0, 0.0, 0)

        ' *** Create a material attribute ***
        Dim materialAttr As IFMaterialIsotropic =
m_lusas.db.createIsotropicMaterial("Steel", 209000000.0, 0.3, 7.8)

        ' *** Create a support attribute (fixed) ***
        Dim fixedSupport As IFSupportStructural =
m_lusas.db.createSupportStructural("Fixed")
```

```
        ' set the freedoms
        fixedSupport.setStructural("R", "R", "R", "F", "F", "F", "F",
"F", "F")

        ' *** Create a support attribute (pinned) ***
        Dim pinnedSupport As IFSupportStructural =
m_lusas.db.createSupportStructural("Pinned")
        ' set the freedoms
        pinnedSupport.setStructural("F", "R", "R", "F", "F", "F", "F",
"F", "F")
        ' *** Create a load attribute ***
        Dim loadAttr As IFLoadingGlobalDistributed =
m_lusas.db.createLoadingGlobalDistributed("UDL")
        ' Set the parameters
        loadAttr.setGlobalDistributed("Length", 0.0, 0.0, -loading, 0.0,
0.0, 0.0, 0.0, 0.0)
        ' *** Assign the attributes to the geometry ***
        ' get the assignment object
        Dim assignment As IFAssignment = m_lusas.assignment()
        ' set the defaults
        assignment.setAllDefaults()
        ' Assign the mesh
        meshAttr.assignTo(beamLine, assignment)
        ' Assign the geometry
        geomAttr.assignTo(beamLine, assignment)
        ' Assign the material
        materialAttr.assignTo(beamLine, assignment)
        ' Assign the loading
        loadAttr.assignTo(beamLine, assignment)

        ' Assign the supports to the points of the line
        ' get the points - Lower Order Features
        Dim pointsArray() As Object = beamLine.getLOFs()

        ' Assign the fixed support to the first point
        fixedSupport.assignTo(pointsArray(0), assignment)
        ' Assign the pinned support to the last point
        pinnedSupport.assignTo(pointsArray(1), assignment)

        ' Set the mesh
        m_lusas.db.updateMesh()

        ' The model is ready to be solved - get the temporary file path
        Dim tempFilePath As String = System.IO.Path.GetTempPath()
        ' Save the model before solving
        m_lusas.db.saveAs(tempFilePath & "beam.mdl")
        ' Solve
        Dim retCode = m_lusas.db.getAnalysis("Analysis 1").solve(True)

        If retCode <> 0 Then
            m_lusas.AfxMsgBox("Failed to solve")
            Return
        End If
        ' Otherwise open the results and find the max bending moments
        m_lusas.db.openAllResults(True, True)

        ' *** Successful analysis - Process the results to determine the
```

```
max bending ***
        Dim maxMom As Double
        Dim nodeNum As Integer
        ' Get the results at each node to determine the max
        For Each element As IFElement In beamLine.getElements()
            For Each node As IFNode In element.getNodes()
                ' Extract the nodal result for the required Entity and
Component
                Dim my As Double = node.getResults("Force/Moment - Thick
3D Beam", "My")
                ' Save the minimum (sagging) moment
                If my < maxMom Then
                    maxMom = my
                    nodeNum = node.getID()
                End If
            Next
        Next


        ' Get the units of the current model for display
        Dim forceUnit As String =
m_lusas.db.getModelUnits().getForceShortName()
        Dim lengthUnit As String =
m_lusas.db.getModelUnits().getLengthShortName()

        ' Set the dialog label
        lblMaxMom.Text = m_lusas.convertToString(maxMom) & forceUnit &
lengthUnit

        ' Quit the application and free the licence
        m_lusas.quit()
    End Sub


    Private Sub btnClose_Click(sender As System.Object, e As
System.EventArgs) Handles btnQuit.Click
        For Each p As Process In
System.Diagnostics.Process.GetProcessesByName("Lusas_m")
            Try
                p.Kill()
                p.WaitForExit()
            Catch ex As Exception

            End Try
        Next
        Me.Close()
    End Sub
End Class
```

Interfacing to LUSAS using C++

Generally, LUSAS recommends that you use VB or any other language that natively supports COM interfaces. C++ does not natively support COM interfaces, thus COM programming in C++ is much more complex, and results in code which is more likely to contain bugs and is harder to read. However, it is possible for experienced C++ programmers to interface to Modeller. A simple example follows:.

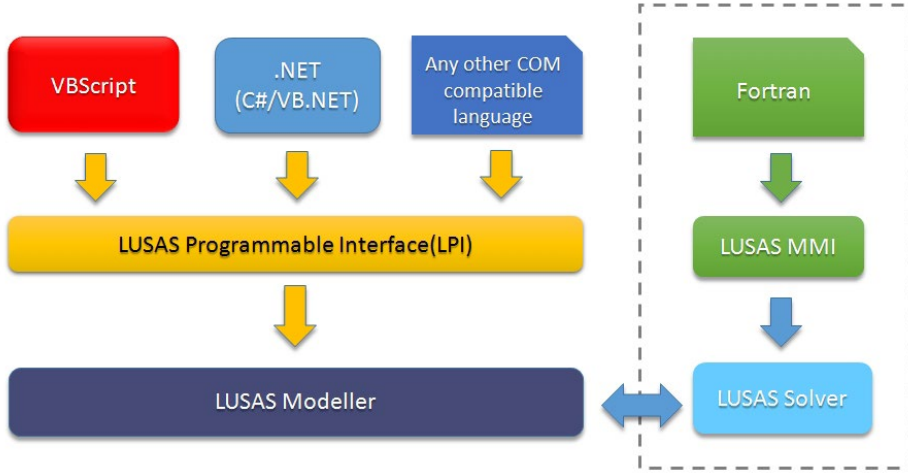
```
#import "C:\LUSAS152\programs\Lusas_m.exe"

// create a modeller

    pModeller = IFModellerPtr("LUSAS.Modeller.18.0");

// create and return a database

    IFDatabasePtr db = pModeller->newDatabase();

// create and return a line

    IFLinePtr l = db->createLineByCoordinates(0, 0, 0, 5, 5, 5);

// calculate line length

    double len = l->getLineLength();
```

Note. The LPI functions often return a base class pointer which often needs to be downcast to the desired type (e.g. attribute -> material). VB will do this for you, but C++ will not. Therefore you must explicitly cast, and catch any exceptions that may result

Note. LPI functions often have VARIANT inputs and outputs. VB will handle conversion between simple data types (integers, strings, objects) and VARIANTs, but C++ will not. Therefore you must be familiar with the use of the VARIANT type. If in doubt, consult Microsoft documentation.

LUSAS Material Model Interface

In addition to the accessing and customising LUSAS Modeller via the LUSAS Programmable Interface, user-defined material models (written in Fortran) can be compiled and built into a customised LUSAS Solver executable by using the LUSAS Material Model Interface (LUSAS MMI).



The use of LUSAS MMI is beyond the scope of this manual. Please contact LUSAS Technical Support for more information.

LUSAS

LUSAS, Forge House, 66 High Street, Kingston upon Thames, Surrey, KT1 1HN, UK
Tel: +44 (0)20 8541 1999 | Fax: +44 (0)20 8549 9399 | info@lusas.com | www.lusas.com