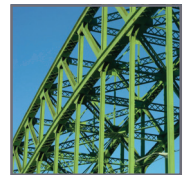


LUSAS



LUSAS Programmable Interface (LPI)

Developer Guide

LUSAS Programmable Interface (LPI) Developer Guide

LUSAS Version 16.0 : 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

Copyright ©1982-2017 LUSAS

All Rights Reserved.

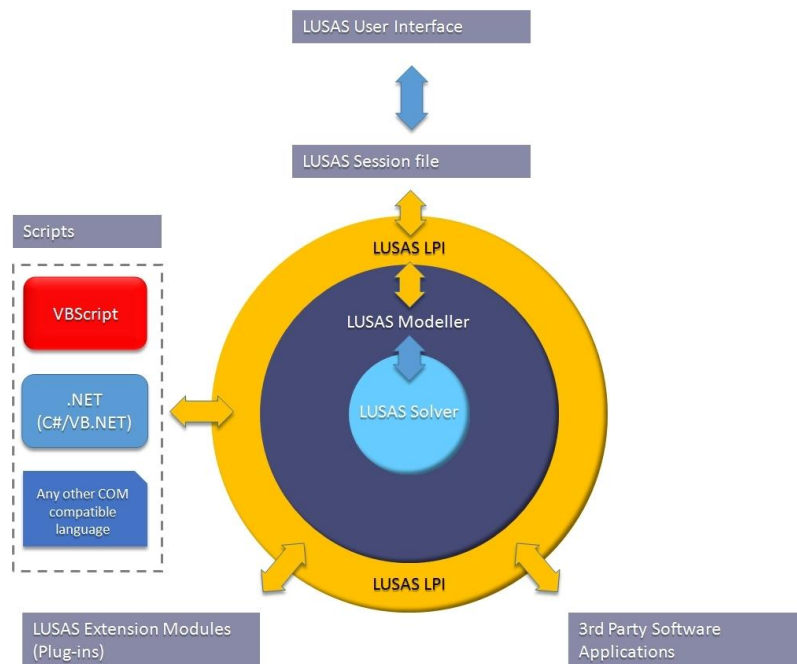
Table of Contents

Introduction	1
Introduction.....	1
Topics covered in this guide	2
LUSAS Programmable Interface (LPI) Customisation and Automation Guide.....	2
Creating dialogs using VB.NET	3
Choosing a development environment	3
Downloading and installing Visual Studio Community	4
Creating a LUSAS dialog.....	4
Module Manager	6
Creating a new module	7
Running Visual Studio	7
Build the project	12
Run the project.....	12
Adding dialog controls.....	16
Defining a ListBox.....	18
Defining a Delete Button	19
Defining a Cancel Button	20
Handling errors	21
General considerations.....	23
Basic dialog design	23
Basic dialog controls	23
Code design considerations.....	24
Multiple Dialogs in a single module	24
Translation considerations.....	28
VB.NET online tutorials	29
VB.NET dialog exercise.....	30
VB.NET dialog solution.....	31
LUSAS via COM	33
Component Technology	33
Application Example	33
Create a new project.....	33
Interfacing to LUSAS using C++.....	43
LUSAS Material Model Interface	45

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. Both Visual Studio Express and Visual Studio Community are free to use and could be used but the **Visual Studio Community** edition, which is a fully featured and extensible Integrated Development Environment (IDE) is the application recommended by LUSAS and the application that is documented in this guide. This is used in preference to the more limited Visual

Studio Express version, which is a closed IDE that cannot use any Visual Studio extensions.

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.

Downloading and installing Visual Studio Community

Visual Studio Community is free to use and can be downloaded from:

<https://www.visualstudio.com/downloads>



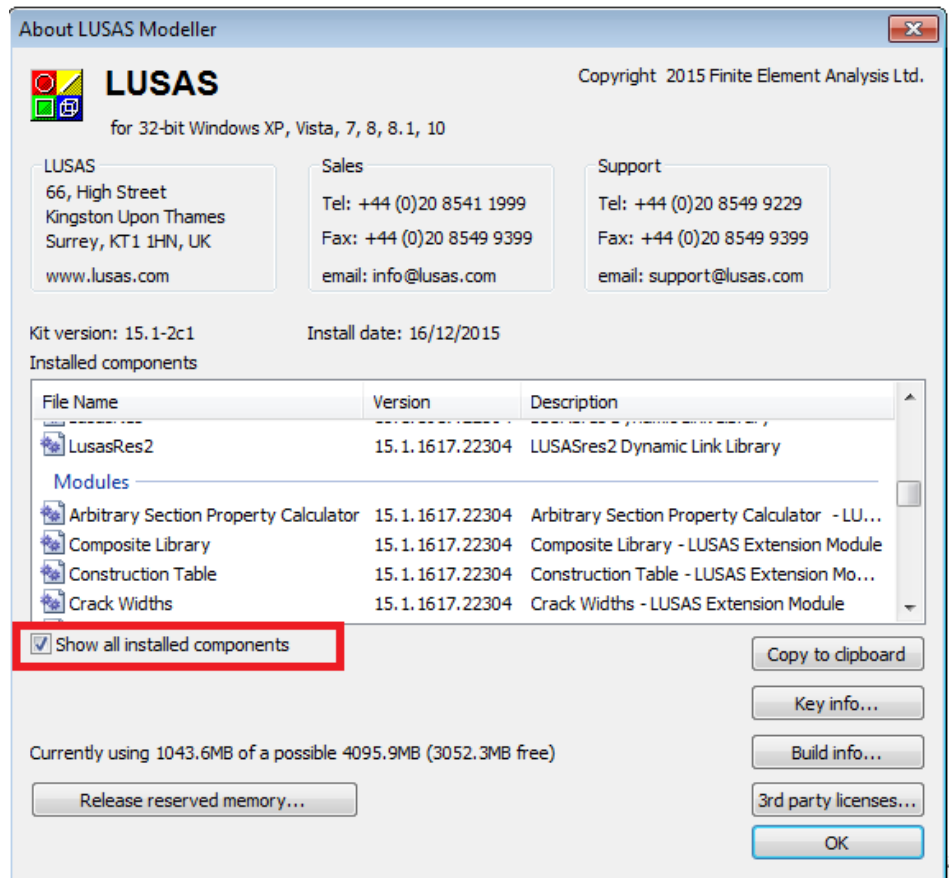
Note. Problems can be experienced when installing Visual Studio Community from a downloaded exe file (relating to the installation being unable to locate the package source as a result of you attempting to install the software when you are part of a company network). To overcome this issue, download the .iso file instead and burn it onto a DVD. Then install it from the DVD (or alternatively virtually mount the iso image).

Visual Studio Community can take a long time to install.

Creating a LUSAS dialog

To create a LUSAS dialog, you will need to add a new **Module** to LUSAS. Modules are like plugins.

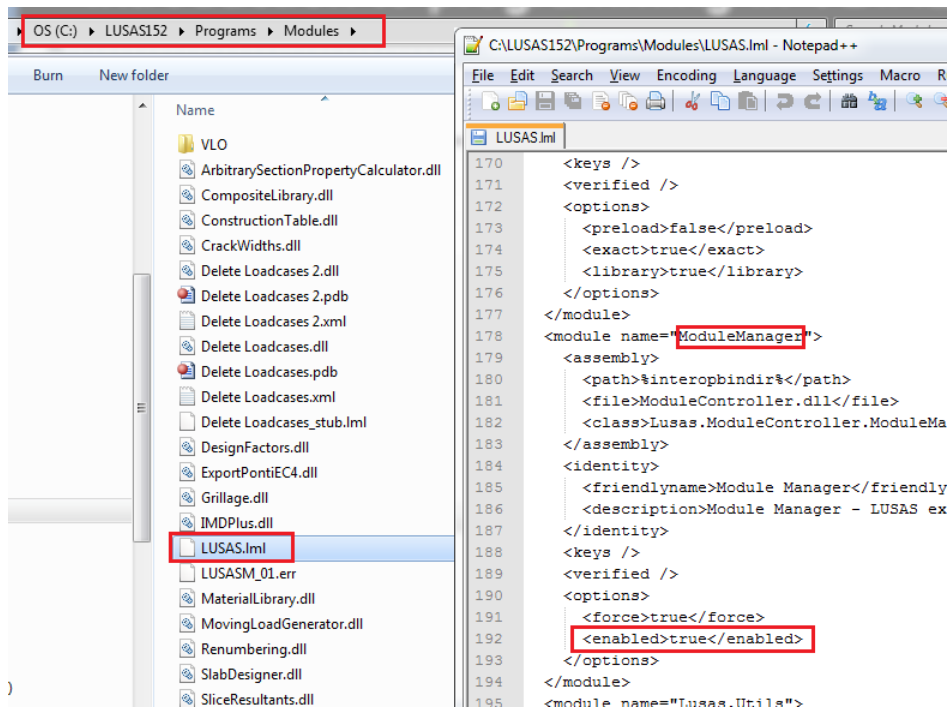
1. Run LUSAS.
2. All the installed modules can be seen by selecting the menu item **Help > About LUSAS Modeller** and checking “**Show all installed components**”



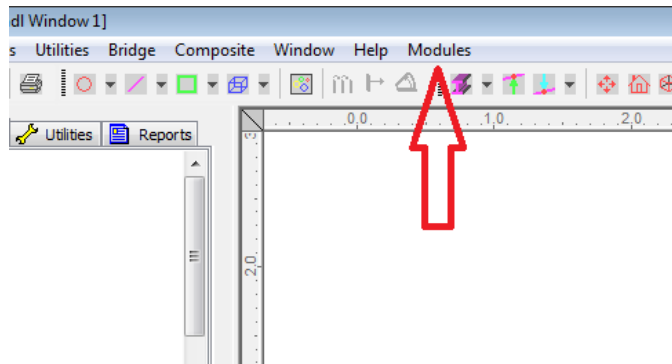
Module Manager

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:\<LUSAS Installation Folder>\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.

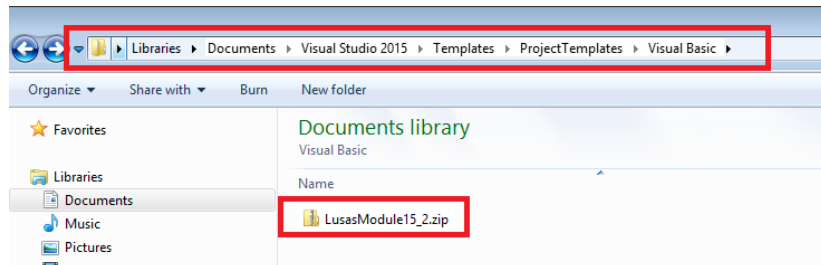


From now on, when you run LUSAS Modeller you will see the **Modules** menu:



Creating a new module

- Copy the LusasModule<version_number>.zip file from the LUSAS installation directory C:\<LUSAS Installation Folder>\Programs to the Visual Studio project template folder. For instance: Libraries\Documents\Visual Studio 2015\Templates\ProjectTemplates\Visual Basic



Running Visual Studio

- Open Visual Studio and select **File> New Project**
- In Visual Studio select the **LUSAS Module<version number>** template for your version of LUSAS).
- 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**.

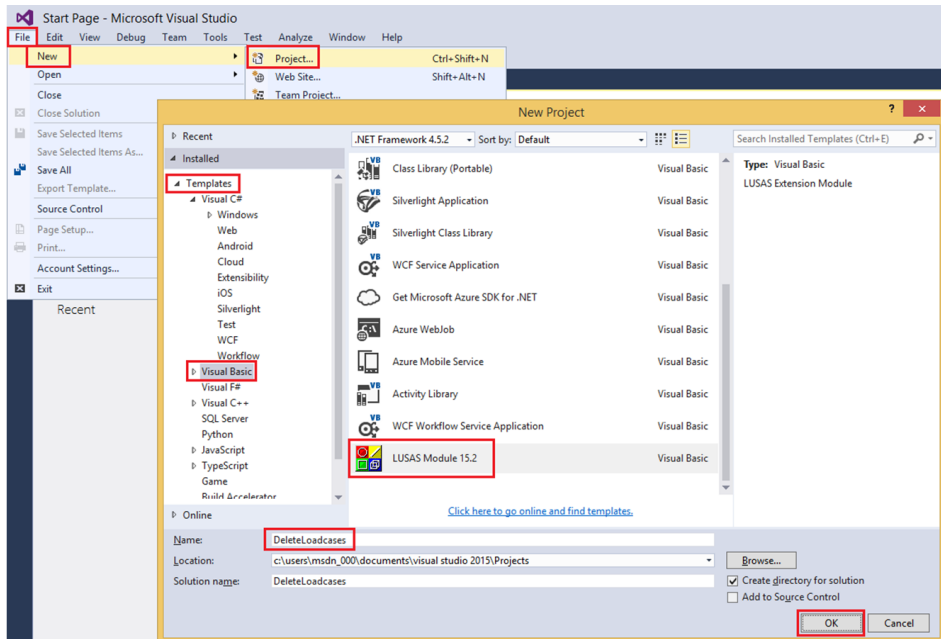


Note. It is important to use a sensible name as this propagates throughout the automatically generated code.

Creating dialogs using VB.NET

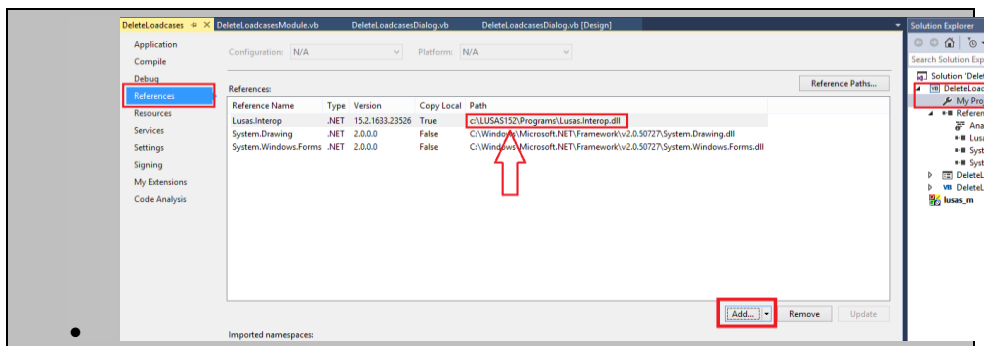


Note. Spaces can be used within Module names, but LUSAS generally avoids the use of spaces in its own scripted file names.



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.

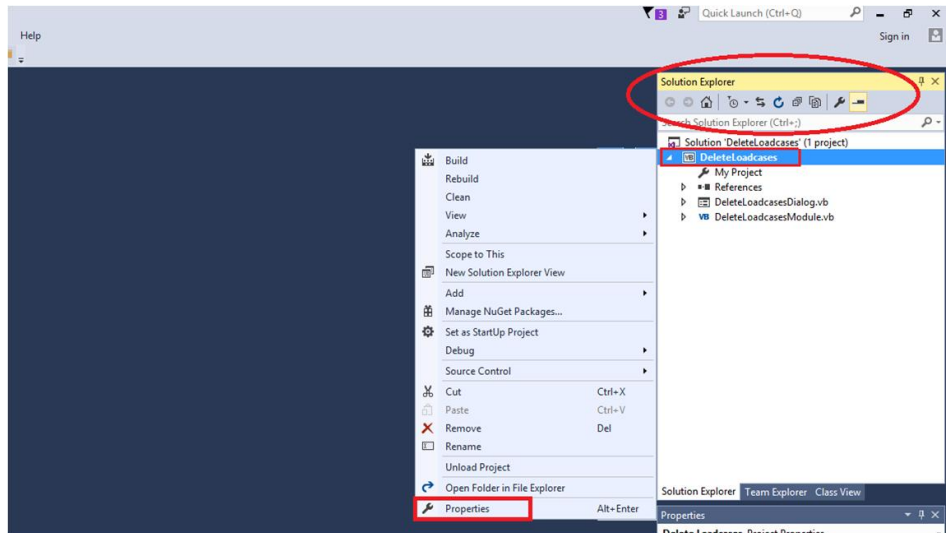
- If this is required, on the **Solution Explorer** double-click on **My Project** and visually check the path for LusasInterop.dll



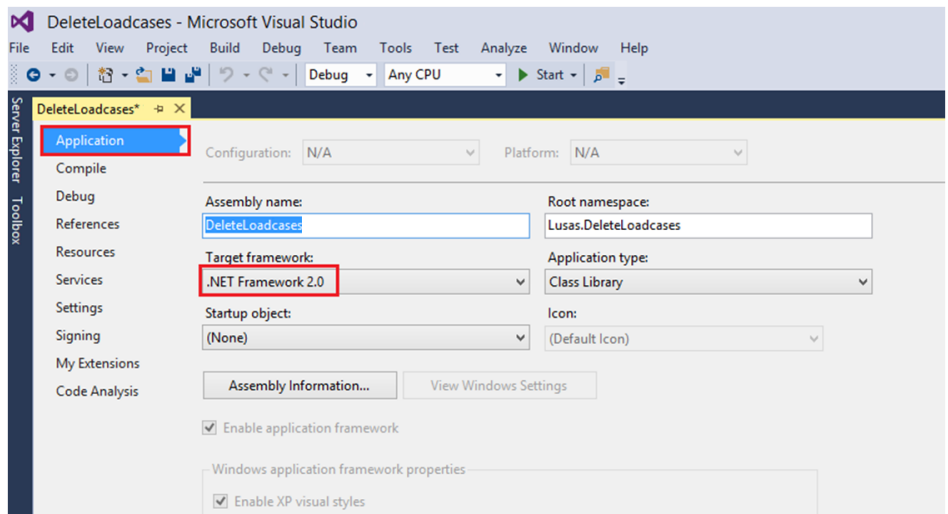
- If the wrong version of LUSAS is being referenced, click **Add** and search for the Lusas.Interop.dll file in the right version path,

Carrying on:

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



- On the **Application** page set the Target framework to **.NET Framework 2.0**

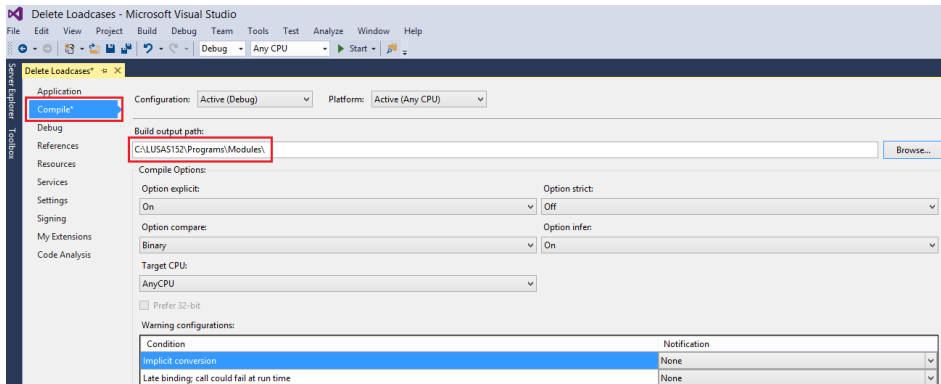


Creating dialogs using VB.NET

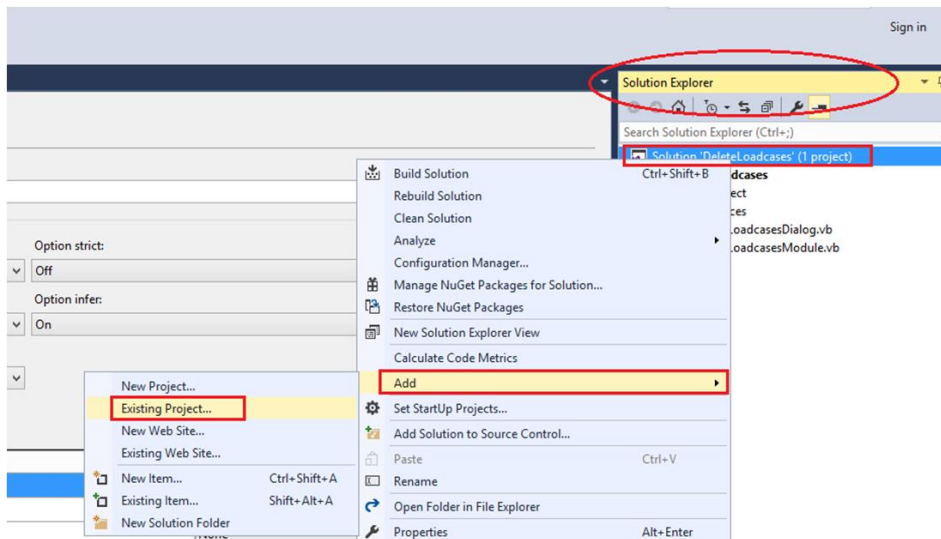


Note. For LUSAS v15.2, .NET Framework 2.0 is needed. Later versions of LUSAS will require a different target framework to be set.

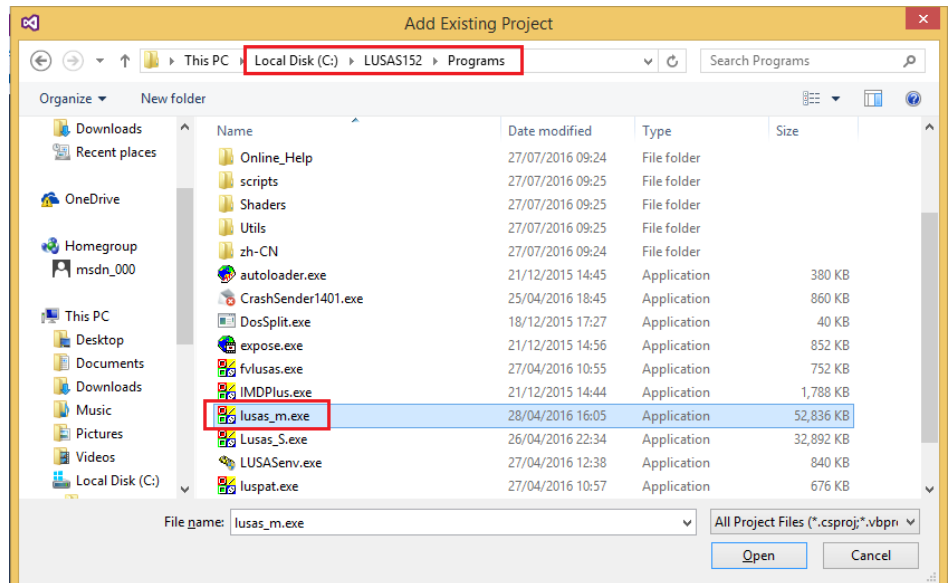
- On the **Compile** page browse and change the build output path to **C:\<LUSAS Installation Folder>\Programs\Modules**



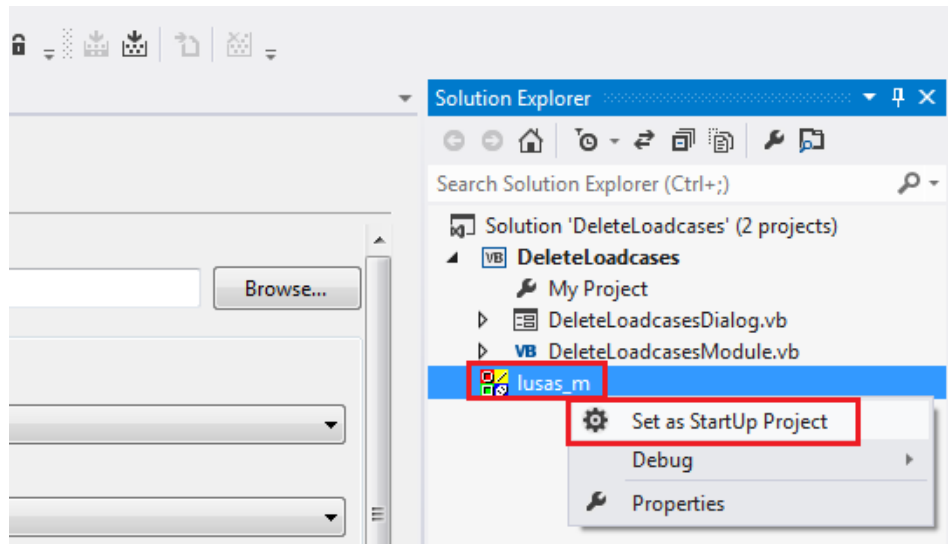
- From the Solution Explorer right-click on Solution 'DeleteLoadcases' and select **Add > Existing Project**



- On the **Add > Existing Project** dialog browse for the location of the version of LUSAS that you wish to run and press **Open**

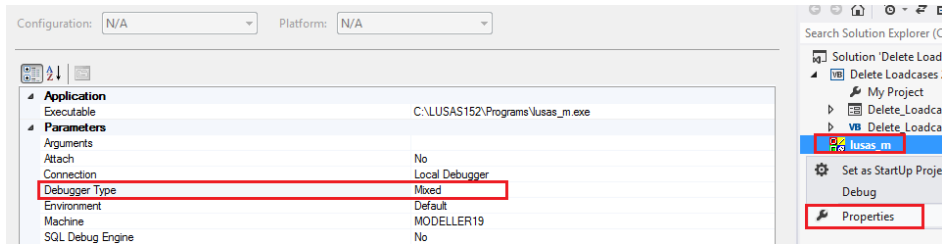


- In the Solution Explorer right-click on the **lusas_m** entry and select **Set as Startup Project**.



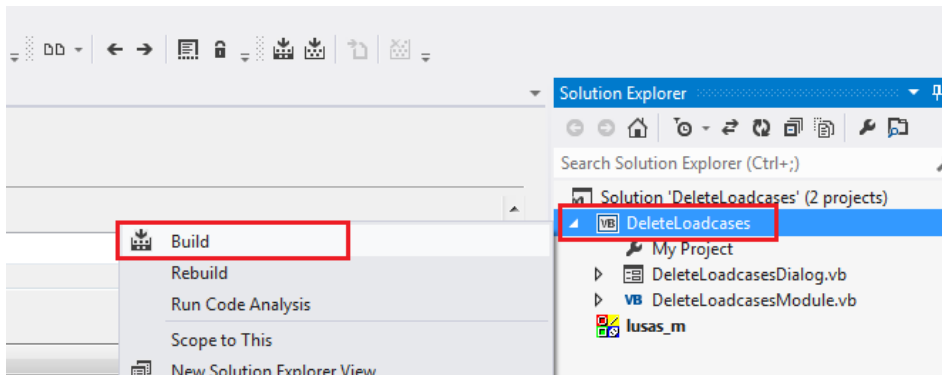
- In the Solution Explorer right-click on the **lusas_m** entry and select **Properties**, and set the Debugger type to **Mixed**

Creating dialogs using VB.NET



Build the project

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



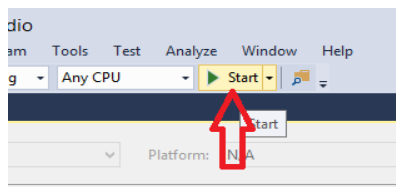
Note. In Visual Studio you can also press **F7** to build the project.

Details of the build will appear in the output window.

```
DeleteLoadcases -> c:\LUSAS151\Programs\Modules\DeleteLoadcases.dll
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

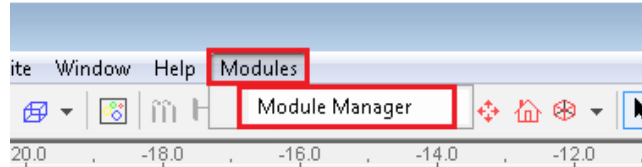
Run the project

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

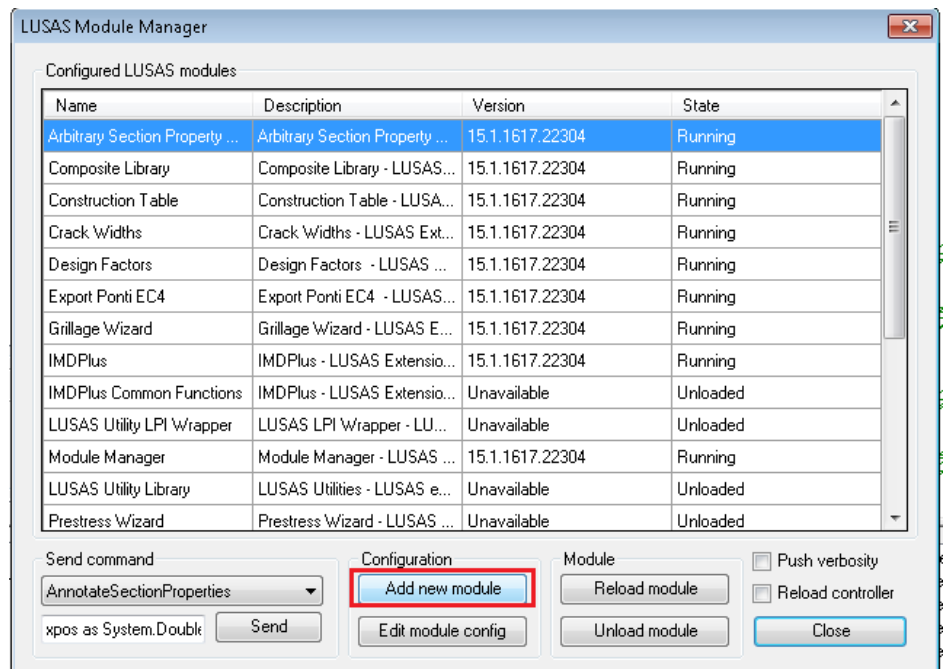


LUSAS Modeller should start.

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

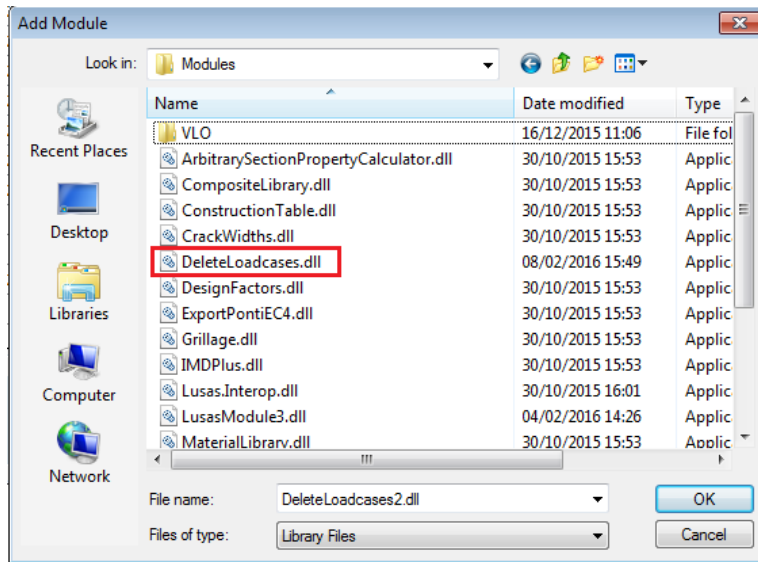


- Click **Add new module**

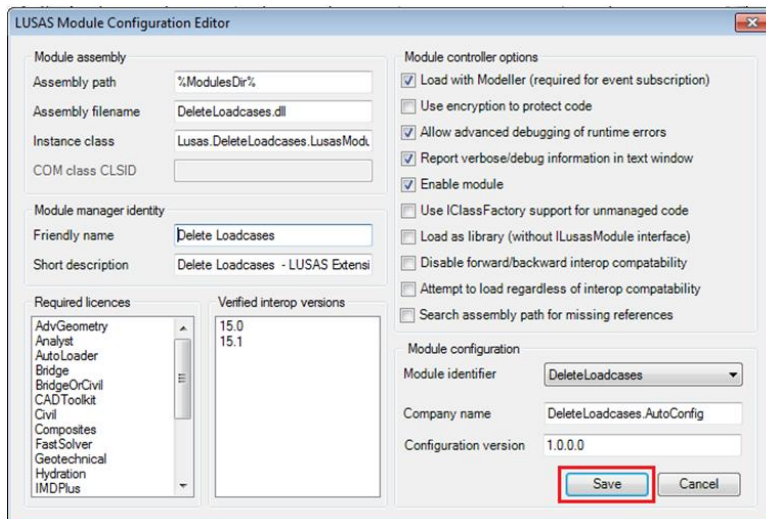


- Browse on the Add Module dialog for **C:\<LUSAS Installation Folder>\Programs\Modules\DeleteLoadcases.dll** and click **OK**

Creating dialogs using VB.NET

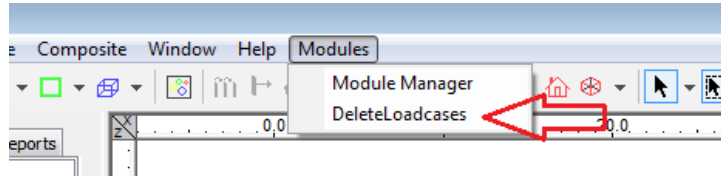


The LUSAS Module Configuration Editor dialog will appear. This permits (amongst many other features) restricting the Module to only work with a particular LUSAS version or licence key. No additional settings should be made on this dialog for this example.

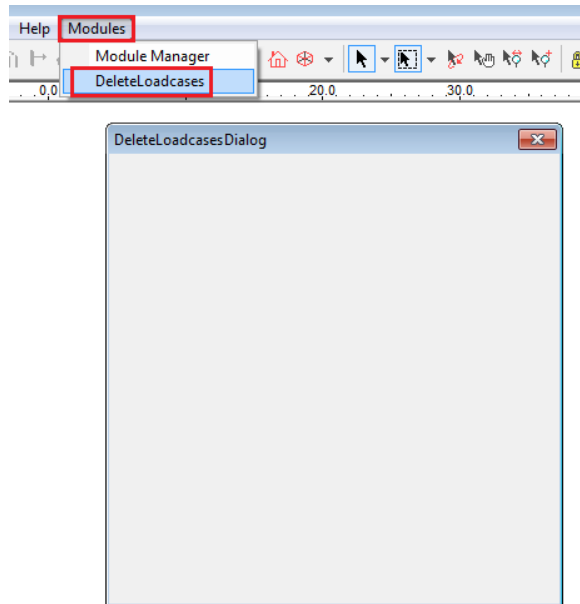


- Press the **Save** button and then **Close**

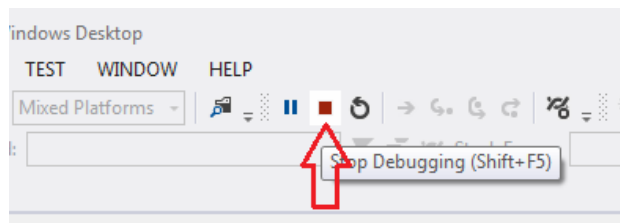
From now on, you will see the **Delete Loadcases** item in the modules menu.



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

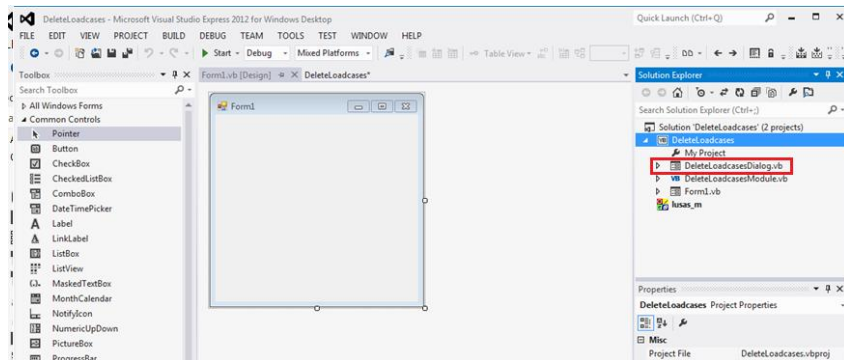


- Click on **Stop Debugging** in Visual Studio to close LUSAS Modeller and amend the project.

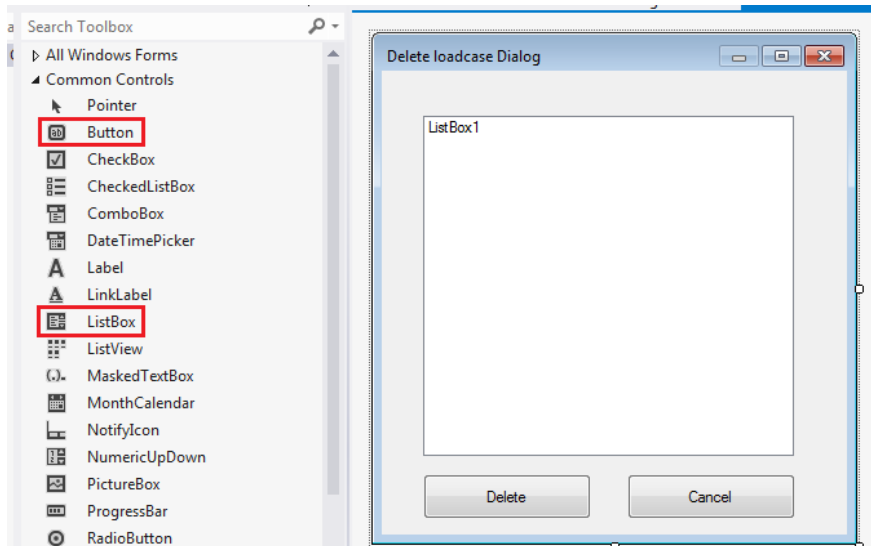


Adding dialog controls

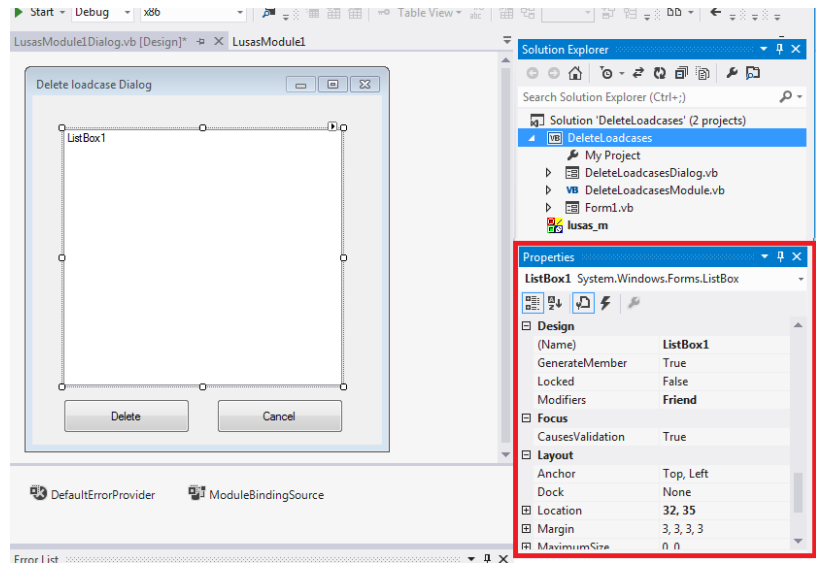
- In Visual Studio double-click on **DeleteLoadcasesDialog.vb** to open a blank dialog.



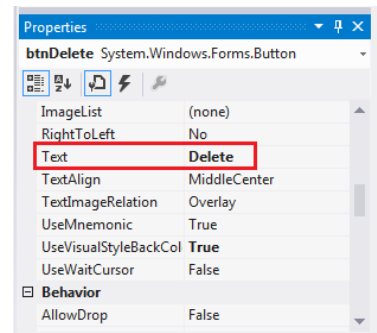
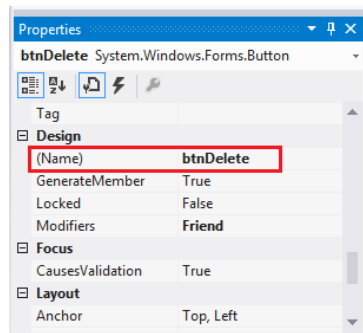
- Add a ListBox and two Buttons



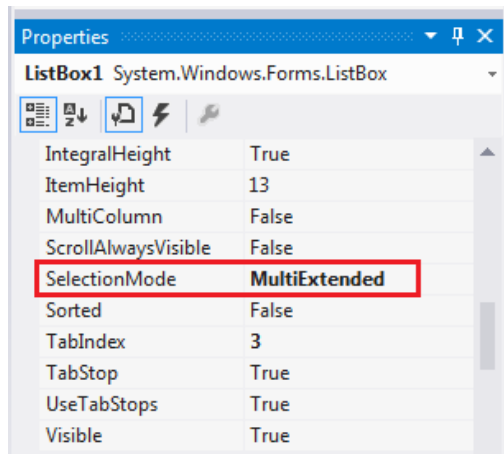
- Select each of the controls to modify its properties (Name, location, size...)



- For Button 1 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 to create an event handler for the form load event

```
Private Sub Delete_LoadcasesDialog_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 script text required. 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 Script required, open the PDF file for this guide and for the remainder of this section of the Guide copy and paste the script 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

```

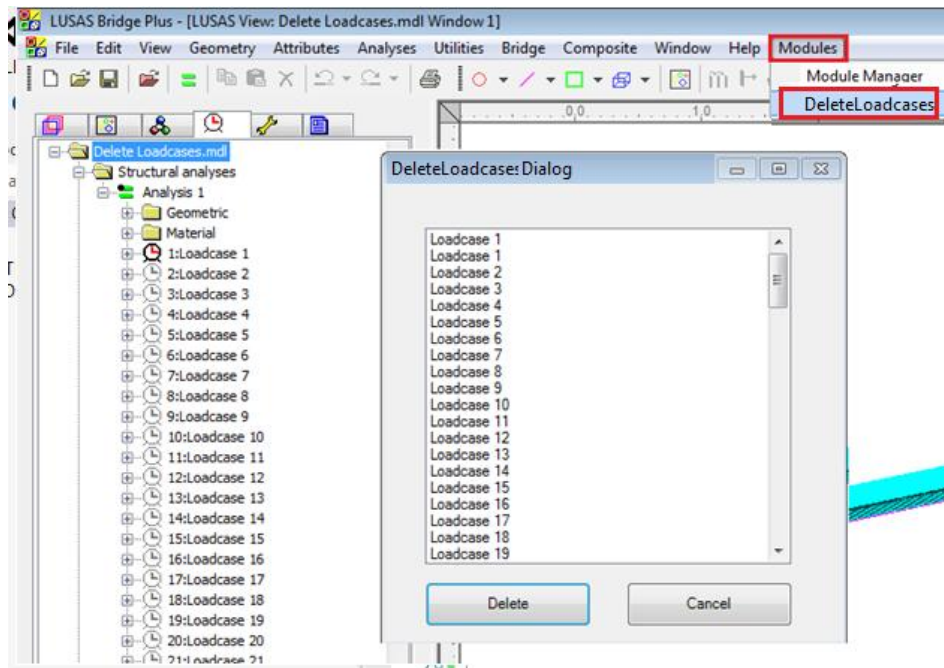
Defining a Cancel Button

To cause the dialog to close:

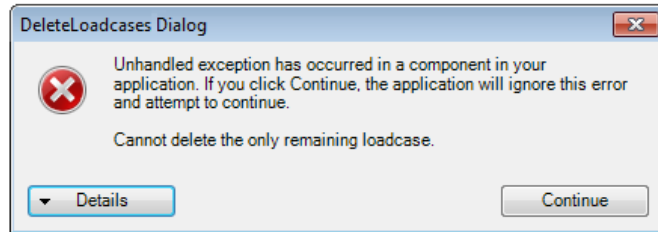
- Double-click on the button **Cancel** to create an event handler for this buttons click even and modify it to:

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

- Now build the project by pressing the **F7** key and run it by pressing the **F5** key.
- In LUSAS Modeller select the menu item **Modules> DeleteLoadcases**, then select which loadcase to delete and press the **Delete** button.



You may have noticed that if you try to delete all loadcases you get the following:



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

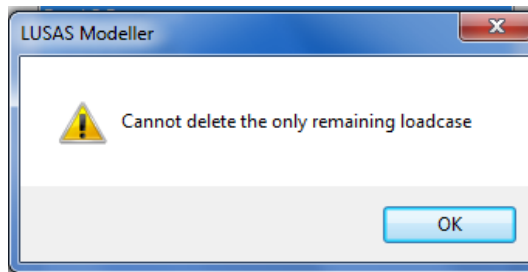
        Call moduleObject.Modeller.AfxMsgBox(ex.Message())

    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.

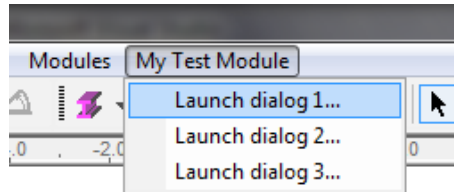
When a module creates a new menu item, Modeller will return a unique id for that menu item, these id's should be stored in “member” variables within the module. Modules can create new menu items in the onRefreshMainMenu function as shown below:

```
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 = menu.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="editingObj">Object that is being edited (nothing when
creating a new object).</param>
''' <param name="clientData">Data that was provided to Modeller when
defining editingObj.</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 editingObj 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 may be enabled or disabled in the **onMenuUpdate** event

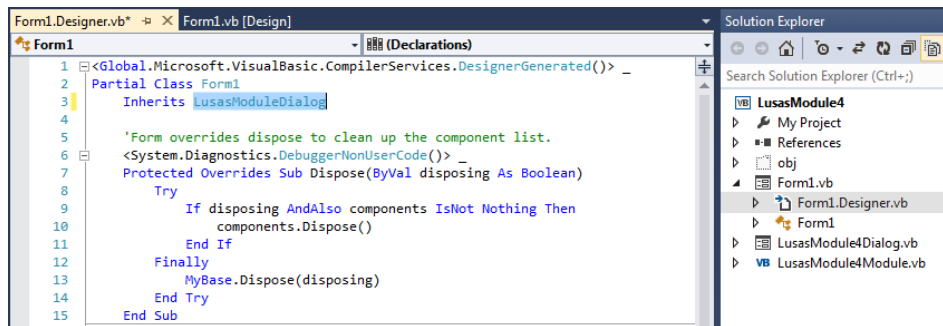
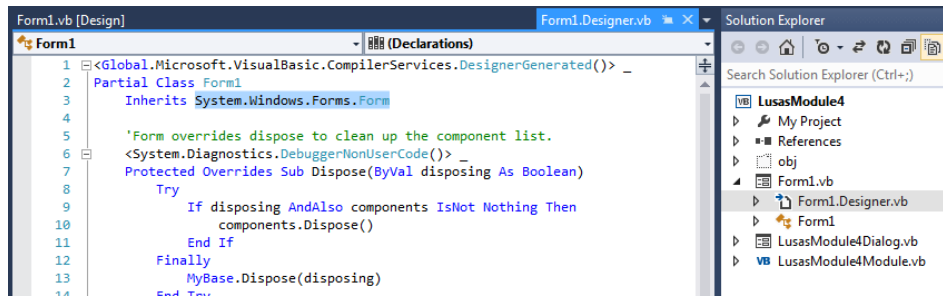
```
''' <summary>
''' Called when a menu entry needs to be drawn.
''' Allows the Module to specify whether the menu item should be disabled
or checked.
''' </summary>
''' <param name="menuID">ID of the menu that has been clicked.</param>
''' <param name="editingObj">Object that is being edited (nothing 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 editingObj.</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 editingObj 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
```



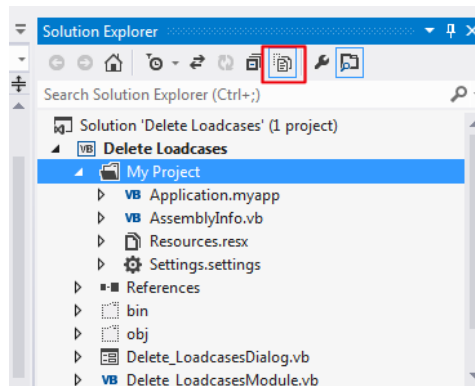
Note. All dialogs must inherit from LusasModuleDialog. When adding a new dialog you should change the code in the dialog designer to inherit from LusasModuleDialog rather than System.Windows.Forms.Form

Creating dialogs using VB.NET



Translation considerations

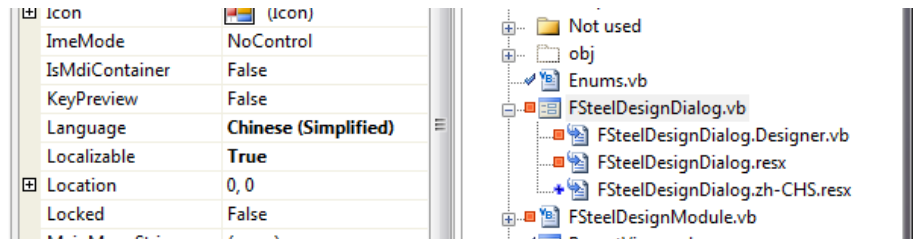
- The default language should always be **English**. All strings should be defined in the Resources.resx file. To access the Resources.resx file pick the **Show All Files** button in the Solution Explorer.



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 an 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:

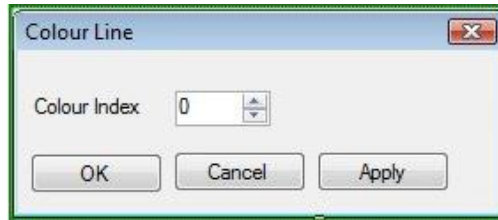
English:

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 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.

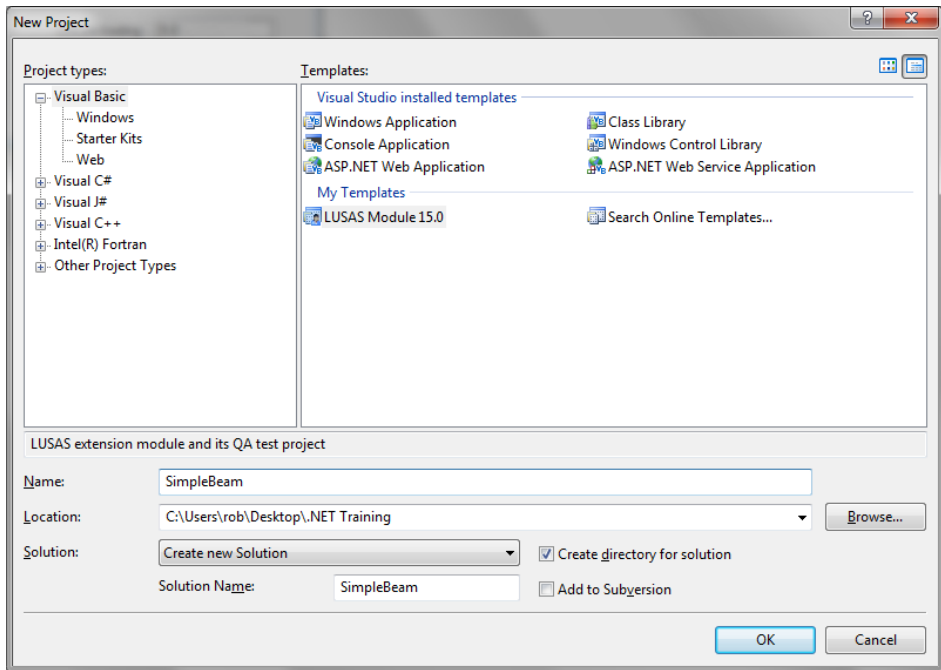
In order to drive LUSAS from a standalone application via COM (Component Object Model), LUSAS must be installed and licenced. When creating a COM instance of LUSAS a licence will be used. The licence will be in use for the lifetime of the instance and must be properly disposed of to release the licence.

Application Example

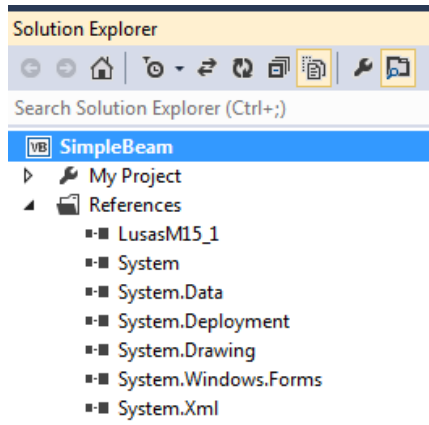
To illustrate the process involved, a stand-alone application called SimpleBeam will be created. The application will accept two parameters, length and load. The application will use LUSAS to analyse the beam and return the results for the maximum bending moment.

Create a new project

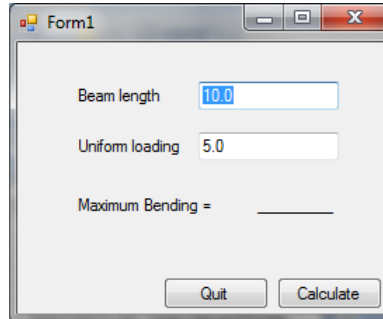
- In Visual Studio create a new Windows Application called SimpleBeam.



- Add a reference to LUSAS Modeller.
- In Solution Explorer, click **Show All Files**
- Right-click on **References** and select **Add reference**
- In the COM tab select **LUSAS Modeller ActiveX Script Language 15.1**
(later versions of LUSAS will require a different selection)



- Create the following dialog:



For simplicity all code is placed in the dialog as follows

```
Imports LusasM15_1

Public Class Form1

    Private m_lusas As LusasM15_1.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 LusasM15_1.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() = 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() = 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()


' Get the solverOptions object and set the defaults
```

```
Dim solverOptions As IFLusasRunOptionsObj =
m_lusas.solverOptions()

solverOptions.setAllDefaults()

' Get the exporter object that will export the model to solver
Dim solverExport As IFTabulateDataObj = m_lusas.solverExport()

' Set the defaults
solverExport.setAllDefaults()

' Set a filename
solverExport.setFilename(tempFilePath &
"beam.dat").setSearchAreaFileOn()

' Export the model as a solver data file (.dat)

Dim returnCode As Integer = m_lusas.db.exportSolver(solverExport,
solverOptions)

If (returnCode <> 0) Then

    ' If the export fails we cannot run the analysis
    MsgBox("The tabulation failed")

    Return

End If

' Save the model before solving
m_lusas.db.saveAs(tempFilePath & "beam.mdl")

' If we successfully exported the .dat file we can run the analysis

returnCode = m_lusas.solve(tempFilePath & "beam.dat",
solverOptions)

If (returnCode <> 0) Then

    ' If the analysis fails we cannot access the results
    MsgBox("The analysis failed")
```

```

Return

Else

    ' if the analysis is sucessfull load the results

    m_lusas.database.openResults(tempFilePath & "beam.mys")

    m_lusas.database.getResultsCache().calculateNow()

End If

' *** 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) momnet

        If my < maxMom Then

            maxMom = my

            nodeNum = node.getID()

        End If

    Next

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 ' String.Format("{0:0.000}{1}{2}", 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

    End For
```



```

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.15.2");

// 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();

```



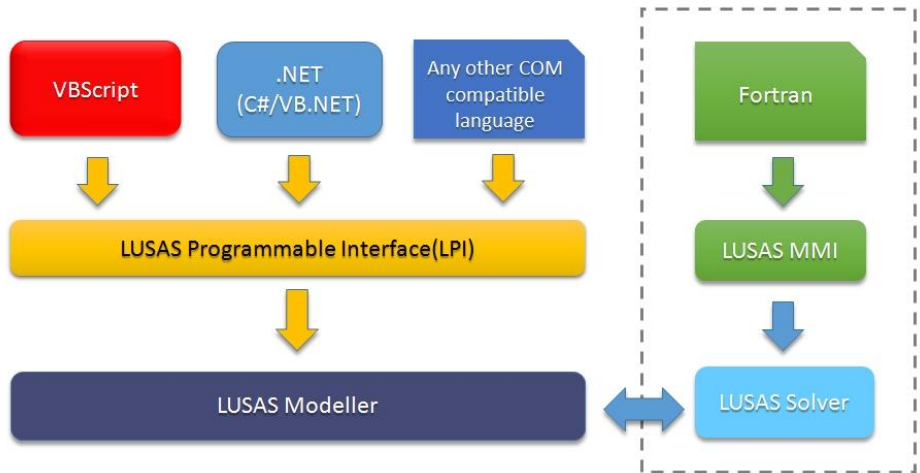
Note. The LPI functions often return a base class pointer which often needs to be downcast to the desired type (e.g. attribute -> material). VB will do this for you, but C++ will not. Therefore you must explicitly cast, and catch any exceptions that may result



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 | Email: info@lusas.com | www.lusas.com