PyramidGenerator

Contents

Project main assumptions

The goal of the project is to create an Extension module which generates the pyramid in Revit:

- The size of the pyramid will be determined by the width and the height parameters.
- Elements of the pyramid will be generated as FamilyInstances of the structural framing category.
- The module will present all defined data in the HTML note.
- The parameters will be saved in the model and it will be possible to modify the structure.
- The module will be implemented based on the WPF technology
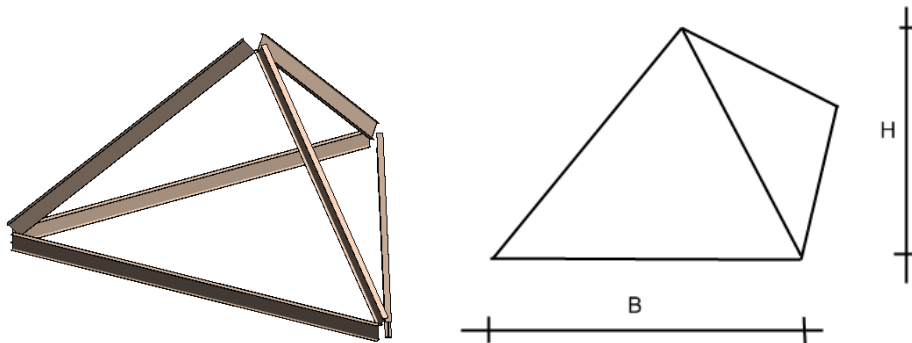
User interface

The module will consist of two layouts:

Parameters
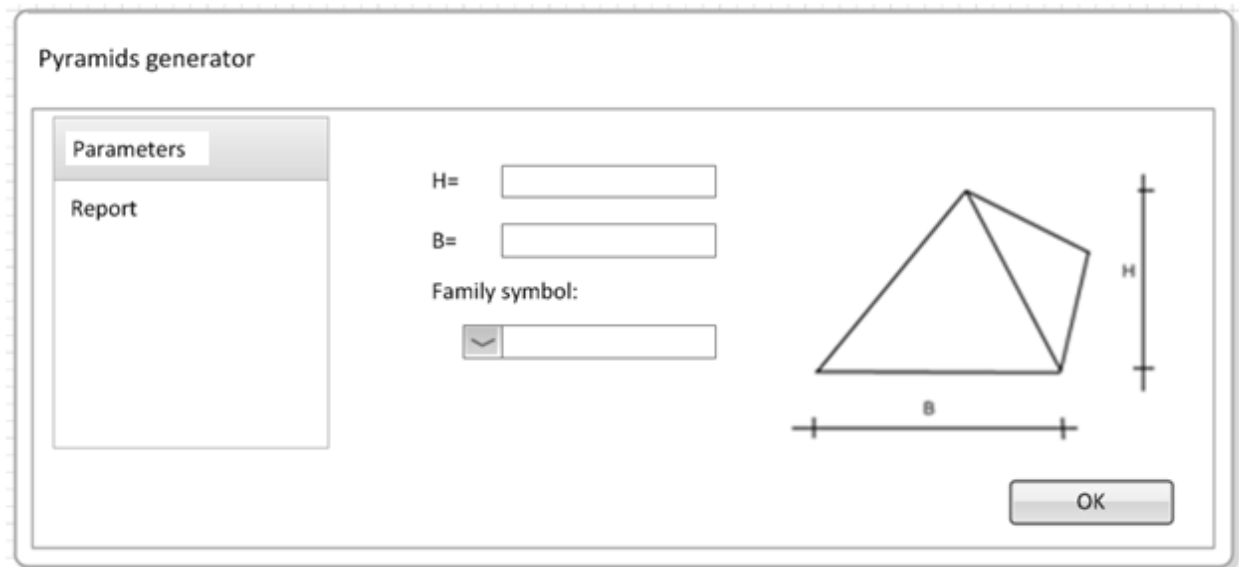
The Parameters layout will allow the user to define following properties of the pyramid:

- The height
- The width

- The Family which will be used for building edges of the pyramid

Report

Pyramids generator

Parameters	REPORT
Report	

OK
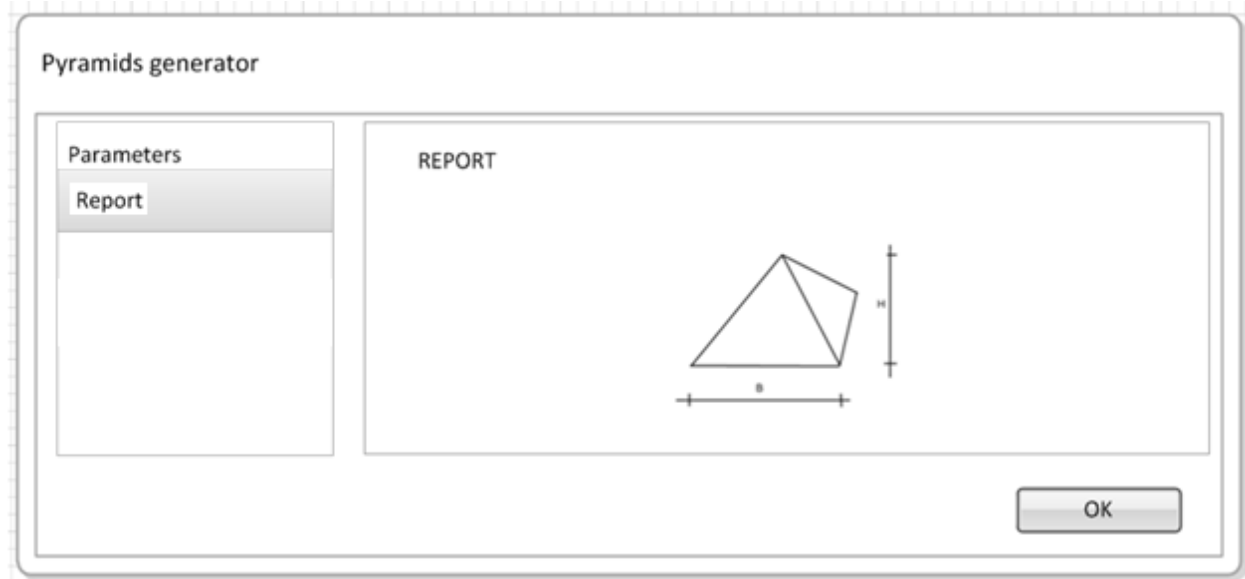
The Report layout will present all parameters of the pyramid in the HTML note.

Additional remarks

For simplification the pyramid will be generated on the first detected level in the point of 0,0,0 coordinates.

1. Step: Project creation

Project wizard

The Extension project is created by the wizard which allows the user to configure its basic settings.

1.1. Start Microsoft Visual Studio (the example is made using VS 2010)

1.2. Start a new project:

    1.2.1.   Start the new project: File->New->Project

    1.2.2.   Select the Revit Extension WPF project:

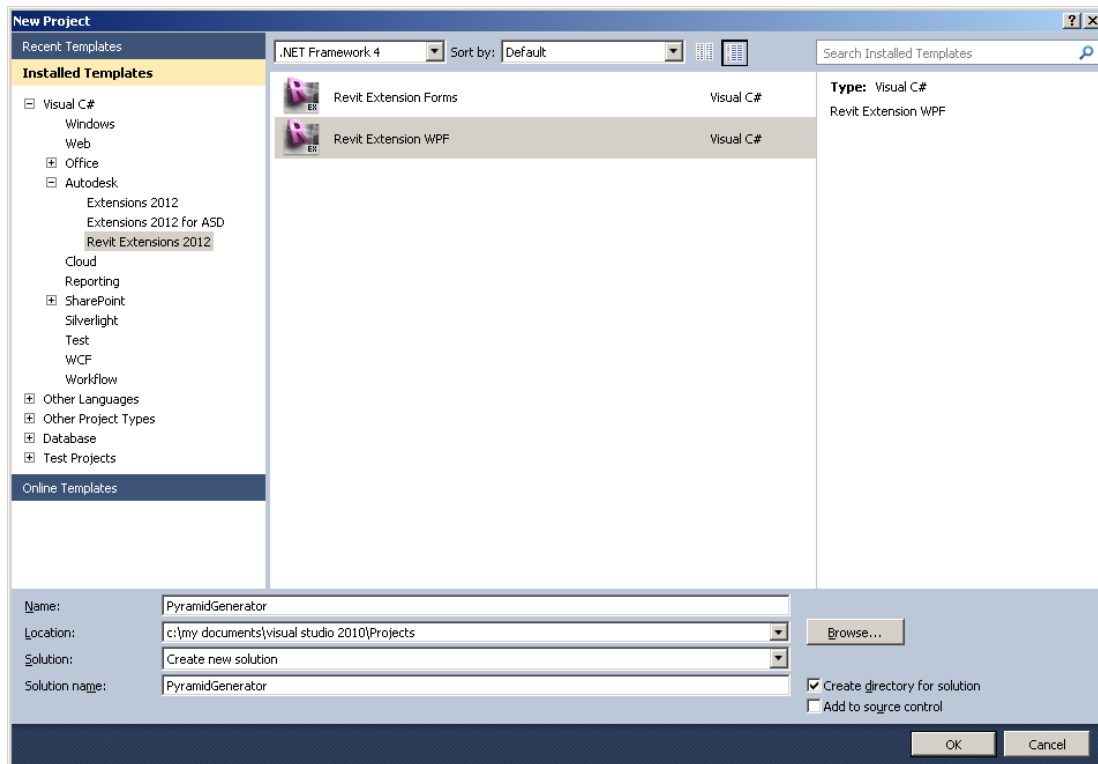   Visual C#->Autodesk->Revit Extensions 2012->Revit Extension WPF

    1.2.3.   Enter the name of the project: PyramidGenerator

    1.2.4.   Click OK button (as a result the wizard form should appear).

1.3. Fill the Revit Extension wizard form:

1.3.1. Define the name of the Extension and its description:

Name: PyramidGenerator
Description: PyramidGenerator

In case of RevitAddin generated by the project the name of the Extension will be used as a text and the description will be used for a tooltip.

In case of integration with Extensions system this name will be shown on the Extension ribbon.

1.3.2. Define RevitAddin options:
   ▪ Product visibility (Architecture, Structure, MEP)
      PyramidGenerator is visible in all Revit products.
   ▪ Vendor information (Id, Description) – up to the user
   ▪ Addin file:
   ▪ The wizard generates the addin file for the current module. This addin file may be copied every time the project is built.
   ▪ The Addin file may point at the assembly file generated in a Debug or in a Release project folder.
1.3.3. Define user interface options
      The main element of the Extension is a dialog. It consists of three main elements:

- Layouts list – to activate a particular layout
- User control – to show an active layout
- Bottom user control – to show an active bottom layout

All of them may exist in different combinations which are available in the wizard:
- User control
- Layouts list + User control
- Layouts list + User control + Bottom user control

Layouts list	
	User control
Bottom user control	
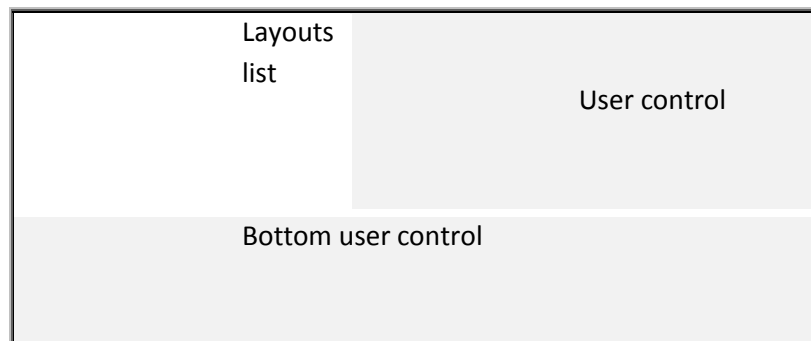
Additionally the module may have added following elements:
- Menu
- ToolBar
- Toolbar Menu
- Status bar

For PyramidGenerator the dialog is needed with a Layouts list and a UserControl (for now without any additional options).

> **Note:**
> If the project is generated and some modification of the user interface is required, it is possible to make it in dedicated methods of the Extension class: see the OnSetLayout method.

1.3.4. Define Advanced settings:

In the advanced settings the user may decide about integration with Extensions system. If this option is used there will be additional files generated which will allow the user to attach to an Extensions ribbon.

PyramidGenerator will be assigned to a new SDK category:

It is also possible to add a start project to the solution. Thanks to this it is possible to launch a module without Revit. It can be very useful while designing the UI. The only thing the user has to remember about is a proper separation of the code which is RevitAPI dependent from API independent part.

1.3.5.   Generate the project (as a result the new project should appear)

Project references

In the project there are many references added. The wizard should resolve all problematic one.

Revit:

- RevitAPI – the part of RevitAPI user interface independent.
- RevitAPIUI – the part of RevitAPI responsible for interaction with user interface.

REX system:

- Autodesk.REX.Framework – the core of the Extension system.
- REX.Foundation, REX.Foundation.WPF – the core of the Extension module.
- REX.Controls, REX.Controls.WPF – WPF controls.

REX tools:

- REX.ContentGenerator – the component which provides tools for Revit families' generation and database access.
- REX.Geometry – the component which provides support for mathematical operations (mainly geometry).

Project files

Generated files may be divided into two groups:

- Functional classes
- User interface classes

Functional classes

There are three functional classes which require attention. There are:

- Extension – the center point of the Extension module.

- ExtensionRevit – the class which is responsible of the module behavior in the Revit context.
- Data – the data container which is accessible for all Extension objects.

Additionally there are generated files: Application, DirectAccess, Foundation, Settings, Results, which don't have to be modified by the user.

Note:
Extension and ExtensionRevit classes were separated in order to split the Revit context from the API independent part. For instance when some elements have to be read from Revit this action should be carried out in the ExtensionRevit class, but when the dialog is created (API independent) it can be done directly in the Extension.

It is not obligatory to keep this convention. Everything can be done directly in the Extension class, but for purity of the code it is recommended to follow this organization of the project. This is also important when the start project is generated by the wizard (in this mode all action associated with Revit should be skipped).

User interface classes

There are two classes generated in the project:

- MainControl
- MainWindow

It is not necessary (and in fact shouldn't) to make any changes in these files.

2. Step: Preparing Data

The best point to start with is the Data class. As it was mentioned before it was designed to store all common data. In case of the PyramidGenerator it should contain all size parameters, a beam family and the list of available FamilySymbols which will be shown on the dialog:

2.1. Prepare the Data structure

```
internal class Data : REXData
{
        public double H { get; set; }
        public double B { get; set; }
        public string FamilySymbol { get; set; }
        public List<string> AvailableFamilySymbols { get; set; }
}
```

Note:
The REXSDK provides support for the units system defined by the product context (e.g. Revit). This issue is described in details in SDK documents. Generally units can be divided into three types:
- Base   – internal base REX units
- Interface – internal product units (e.g. Revit)
- Display   – units set currently in the project (e.g. Revit project)

For instance if we have a Revit project with the unit for the Length category set to millimeters there are:
- Base   – meter
- Interface – foot
- Display   – millimeter

Generally controls should be formatted according display units, data should be stored in base units and all operation on products API has to be made in interface units.
It is up to the user if he or she follows this recommendation. In this example all data are organized based on it.

2.2. Implement the Data::OnSetDefaults method

When the Extension launches after the Data creation the OnSetDefault method is called in order to initialize Data members:

```
protected override void OnSetDefaults(REXUnitsSystemType UnitsSystem)
{
    H = 5;//m
    B = 10;//m
    FamilySymbol = "";
    AvailableFamilySymbols = new List<string>();
}
```

3. Step: Preparing layouts

One of the most important user's tasks is to create layouts. As a layout it is seemed the control which is embedded in the module's dialog (other elements: the list, the dialog etc. are created automatically):
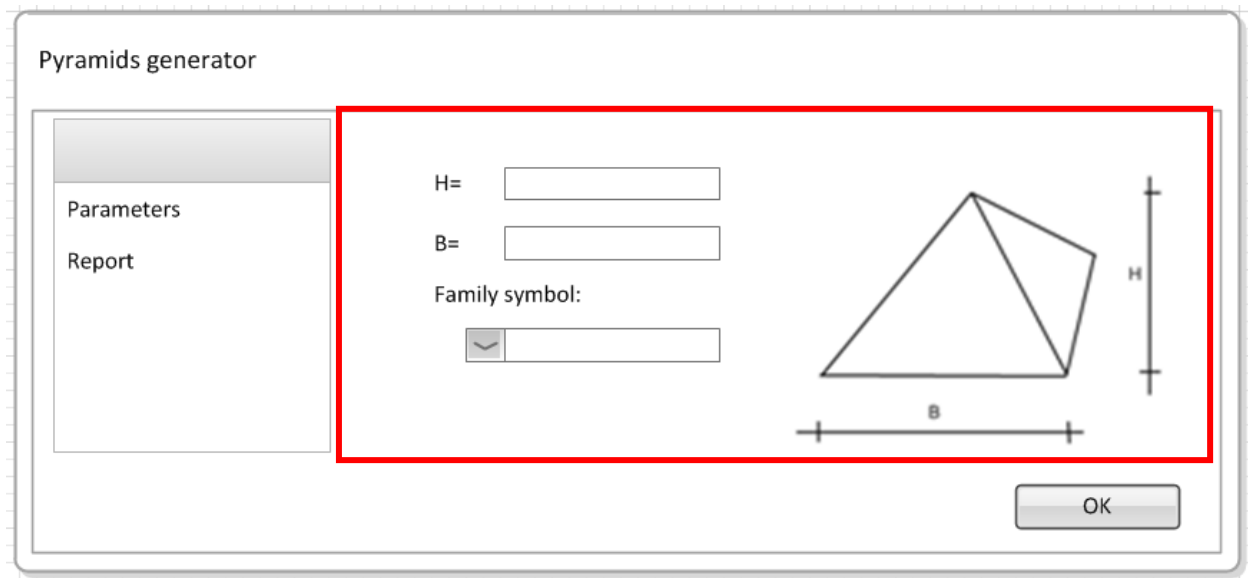
On the start there is SubControl.xaml
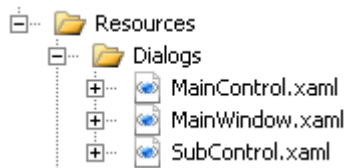
The user can use it directly or just create new files.

Pyramid layouts

In this example there is need for two controls which represents two layouts:

- PyramidParametersControl.xaml   – represents the Parameters layout
- PyramidNoteControl.xaml   – represents the Report layout.

3.1. Parameters layout

3.1.1.   Add the new control to the project

3.1.1.1. Remove SubControl from the project.

3.1.1.2. Add the PyramidParametersControl to the project:

3.1.1.2.1.   Add the New Item (Right click on Dialogs folder->Add->New Item...)

3.1.1.2.2. Choose the template of Extension control (Visual C#->Autodesk->Extension 2012->Extension Control WPF)

3.1.1.2.3. Enter the name of the new control (PyramidParametersControls)
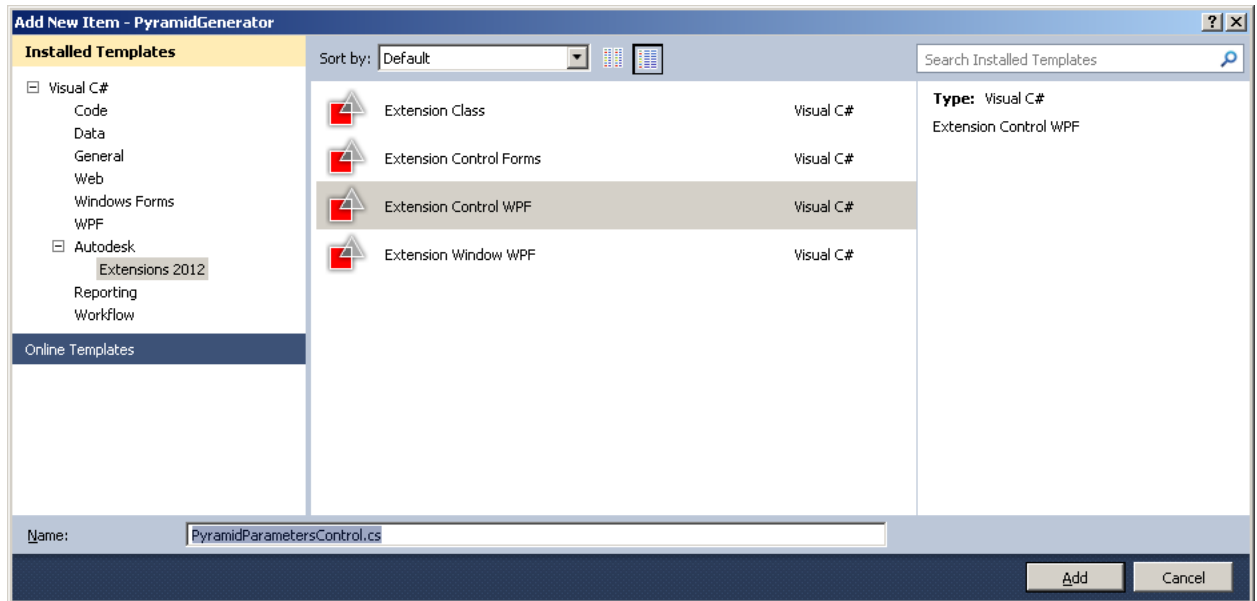
3.1.1.2.4. Accept the choice with the Add button.

The new control was added to the project.

3.1.2. Prepare the layout

The Parameters layout should contain three labels, two editboxes, one combobox and the picture of the pyramid. Because text fields will display the length value the REXEditBox or REXUnitEditBox control should be used (they provide automatic support for Revit units). In order to show how to use them both the text field which presents the height of the pyramid will be created as the REXEditBox and the control which describes the width of the pyramid will be the instance of the REXUnitEditBox. Other controls will be made based on classical Windows controls.

3.1.2.1. Prepare the arrangement of the layout:
3.1.2.1.1. Add controls to the grid
3.1.2.1.2. Set appropriate properties of controls
3.1.2.1.3. Create the image of the pyramid and add it to the project
3.1.2.1.4. Assign the added bitmap as a source of the image control
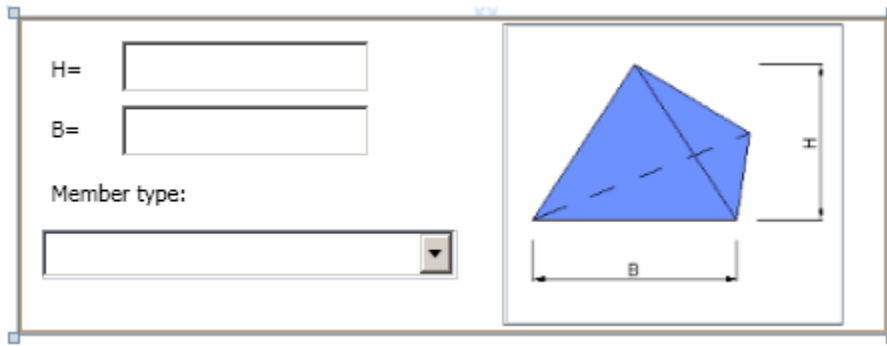
```xml
<rexf:REXExtensionControl
x:Class="REX.PyramidGenerator.Resources.Dialogs.PyramidParametersControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:rexf="clr-namespace:REX.Common;assembly=REX.Foundation.WPF"
    xmlns:rexc="clr-namespace:REX.Controls.WPF;assembly=REX.Controls.WPF"
    Height="158" Width="435">
    <Grid Height="158" Width="435">
            <Label Name="labelH" Content="H=" HorizontalAlignment="Left"
            Width="34" Height="24" VerticalAlignment="Top" Margin="12,14,0,0"
            />
            <rexc:REXEditBox x:Name="editH" RememberCorrect="True"
            Height="26" Margin="0,12,259,0" VerticalAlignment="Top"
            HorizontalAlignment="Right" Width="124"/>
            <Label Name="labelB" Content="B=" HorizontalAlignment="Left"
            Width="34" Height="24" VerticalAlignment="Top" Margin="12,44,0,0"
            />
            <rexc:REXUnitEditBox x:Name="editB" RememberCorrect="True"
            Height="26" Margin="52,44,0,0" VerticalAlignment="Top"
            HorizontalAlignment="Left" Width="124"
            UnitType="Dimensions_StructureDim" Power="1"/>
            <Label Name="labelFamSymbol" Content="Member type:"
            HorizontalAlignment="Left" Width="164" Margin="12,76,0,58" />
            <ComboBox Name="comboFamilySymbol" Margin="12,106,216,0"
            Height="24" VerticalAlignment="Top" />
            <Border Margin="0,3.5,22,3.5" Width="170" BorderThickness="1"
            BorderBrush="Gray" HorizontalAlignment="Right"
            Background="White">
                    <Image  VerticalAlignment="Top"
                    Source="/PyramidGenerator;component/Resources/Other/Images/
                    IMAGE_Pyramid.png" HorizontalAlignment="Left"></Image>
            </Border>
        </Grid>
</rexf:REXExtensionControl>
```

Note:

The REXEditBox and the REXUnitEditBox provides the same functionality (more information about this controls can be found in dedicated documents). The only significant difference is the way of managing it. The REXEditBox is managed through the outer DFM object (more about it in next steps) while the REXUnitEditBox can be used directly. At the design stage it is also possible to define the UnitType and the Power of the REXUnitEditBox (for REXEditBox it is made during registration process).

```xml
<rexc:REXEditBox x:Name="editH" />
```

```
<rexc:REXUnitEditBox x:Name="editB" UnitType="Dimensions_StructureDim"
Power="1"/>
```

3.1.2.2. Register unit controls

In the current module there are two unit controls: the REXUnitEditBox and the REXEditBox. To make them work it is necessary to connect them with the Extension unit system. The following actions should be taken in the constructor of the PyramidParametersControl:

3.1.2.2.1. Register REXEditBox control in the DFM

```
ThisMainExtension.DFM.AddUnitObject(editH,
EUnitType.Dimensions_StructureDim, 1);
```

3.1.2.2.2. Set the unit engine for the REXUnitEditBox:

```
editB.UnitEngine = ThisExtension.Units.UnitsBase;
```

Note:
The usage of unit controls should proceed in the following order:
- Connect with the unit system
- Setting constraints
- Setting values

3.1.2.3. Set constraints

For the unit control it is good to provide some constraints which will limit the range of values input into it. In the current example parameters B and H should be greater than 1 meter. In both cases the limit can be set in different units, in the current example base units are used. The following actions should be taken in the constructor of the PyramidParametersControl:

3.1.2.3.1. Set a minimum value for the REXEditBox:
```
ThisMainExtension.DFM.SetBaseMinValue(editH, true, 1);
```

3.1.2.3.2. Set a minimum value for the REXUnitEditBox
```
editB.RangeMinCheck = true;
editB.SetBaseMinValue(1);
```

Note:
Additionally the user is able to define its own validation rules by providing the external validation methods (through DFM or directly in the control):
`SetUserValidation` – for single values

```
SetUserValidationComplex – for complex values

The method has to decide if the specified value is against internal constraints. For example:
        private bool Valid(double val, IUnitObject uo)
        {
            if (val > 3)
                return true;
            else if (val < 0)
                return true;
            else
                return true;
        }
```

3.1.2.4. Fill the controls with current Data.

When the module starts, the controls should be filled with parameters from the Data object:

3.1.2.4.1.   Create the SetDialog method

3.1.2.4.2.   Fill editH through the DFM object:

```
ThisMainExtension.DFM.SetDataBase(editH,
ThisMainExtension.Data.H);
```

3.1.2.4.3.   Fill editB through direct methods:

```
editB.SetDataBaseValue(ThisMainExtension.Data.H);
```

3.1.2.4.4.   Fill comboFamilySymbol with the list of available FamilySymbols and set the selected element:

```
        comboFamilySymbol.Items.Clear();
foreach (string familyString in
ThisMainExtension.Data.AvailableFamilySymbols)
            {
                comboFamilySymbol.Items.Add(familyString);
            }

comboFamilySymbol.SelectedItem =
ThisMainExtension.Data.FamilySymbol;
```

3.1.2.5. Update Data with parameters set in the control

Extension Data has to be updated with parameters set in the control when the dialog is validated (on OK). Additionally it has to be updated before the report refresh and before saving the parameters to a file:

3.1.2.5.1.   Create the SetData method

3.1.2.5.2.   Update the height

```
ThisMainExtension.Data.H =
ThisMainExtension.DFM.GetDataBase(editH);
```

3.1.2.5.3.   Update the width

```
        ThisMainExtension.Data.B = editB.GetDataBaseValue();
```

3.1.2.5.4. Update the family

```
ThisMainExtension.Data.FamilySymbol =
comboFamilySymbol.SelectedItem.ToString();
```

> **Note:**
> The convention of creating SetData and SetDialog method is recommended but not obligatory. It follows the logic of Extension system which will be presented later.
>
> It is possible to use binding as well by using properties of the REXEditBox and REXUnitEditBox:
> - REXEditBox.DoubleValueProperty
> - REXUnitEditBox.DoubleValueBaseProperty

3.2.   Note layout

3.2.1. Add the PyramidNoteControl control to the project (in the same way as the PyramidParametersControl)

3.2.2. Prepare the arrangement of the layout:

The PyramidNoteControl layout is very simple. It should contain only one control which would present the HTML report. The WebBrowser control is good enough for this purpose.

```xml
<rexf:REXExtensionControl
x:Class="REX.PyramidGenerator.Resources.Dialogs.PyramidNoteControl"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:rexf="clr-
namespace:REX.Common;assembly=REX.Foundation.WPF"
    xmlns:rexc="clr-
namespace:REX.Controls.WPF;assembly=REX.Controls.WPF"
    Height="300" Width="500">
    <Grid>
        <WebBrowser Name="webBrowser"/>
    </Grid>
</rexf:REXExtensionControl>
```

3.2.3. Fill the note

The html note will be created through ROHTML component provided with the SDK. The project should have added the reference to its interop on the start:

> **Note:**
> The ROHTML is a very powerful component which provides ready solutions for data presentation. The document is represented by IXHTMLDocument which allows to define predefined elements (through

the Body member) with different styles:
1. Headers
2. Values
3. Texts
4. Tables
5. Note headers

It is also possible to embed own html code (just put the code as the input value instead of regular string).

The ROHTML generates the document which may be saved in html, htm or text format.

3.2.3.1. Add the member of IXHTMLDocument to the PyramidNoteControl:

```
IXHTMLDocument HtmlDocument;
```

3.2.3.2. Add the member which stores the path to the current document:

```
string DocumentPath;
```

3.2.3.3. Create the SetDialog method

3.2.3.3.1.   Initialize the HtmlDocument and release the current one:

```
if (HtmlDocument != null)
{
    HtmlDocument = null;
    ThisExtension.ROHTMLReleaseDocument();
}

HtmlDocument = ThisExtension.ROHTMLDocument();
```

3.2.3.3.2.   Remove the current file and create the path to the new one (in the example every time it has a different name – due to the WebBrowser refresh problems):

```
if (File.Exists(DocumentPath))
{
     try
    {
            File.Delete(DocumentPath);
    }
    catch{}
}

Guid g = System.Guid.NewGuid();
DocumentPath = GetTempPath() + "\\" + g.ToString() +
".mht";
```

The destination directory is created by the GetTempPath method which creates a new folder (PyramidGenerator) in the user's temporary directory:

```csharp
public string GetTempPath()
{
    string path = Environment.GetEnvironmentVariable("temp");
    if (!Directory.Exists(path))
    {
        path =
        System.IO.Path.GetFullPath(System.IO.Path.Combine(Env
        ironment.GetFolderPath(Environment.SpecialFolder.Loca
        lApplicationData), "..\\Temp"));
        if (!Directory.Exists(path))
            Directory.CreateDirectory(path);
        }

        path = path + "\\PyramidGenerator";
        if (!Directory.Exists(path))
            Directory.CreateDirectory(path);

        return path;
    }}
```

3.2.3.3.3.   Generate a short note with information about parameters of the current structure.

```csharp
HtmlDocument.Body.AddHeader(1, "Pyramid parameters");
string textH =
ThisExtension.Units.DisplayTextFromBase(ThisMainExtensio
n.Data.H,Autodesk.REX.Framework.EUnitType.Dimensions_Str
uctureDim, true);
HtmlDocument.Body.AddValue2("H", textH);

string textB =
ThisExtension.Units.DisplayTextFromBase(ThisMainExtensio
n.Data.B,Autodesk.REX.Framework.EUnitType.Dimensions_Str
uctureDim, true);
HtmlDocument.Body.AddValue2("B", textB);
```

3.2.3.3.4.   Save the current document:

```csharp
HtmlDocument.SaveAsSingleFile(DocumentPath);
```

3.2.3.3.5.   Set the file as a source of webBrowser:

```csharp
webBrowser.Navigate(new Uri(DocumentPath));
```

3.2.4.   Clear the resources when dialog is closed

When the Extension is closing all resources should be released. In case of the Note layout there is a folder with the html document:

```csharp
public void Release(){
{
    if (File.Exists(DocumentPath))
    {
        File.Delete(DocumentPath);
    }
```

```csharp
        string tempPath = GetTempPath();

        if (Directory.Exists(tempPath))
        {
                Directory.Delete(tempPath);
        }
}
```
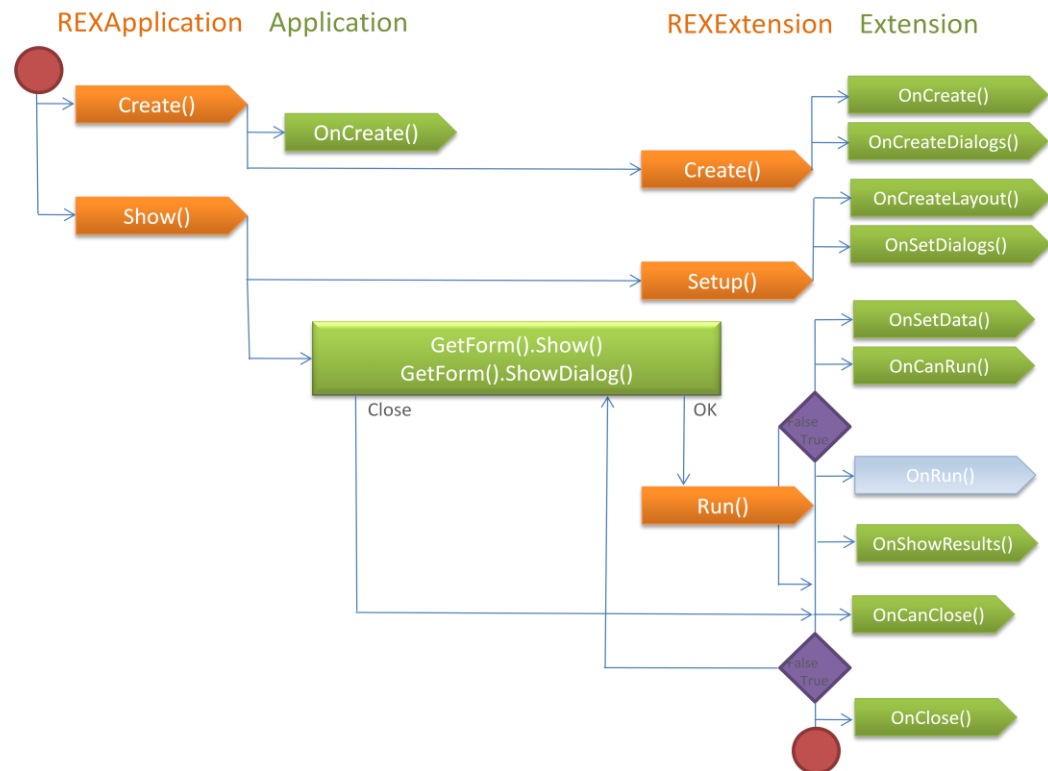
4. Step: Extension class

The center point of the Extension module is the Extension class. It allows the user to manage the whole module. The Extension class may be used directly but to make the code a little bit better organized there is an additional class ExtensionRevit which allows separating the API independent from the API dependent code.

Note:

The cascade of methods called during the life of the module:

The Extension class provides definitions for the group of methods, including:

1. OnCreate
2. OnCreateDialogs
3. OnCreateLayout
4. OnSetDialogs
5. OnSetData
6. OnCanRun
7. OnRun

The programmer's task is to fill appropriate methods with the code to manage the module. It can be done directly in the body of the method or in the dedicated ExtensionRef object (in the method that suits it):

```
public override void OnCreate()
{
    base.OnCreate();
```

```
            // insert code here.

            if (ExtensionRef != null)
                ExtensionRef.OnCreate();
        }
```

The ExtensionRef object allows to separate parts which are context dependent (e.g. to be able to launch the module outside the Revit environment for a design part).

4.1. Prepare the members of prepared controls in the Extension class

```
private Resources.Dialogs.PyramidNoteControl NoteControlRef;
private Resources.Dialogs.PyramidParametersControl ParameterControlRef;
```

4.2. Implement the ExtensionRevit::OnCreate method

The OnCreate method is called at the beginning of the module start. This is the place where:

- All not UI objects should be initialized
- Revit selection should be analyzed
- Data should be filled
- Etc.

If the module cannot be launched (e.g. because of inappropriate selection) the user may inform the system about this fact in the OnCreate method. It is done through the Errors member of the System object:

```
System.SystemBase.Errors.AddError("ErrCreate", "Error 1", null);
```

Note:
If the error is added to the System the module won't launch. In case of a single error it will be presented as a simple MessageBox; otherwise errors will be presented as a list.

If the user would like to inform about problems in his own way or skip the information part it can be obtained by setting the ShowErrorsDialog flag to false:

```
System.SystemBase.Errors.AddError("", "error", null);
this.ExtensionContext.Control.ShowErrorsDialog = false;
```

The same behavior can be obtained by calling:
```
this.SetCreateFail();
```

The PyramidGenerator module has to fill the Data with available Families from the Revit project. If the list of read element is empty the module won't launch. Because the described step is Revit dependent all actions will be taken in the ExtensionRef object.

4.2.1. Add a dictionary to the ExtensionRevit class. This dictionary will contain original FamilySymbols read from the project:

```
Dictionary<string, FamilySymbol> Families;
```

4.2.2. Add a LevelReference to store the level for pyramid generation:

```
Level LevelReference;
```

4.2.3. Fill the OnCreate method (ExtensionRevit):

```csharp
public override void OnCreate()
{
    FilteredElementCollector collector = new
    FilteredElementCollector(ThisExtension.Revit.ActiveDocument );

    IList<ElementFilter> filterList = new List<ElementFilter>();
    filterList.Add(new
    ElementCategoryFilter(BuiltInCategory.OST_StructuralFraming));
    filterList.Add(new ElementClassFilter(typeof(FamilySymbol)));
    LogicalAndFilter filter = new LogicalAndFilter(filterList);

    IList<Element> elements =
    collector.WherePasses(filter).ToElements();

    if (elements == null || elements.Count == 0)
        {
        ThisExtension.System.SystemBase.Errors.AddError("Error1",
        "No families were detected in the project", null);
    }
    else
    {
        Families = new Dictionary<string, FamilySymbol>();

        foreach (Element el in elements)
        {
            FamilySymbol familySymbol = el as FamilySymbol;

            string familyName = familySymbol.Family.Name + ": " +
            familySymbol.Name;
            Families.Add(familyName, familySymbol);

            ThisMainExtension.Data.AvailableFamilySymbols.Add(familyNam
            e);
        }
    }

    elements = collector.WherePasses(new
    ElementClassFilter(typeof(Level))).ToElements();
    if (elements == null || elements.Count == 0)
        {
        ThisExtension.System.SystemBase.Errors.AddError("Error2",
        "No level was detected", null);
    }
    else
    {
```

```
            LevelReference = elements[0] as Level;
        }
    }
```

It is not necessary to make any additional steps inside the main OnCreate method in the Extension class.

4.3. Implement the Extension::OnCreateDialogs method:

The OnCreateDialogs method should initialize all controls and dialogs in the module. They are context independent (RevitAPI objects are not used) and can be initialized directly in the Extension class (the OnCreateDialogs method in ExtensionRevit may stay empty in this case).

```
public override void OnCreateDialogs()
{
    base.OnCreateDialogs();

    NoteControlRef = new PyramidNoteControl(this);
    ParameterControlRef = new PyramidParametersControl(this);

    if (ExtensionRef != null)
        ExtensionRef.OnCreateDialogs();
}
```

4.4. Implement the Extension::OnCreateLayout method:

The OnCreateLayout method allows the user to arrange the appearance of the module. There are two main steps to make:

4.4.1. Define the list of items on the dialog:

The structure of the layout is determined by the ConstOptions. In case of PyramidGenerator it should contain:

```
Layout.ConstOptions = (long)REXUI.SetupOptions.HSplitFixed
                    | (long)REXUI.SetupOptions.VSplitFixed
                    | (long)REXUI.SetupOptions.TabDialog
                    | (long)REXUI.SetupOptions.List
                    | (long)REXUI.SetupOptions.FormFixed;
```

This part was generated by the wizard and can be modified by the user in case of required changes.

4.4.2. Register controls as layouts:

```
Layout.AddLayout(new REXLayoutItem(REXLayoutItem.LayoutType.Layout,
"Parameters", "", "Parameters", (long)0,ParameterControlRef, null,
parametersImage));
Layout.AddLayout(new REXLayoutItem(REXLayoutItem.LayoutType.Layout,
"Report", "", "Report", (long)0,NoteControlRef, null, noteImage));
```

Remark: parametersImage and noteImage have to be prepared and added to Resources/Other/Images (they can be taken from the PyramidGenerator sample provided with the document). They may be loaded as following:

```
BitmapImage logoImage =
REXLibrary.GetResourceImage(GetType().Assembly,
"Resources/Other/Images/REX_logo.png");
BitmapImage parametersImage =
REXLibrary.GetResourceImage(GetType().Assembly,
"Resources/Other/Images/ICON_Parameters.png");
BitmapImage noteImage =
REXLibrary.GetResourceImage(GetType().Assembly,
"Resources/Other/Images/ICON_Note.png");
```

Note:

ConstOptions:
The ConstOptions enumerator offers a wide range of possibilities:

```
public enum SetupOptions
{
    None,
    List,
    TabDialog,
    TabView,
    TabNote,
    DialogBottom,
    Status,
    Menu,
    ToolBar,
    ToolBarInner,
    ToolMenu,
    ActivateControl,
    FormFixed,
    VSplitFixed,
    HSplitFixed,
    DisableDocking,
    ToolbarIncrementForm,
    AutoSizeForm,
}
```
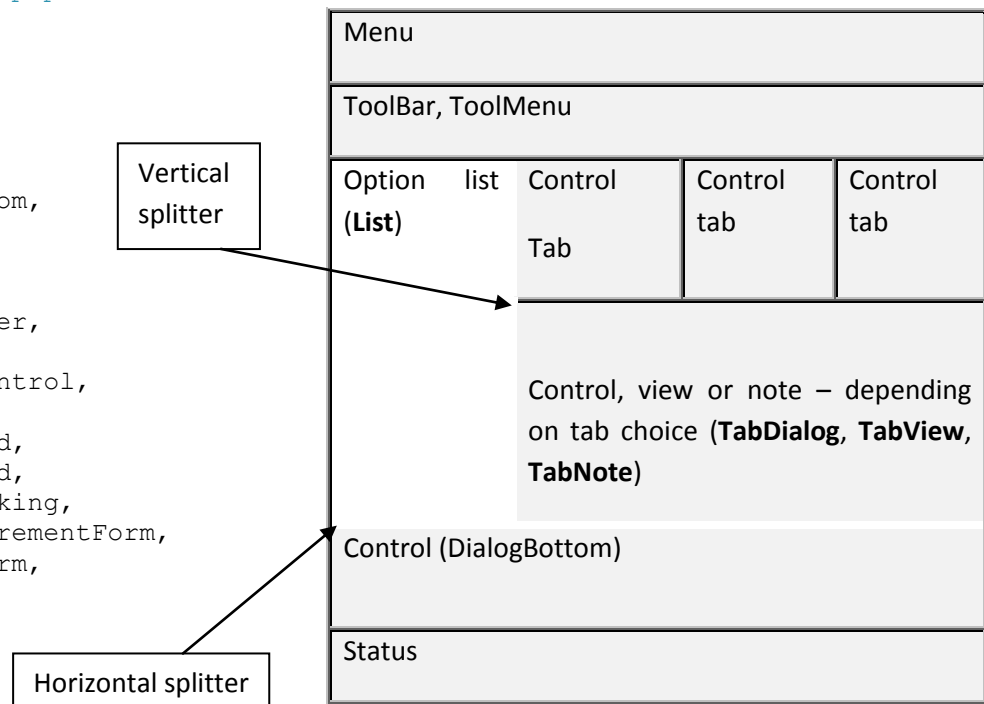

Groups:
In case of Layouts it is possible to group them in specified categories. In this case the new category has to be added and define the layout with the appropriate name of the group:

```
Layout.AddLayout(new  REXLayoutItem(REXLayoutItem.LayoutType.Group,  "Group",
"", Resources.Strings.Texts.LIST_Data, 0, null, null));
```

```
Layout.AddLayout(new   REXLayoutItem(REXLayoutItem.LayoutType.Layout,   "L1",
"Group", "Layout 1", 0, SubControlRef, null, 0));
```

Group

Layout 1

There are also predefined schemes which combines two steps mentioned above for instance:
```
Layout.AddLayout_Dialog(…)
Layout.AddLayout_Dialog_Menu(…)
```

4.5. Implement the Extension::OnSetDialog method:

The OnSetDialog method should initialize data in controls (based on the Data structure). In the current example there were prepared SetDialog methods in each layout control which may be used here:

```
public override void OnSetDialogs()
{
    base.OnSetDialogs();

    NoteControlRef.SetDialog();
    ParameterControlRef.SetDialog();

    Layout.SelectLayout("Parameters");

    if (ExtensionRef != null)
        ExtensionRef.OnSetDialogs();
}
```

Additionally the Parameters layout is set as the selected one.

4.6. Implement the Extension::OnActivateLayout method:

The OnActivateLayout method is called when the selected layout changes. In the current example it will be used for refreshing the note. Before the note regeneration it is obligatory to update the parameters in the Data structure (which may be done by the SetData method prepared in the PyramidParametersControl). Note that this way is not optimal and it doesn't monitor the modification status:

```
public  override  void  OnActivateLayout(REXLayoutItem  LItem,  bool
Activate)
{
    base.OnActivateLayout(LItem, Activate);

    ParameterControlRef.SetData();
    NoteControlRef.SetDialog();

    if (ExtensionRef != null)
        ExtensionRef.OnActivateLayout(LItem, Activate);
}
```

4.7. Implement the Extension::OnSetData method:

The OnSetData method is run at the beginning of executing the dialog (after OK). In this method all parameters in the Data structure should be updated with parameters set in the control. In the current example the SetData method in the PyramidParametersControl was created for this purpose:

```
public override void OnSetData()
{
    base.OnSetData();

    ParameterControlRef.SetData();

    if(string.IsNullOrEmpty(Data.FamilySymbol))
            System.SystemBase.Errors.AddError("Error","The family
            wasn't set", null);

    if (ExtensionRef != null)
        ExtensionRef.OnSetData();
}
```

Additionally the error is added to the system. Thanks to this the user will be informed about the problem after the module close.

4.8. Implement the Extension::OnCanRun method:

The OnCanRun method checks if parameters set by the user are correct. In the PyramidGenerator module the parameter which may cause problems is the selected family. It is possible to leave the selection empty and in this case the generation shouldn't be carried out (in case of editB and editH the correctness of the data is assured thanks to applied constraints and RememberCorrect flag which forces the control to back to the previous correct value if validation fails). To provide such functionality the OnCanRun method may be used. If it returns false the module stops its execution and as a result generation stops.

```
public override bool OnCanRun()
{
    bool ok = (!string.IsNullOrEmpty(Data.FamilySymbol) &&
    Data.AvailableFamilySymbols.Contains(Data.FamilySymbol));

    return ok;
}
```

Note:

It is possible to block the closing process of the dialog when some errors are detected (to give a chance to the user to enter a proper data). There are three ways to obtain it:

a. Implement OnCanClose method – if the method returns false the window won't be closed
b. Call the SetCanClose in the OnCanRun method – `SetCanClose(false);`
c. Call the RunQuestion method in the OnCanRun method and return its result. In this case it is

not necessary to check the correctness of the data inside the OnCanRun method. The error list (to which the error was added in the OnSetData method) will be checked automatically; the user will be informed about problems and ask about continuation.

```
public override bool OnCanRun()
{
    return
RunQuestion(Resources.Strings.Texts.MessageBox_CaptionQuestion,
Resources.Strings.Texts.MessageBox_ErrorQuestion);
}
```

It is good to notice that after showing the error list it is possible to react on errors selection changes. Such functionality is provided by the OnErrorSelected method in the Extension class.

4.9. Implement the ExtensionRevit::OnRun

The most important method which is called during the execution is the OnRun method. In this method the goal of the module should be achieved (e.g. element generation).

In the current example the goal of the module is to create the pyramid based on parameters set in the Data structure. The generation process is RevitAPI dependent and it will be carried out in the ExtensionRevit class.

Due to the assumption the pyramid will be generated in the 0,0,0 point:
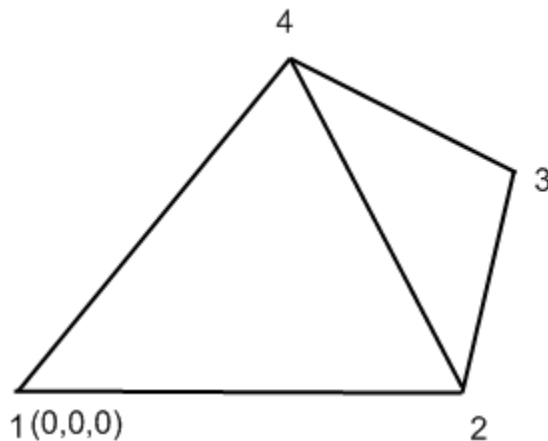

```
public override void OnRun()

{

double b =ThisExtension.Units.Interface(ThisMainExtension.Data.B,
EUnitType.Dimensions_StructureDim,1, REXInterfaceType.Revit);
double h = ThisExtension.Units.Interface(ThisMainExtension.Data.H,
EUnitType.Dimensions_StructureDim,1, REXInterfaceType.Revit);

double x1 = 0;
```

```
double y1 = 0;
double x2 = b;
double y2 = 0;
double x3 = b/2;
double y3 = b*Math.Sqrt(3)/2;
double x4 = b/2;
double y4 = y3/3;

            FamilySymbol familySymbol =
Families[ThisMainExtension.Data.FamilySymbol];

ThisExtension.Progress.Steps = 6;
ThisExtension.Progress.Header = "Generation";
ThisExtension.Progress.Text = "...";
ThisExtension.Progress.Show(ThisExtension.GetWindowForParent());

GenerateBeam(new XYZ(x1, y1,0),new XYZ(x2, y2,0), familySymbol);
GenerateBeam(new XYZ(x2, y2,0),new XYZ(x3, y3,0), familySymbol);
GenerateBeam(new XYZ(x1, y1,0),new XYZ(x3, y3,0), familySymbol);
GenerateBeam(new XYZ(x1, y1,0),new XYZ(x4, y4,h), familySymbol);
GenerateBeam(new XYZ(x2, y2,0),new XYZ(x4, y4,h), familySymbol);
GenerateBeam(new XYZ(x3, y3,0),new XYZ(x4, y4,h), familySymbol);

ThisExtension.Progress.Hide();
}

void GenerateBeam(XYZ start, XYZ end,FamilySymbol familySymbol)
{
ThisExtension.Progress.Step();
Line line = ThisExtension.Revit.Application().Create.NewLine(start,
end, true);
ThisExtension.Revit.ActiveDocument.Create.NewFamilyInstance(line,
familySymbol, LevelReference,
Autodesk.Revit.DB.Structure.StructuralType.Brace);
}
```

There are two things which are worth pointing in the code above.

o   The first is recalculating the parameters of the pyramid to Revit units (in data the current module stores parameters in base units).

```
double b = ThisExtension.Units.Interface(ThisMainExtension.Data.B,
EUnitType.Dimensions_StructureDim,1, REXInterfaceType.Revit);
```

o   The second is the usage of the progress bar (in the current example it is not perfectly seen because of the short process of generation).

```
            ThisExtension.Progress.Steps = 6;
            ThisExtension.Progress.Header = "Generation";
            ThisExtension.Progress.Text = "...";

ThisExtension.Progress.Show(ThisExtension.GetWindowForParent());
            ThisExtension.Progress.Step();
        …
            ThisExtension.Progress.Hide();
```

It is very important to set an appropriate window as a parent for the progress (as well as other dialogs shown on the top of the Extension module). There is a dedicated method which finds this parent automatically - `ThisExtension.GetWindowForParent());`

Note:

There is an additional option of calling a progress bar with the possibility of breaking the process (break button).

It can be found in the System object:

```
ThisExtension.System.ProgressTask
```

To attach to break button event use the ProgressEventHandler:

```
ThisExtension.System.ProgressTask.ProgressEventHandler += new
REXProgressEventHandler(ProgressTask_ProgressEventHandler);
```

4.10. Implement the Extension::OnClose method

The OnClose method is called when the Extension is aborting. It is recommended to release all resources here (the dedicated method was prepared before in the Note control).

```csharp
public override void OnClose()
      {
          base.OnClose();

          NoteControlRef.Release();

          if (ExtensionRef != null)
              ExtensionRef.OnClose();
      }
```

5. Step: Serialization

One of the benefits of using REXSDK is a serialization mechanism. It is possible to save (and load) data in:

- The text file
- The Revit element
- The Revit project

There are two main goals of the serialization in PyramidGenerator:

- Modification of an existing pyramid
- Saving parameters to the file

5.1. Extending the Data

The access point to the serialization system is the Data structure (presented before). In order to implement serialization two methods have to be overridden:

- OnSave – called during serialization
- OnLoad – called during deserialization

Note:

OnSave and OnLoad methods have two versions:

- Manual - the user has to implement serialization saving and loading data one by one:

```csharp
protected override bool OnSave(ref BinaryWriter Data)
{
    Data.Write(H);
    return true;
}
protected override bool OnLoad(ref BinaryReader Data)
{
    H = Data.ReadDouble();
    return true;
}
```

- .NET – the .NET serialization mechanism is used. There is a dedicated class in the Data.cs – DataSerializable which has to be implemented.

The choice of the method is up to the user. The manual option may need a little bit more work but it is fast and the result stream is not to large. The .NET mode is more automatic (there may be used objects which implements .NET serialization) but the result stream is larger which may be important while saving a lot of data in hundreds of elements inside Revit.

Additionally there is a VersionCurrent number which should be modified every time the serialization terms changes (new parameters, parameters type modification). In this case it is important to check the loaded version:

```csharp
protected override bool OnLoad(ref BinaryReader Data)
{
```

```
        H = Data.ReadDouble();
        B = Data.ReadDouble();

        if (VersionLoaded >= 1)
        {
        }

        return true;
    }
```

Steps:

5.1.1. Before implementation of the serialization the Data structure has to be extended with the list of elements ids:

```
public List<int> ElementIds { get; set; }
```

5.1.2. Initialize ElementIds in the OnSetDefaults method:
```
ElementIds = new List<int>();
```

5.1.3. Implement Data::OnSave and Data::OnLoad (manual version):

```
protected override bool OnSave(ref BinaryWriter Data)
{
        Data.Write(H);
        Data.Write(B);
        Data.Write(FamilySymbol);

        if (Mode != DataMode.ModeFile)
        {
            Data.Write(ElementIds.Count);

            for (int i = 0; i < ElementIds.Count; i++)
                Data.Write(ElementIds[i]);
        }

        return true;
}

protected override bool OnLoad(ref BinaryReader Data)
{
    H = Data.ReadDouble();
    B = Data.ReadDouble();
    FamilySymbol = Data.ReadString();

    if (Mode != DataMode.ModeFile)
    {
        ElementIds = new List<int>();
        int count = Data.ReadInt32();

        for (int i = 0; i < count; i++)
            ElementIds.Add(Data.ReadInt32());
```

```
        }

        return true;
    }
```

It is important to remember that in case of the file it is not necessary to save ids of elements because the most important part concerns parameters (in fact after loading a file the current list of elements cannot be changed)

5.2. Saving data to the file

The SDK provides ready solution for saving the data in rxd file.

Note:

TheExtension provides a group of methods responsible for serialization to file and string which may be called in any place of the code:

```
ThisExtension.System.SaveToFile
ThisExtension.System.LoadFromFile
ThisExtension.System.SaveToString
ThisExtension.System.LoadFromString
ThisExtension.System.SaveToStringEx
ThisExtension.System.LoadFromStringEx
```

5.2.1. Make the toolbar of the module visible by adding the ToolMenu flag in the OnCreateLayout method of the Extension class (in order to have access to the File menu):

```
Layout.ConstOptions |= (long)REXUI.SetupOptions.ToolMenu;
```

Additionally the ToolMenu may be configured using CommandOptions:

```
Layout.CommandsOptions = (long)REXUI.CommandOptions.ToolMenuFile
                       | (long)REXUI.CommandOptions.ToolMenuFileOpen
                       | (long)REXUI.CommandOptions.ToolMenuFileSave
                       | (long)REXUI.CommandOptions.ToolMenuFileSaveAs
                       | (long)REXUI.CommandOptions.ToolMenuHelp
                       | (long)REXUI.CommandOptions.ToolMenuHelpAbout;
```
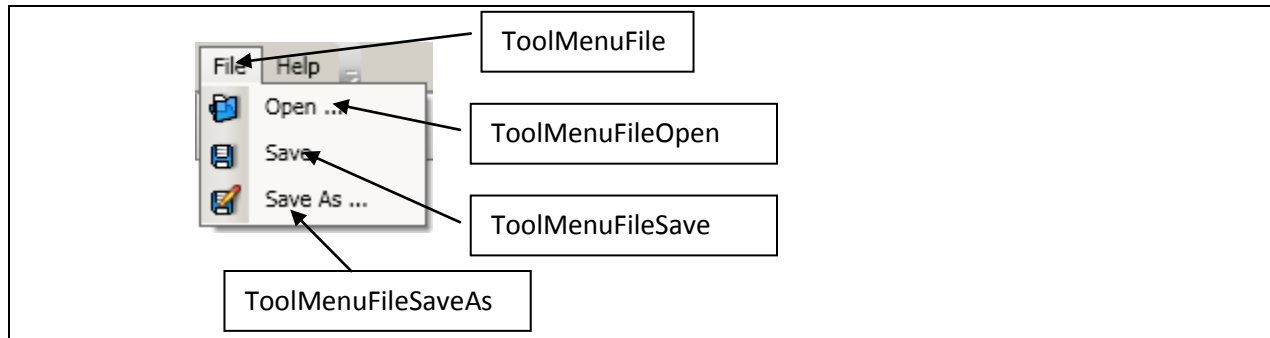

Note:

The CommandOptions allows to define which items should be visible on:
- Menu
- ToolMenu
- Toolbar
  Remember: In order to see this changes the main element (e.g. Menu) has to be added to ConstOptions in the Layout object.

5.2.2. Update the parameters in the Data structure in the OnDataSave method which is called before saving the file:

```
public override bool OnDataSave()
{
        ParameterControlRef.SetData();
        return true;
}
```

5.2.3. Update the dialog controls after loading the file in the OnDataLoaded method:

```
public override void OnDataLoaded(bool Result)
{
      if (Result)
      {
          ParameterControlRef.SetDialog();
          NoteControlRef.SetDialog();
      }
}
```

Note:

If there is any additional action needed after saving the file use the OnDataSaved method.
If there is any additional action needed before opening the file use the OnDataLoad method.

5.3. Saving data in a Revit element

As it was mentioned before the module's data may be serialized in a Revit element. There are dedicated methods to obtain it:

- To save data: `ThisExtension.Revit.Parameters.SaveToHost(element);`
- To load data: `ThisExtension.Revit.Parameters.LoadFromHost(element);`

In case of the PyramidGenerator there are six beams generated. In order to simplify the example the data will be saved to each of them.

Note:

The other more optimal implementation is to select one of the beams as a main one and store the data

in it. Other beams in this case should point this beam as a data container. This can be done:

- Manually (just change the Data parameters which are serialized in case of the main beam):

```csharp
protected override bool OnLoad(ref BinaryReader Data)
{
    MainBeam = Data.ReadBoolean();
    MainBeamId = Data.ReadInt32();

    if (MainBeam)
    {
        //read parameters
    }
}

protected override bool OnSave(ref BinaryWriter Data)
{
    Data.Write(MainBeam);
    Data.Write(MainBeamId);

    if (MainBeam)
    {
        //write parameters
    }
}
```

- Using the mechanism provided with the REXSDK, which allows the user to store the id of one element in the other element (it treats the first element as its virtual host).
  - To save the id of the host:

```csharp
ThisExtension.Revit.Parameters.SetHostId(host,elem);
```

  - To get the id of the host:

```csharp
ThisExtension.Revit.Parameters.GetHostId(elem);
```

Example:

```csharp
int hostId = ThisExtension.Revit.Parameters.GetHostId(el);
if (hostId != el.Id.IntegerValue)
{
    ElementId ei = new ElementId(hostId);
    Element hostElement = ActiveUIDocument.Document.get_Element(ei);
    ThisExtension.Revit.Parameters.LoadFromHost(hostElement);
}
else
ThisExtension.Revit.Parameters.LoadFromHost(el);
```

5.3.1. Load the Data based on the current selection in the OnCreate method of the ExtensionRevit class:

```csharp
foreach (Element el in
ThisExtension.Revit.ActiveUIDocument.Selection.Elements)
{
    if (ThisExtension.Revit.Parameters.LoadFromHost(el))
    {
        break;
    }
}
```

5.3.2. Remove all modified beams before generation in the OnRun method of the ExtensionRevit class:

```csharp
void RemoveAllGeneratedBeams()
{
        foreach (int id in ThisMainExtension.Data.ElementIds)
        {
                Element el =
                ThisExtension.Revit.ActiveDocument.get_Element(new
                ElementId(id));
                if (el != null)
                        ThisExtension.Revit.ActiveDocument.Delete(el);
        }
}
```

5.3.3. Update the list of ElementIds in the GenerateBeam method of the ExtensionRevit class:

```csharp
void GenerateBeam(XYZ start, XYZ end,FamilySymbol familySymbol)
{
        ThisExtension.Progress.Step();

        Line line =
        ThisExtension.Revit.Application().Create.NewLine(start,
        end, true);

        FamilyInstance familyInstance =
        ThisExtension.Revit.ActiveDocument.Create.NewFamilyInstance
        (line, familySymbol, LevelReference,
        Autodesk.Revit.DB.Structure.StructuralType.Brace);

        if (familyInstance != null)
                ThisMainExtension.Data.ElementIds.Add(familyInstance.
                Id.IntegerValue);
}
```

5.3.4. Save the data to all generated beams at the end of the OnRun method of the ExtensionRevit class:

```csharp
void SaveDataToAllBeams()
{
        foreach (int id in ThisMainExtension.Data.ElementIds)
        {
          Element el =
          ThisExtension.Revit.ActiveDocument.get_Element(new
          ElementId(id));
           if (el != null)
              ThisExtension.Revit.Parameters.SaveToHost(el);
        }
}
```

6. Step: Running PyramidGenerator

Now when the project is ready and compilation succeeded the PyramidGenerator is ready to use. There are several ways to launch the module:

- Launching the module using the start project generated by the wizard:
  It is possible to launch the module using the start project generated by the wizard (if this option was chosen). This project should be set as a StartUp Project in the Visual Studio 2010. The user starts debugging and the start project automatically launches the module. As it was mentioned before there should be a proper organization of the code applied (RevitAPI part has to be skipped in this mode – see Extension and ExtensionRevit classes). This mode is recommended in the first stage of Extension creation when the User Interface has to be prepared.

- Using Revit addin generated by the wizard (it is launched by Revit through the DirectAccess class):

```xml
<?xml version="1.0" encoding="utf-8"?>
<RevitAddIns>
  <AddIn Type="Command">
    <Assembly>c:\.....\Debug\PyramidGenerator.dll</Assembly>
    <ClientId>a42b9641-c7f2-422d-9215-90b3d69535ec</ClientId>
    <FullClassName>REX.PyramidGenerator.DirectRevitAccess</FullClassName>
    <Text>PyramidGenerator</Text>
    <Description>PyramidGenerator generates pyramids</Description>
    <VendorId>SDK Vendor</VendorId>
    <VendorDescription>SDK</VendorDescription>
  </AddIn>
</RevitAddIns>
```
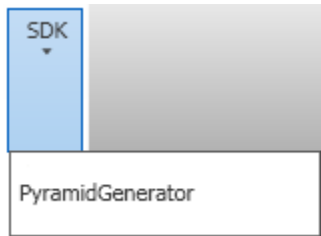
In this case the extension will be available in Revit External Tools:
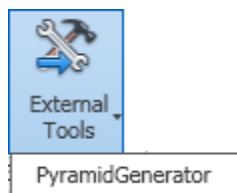
- Create the PyramidGenerator folder in "… \Extensions 2012\Modules\External" and copy the content of Debug\bin folder to it:
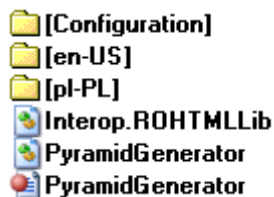
In this case the Extension will be available on the Extension ribbon in the SDK category:
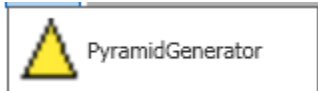
This way will work if the Extensions integration option is chosen in the wizard.

Note:

In the bin folder there are following folders:

Configuration – contains all language independent file for the current module:

- REX.bmp – the icon of the module which may be showed in the Revit ribbon e.g.:

- REX_Revit.xml – contains all information about the way of launching the module in Revit (access class, name, description etc.)
- Settings.xml – contains setting of the module (especially visibility in different contexts: Structure, MEP etc.)
- en-US, fr-FR,… – contains settings which are language dependent (e.g. REX_Revit with translated names) and resources files

---

- Launch the module from the user's Revit ExternalApplication ribbon. This method assumes that there is some ExternalApplication (added to ribbon on Revit start) which loads the Extension through the DirectAccess class. For instance:

```
public Result OnStartup(UIControlledApplication application)
{
        RibbonPanel ribbonSamplePanel =
        application.CreateRibbonPanel("Extension SDK");
        SplitButtonData splitButtonData = new
        SplitButtonData("ExtensionSDK", "SDK");
        SplitButton splitButton =
        ribbonSamplePanel.AddItem(splitButtonData) as SplitButton;

        PushButton pushButton = splitButton.AddPushButton(new
        PushButtonData("PyramidGenerator", "Pyramid", "..",
        "REX.PyramidGenerator.DirectRevitAccess"));
        return Result.Succeeded;
 }
```

- Launch the module from the other Extension:

```
REXApplicationInstance AppInst = new REXApplicationInstance();
string modulePath = @"..\PyramidGenerator.dll";
```

```
if (AppInst.LoadExtension(modulePath,
"REX.PyramidGenerator.Application"))
{
    REXContext context = new REXContext(ThisExtension.Context, false);
    if (AppInst.Extension.Create(ref context))
    {
        AppInst.Extension.Show();
    }
}
```


Note:

It is possible to load Extension using the LoadStandardExtension method:
`AppInst = REXAccessibility.LoadStandardExtension("PyramidGenerator");`

In this case however the module has to be located in REX system folders (e.g. External) or the specific key has to be added to Windows registries which is done automatically when the REX installation is run (this installation which is prepared by the SDK wizard for the current module – additional options in the wizard).

Result
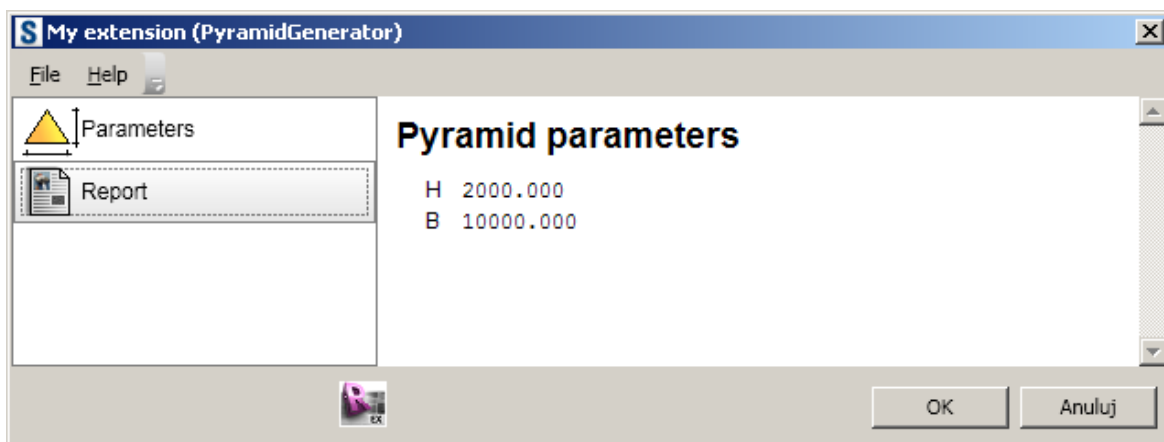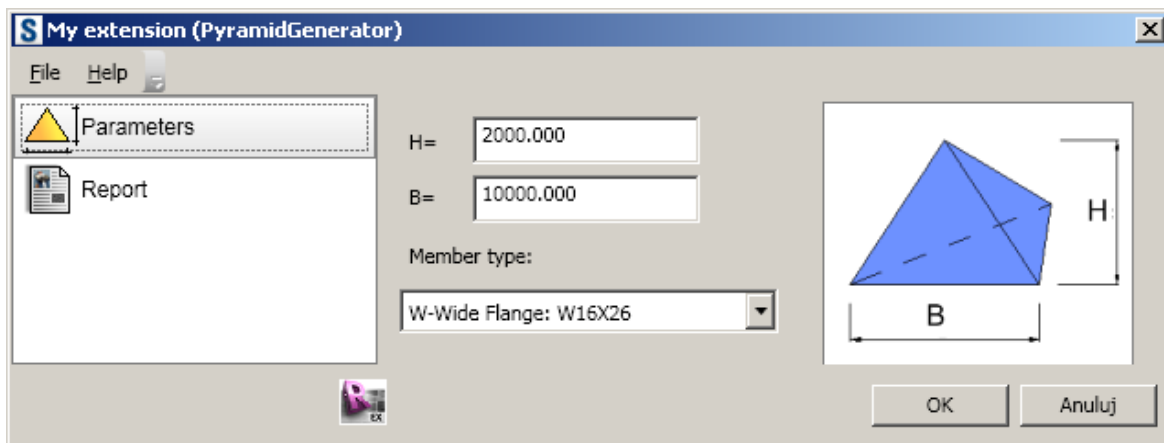
Revit structure:
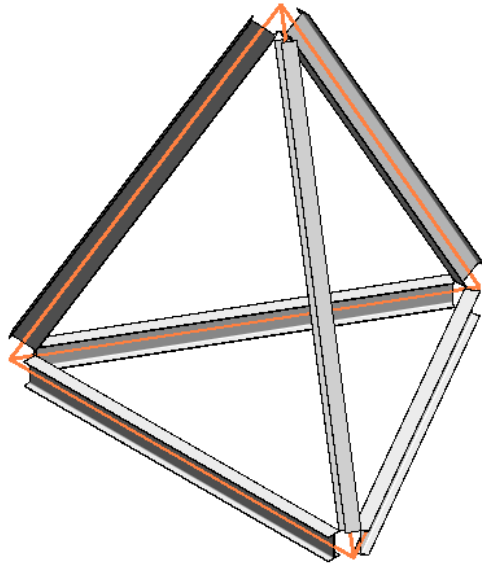
7. Step: Exchanging data with external modules

As it was said in the step 6 the Extension module may be run by the other Extension module. Except launching it is also possible to communicate with it. It can be done after module creation:

```
if (AppInst.LoadExtension(modulePath, "REX.PyramidGenerator.Application"))
{
    REXContext context = new REXContext(ThisExtension.Context, false);
    if (AppInst.Extension.Create(ref context))
    {
        //----->Communicate with the module here

        AppInst.Extension.Show();
    }
}
```

> **Note:**
>
> The Extension may be shown as:
> - Modal dialog
> - Dialog
> - Control
>
> It is controlled by the context and its ControlMode:
>
> ```
> REXContext context = new REXContext(ThisExtension.Context, false);
> context.Control.ControlMode = REXControlMode.Control;
> if (AppInst.Extension.Create(ref context))
> {
>     object control = AppInst.Extension.Control();
>     //this control may be embedded in the other dialog
> }
> ```

It is also possible to give up showing the module and run it in a "silent mode" which allows using module as a kind of a server providing required functionality (in case of PyramidGenerator it is pyramid generation). In the current example it will be possible to generate pyramid in any Extension module by using PyramidGenerator. It can be done in two ways:

7.1. Implement the Extension::OnCommand method

The OnCommand method was created in order to provide communication between the Extension and the external world. All necessary information is passed through the REXCommand object. In the class there is CommandObject member which can store any structure. In order to simplify it, there will be a simple dictionary with the name of the parameter as a key and an object as a value.

```csharp
public override object OnCommand(ref REXCommand CommandStruct)
{
    Dictionary<string, object> dictionary = CommandStruct.CommandObject
as Dictionary<string, object>;

    if (dictionary != null)
    {
        try
        {
            Data.H = Convert.ToDouble(dictionary["H"]);
            Data.B = Convert.ToDouble(dictionary["B"]);
            Data.FamilySymbol = dictionary["FamilySymbol"].ToString();
            ExtensionRef.OnRun();
            return true;
        }
        catch
        {
            return false;
        }
    }
    else
    {
        return base.OnCommand(ref CommandStruct);
    }
}
```

The code in a client code:

```csharp
REXApplicationInstance AppInst = new REXApplicationInstance();
string modulePath = @"..\PyramidGenerator.dll";

if (AppInst.LoadExtension(modulePath,
"REX.PyramidGenerator.Application"))
{
    REXContext context = new REXContext(ThisExtension.Context, false);
    if (AppInst.Extension.Create(ref context))
    {
        Dictionary<string, object> dictionary = new Dictionary<string,
object>();
        dictionary.Add("H", 20);
        dictionary.Add("B", 30);
        dictionary.Add("FamilySymbol", "W-Wide Flange: W12X26");
        REXCommand command = new REXCommand(REXCommandType.Command,
dictionary);

        AppInst.Extension.Command(ref command);
        AppInst.Extension.Close();
    }
}
```

Note:

The REXCommand class provides possibility to manage the whole extension. There are several predefined commands which may be used by the user:

```csharp
public enum REXCommandType
{
    Apply = 1,
    OK = 2,
    Run = 3,
    Close = 4,
    Help = 5,
    About = 6,
    Command = 7,
    Query = 8,
    Status = 20,
}
```

7.2. Implement the Extension::GetAPI method

The other method is to provide communication with the module by exposing the dedicated API:

7.2.1. Prepare the Pyramid Generator interface.

To simplify the example the interface will be prepared in the same assembly:

```csharp
public interface IPyramidGenerator
{
    double H { get; set; }
    double B { get; set; }
    string FamilySymbol { get; set; }
    void Generate();
}
```

7.2.2. Implement the interface in the Extension class:

```csharp
class Extension : REXExtension, IPyramidGenerator

public double H
{
    get{return Data.H;}
    set{Data.H = value;}
}
public double B
{
    get{return Data.B;}
    set{Data.B = value;}
}
public string FamilySymbol
{
    get{return Data.FamilySymbol;}
    set{Data.FamilySymbol = value;}
}
```

```csharp
public void Generate()
{
    if (ExtensionRef != null)
        ExtensionRef.OnRun();
}
```

7.2.3. Return the current instance of Extension in the GetAPI method:

```csharp
public override Object GetAPI()
{
    return this;
}
```

7.2.4. Add the reference to the PyramidGenerator.dll in the client application and generate the pyramid through the API interface:

```csharp
REXApplicationInstance AppInst = new REXApplicationInstance();
string modulePath = @"..\PyramidGenerator.dll";

if (AppInst.LoadExtension(modulePath,
"REX.PyramidGenerator.Application"))
{
    REXContext context = new REXContext(ThisExtension.Context,
    false);

    if (AppInst.Extension.Create(ref context))
    {
        REXCommand command = new REXCommand(REXCommandType.Query,
        REXConst.RunContext.QueryAPIInterface);
         object api = AppInst.Extension.Command(ref command);
        REX.PyramidGenerator.IPyramidGenerator apiPyramidGenerator
        = api as REX.PyramidGenerator.IPyramidGenerator;

        apiPyramidGenerator.H = 2;
        apiPyramidGenerator.B = 3;
        apiPyramidGenerator.FamilySymbol ="W-Wide Flange: W12X26";
        apiPyramidGenerator.Generate();
        AppInst.Extension.Close();
    }
}
```

Note:

In the example presented above there is no validation of Data (that has to be done obligatory) as well as checking the UserInterface context (there is a ProgressBar which shouldn't be visible).

It is also good to set the information about NOUI mode in the context in the clients code:

```csharp
if (context.Extension.References.ContainsKey(REXConst.ENG_SystemRefNoUI))
    context.Extension.References[REXConst.ENG_SystemRefNoUI] = true;
else
    context.Extension.References.Add(REXConst.ENG_SystemRefNoUI, true);
```

Thanks to this the part responsible for UI won't be taken to consideration.