TABLE OF CONTENTS

1. WHAT IS REVIT EXTENSIONS SDK

Extension SDK is a development environment for Rapid Application Development purposes that helps to create, deploy and activate add-ins based on Revit Extension technology.

The core part of the Revit Extensions SDK is implemented as a form of Microsoft Visual Studio C# templates. Using the template provided, you can quickly build and deploy an add-in that has a similar look & feel to Autodesk Revit Extensions.

The Extensions SDK take advantage of the Extensions Framework installed with Revit 2012 including dialog creation feature, advance controls, a units engine and the content generator component.

Extensions are some external commands and could be used as all Revit External Command and provide a pointer to the Revit API.

The Extensions SDK is composed of:

- Visual Studio Templates (C#)
    - UI definition
    - Interactions
    - Localization support
    - Deployment
    - Microsoft Installation project
- Documentation
    - Getting Started
    - Design guidelines
    - Samples

2. INSTALLATION AND CONFIGURATION

2.1. System requirements

- Visual Studio 2010
- .NET Framework 3.5 or 4.0
- Revit Architecture 2012, Revit Structure 2012 or/and Revit MEP 2012 installed

2.2. Content

Folder	File	Description
Documentation	Getting Started for Extensions SDK.pdf	Getting started document contains information about the basics of Extensions SDK and how to create a first Extension.
	User Manual for Extensions SDK.pdf	User manual document contains detailed information about Extensions framework.
	Design Guidelines for Extensions SDK.pdf	Design Guidelines document describes how to design Extensions Graphical User Interface.

	StepByStep.pdf	A step by step tutorial to explain how to create an extension
	ExtensionFrameworkAPI2012.chm	File containing documentation on the Extensions SDK framework.

Folder	File	Description
	Unit	Sample to learn how to take advantage of the Extension Unit Engine.
	FrameGenerator	Sample to learn how to interact with Revit.
	Serialization	Sample to learn how to use Extensions serialization.
	ElementReportHTML	Sample to learn how to use the HTML report.
	ContentGeneratorWPF	Sample to learn how to use the content generator component.
	ExtensionRevitLauncher	Sample to learn how to connect your extensions to Revit Ribbon.
	PyramidGenerator	Sample describing step by step how to create your own extension.

Folder	File	Description
Visual Studio templates	Items	This directory contains files to copy on items template folder from Visual Studio.
	Project	This directory contains files to copy on items project folder from Visual Studio.

2.3. Configuration

To properly configure Visual Studio 2010, Extensions templates should be copied to dedicated folders

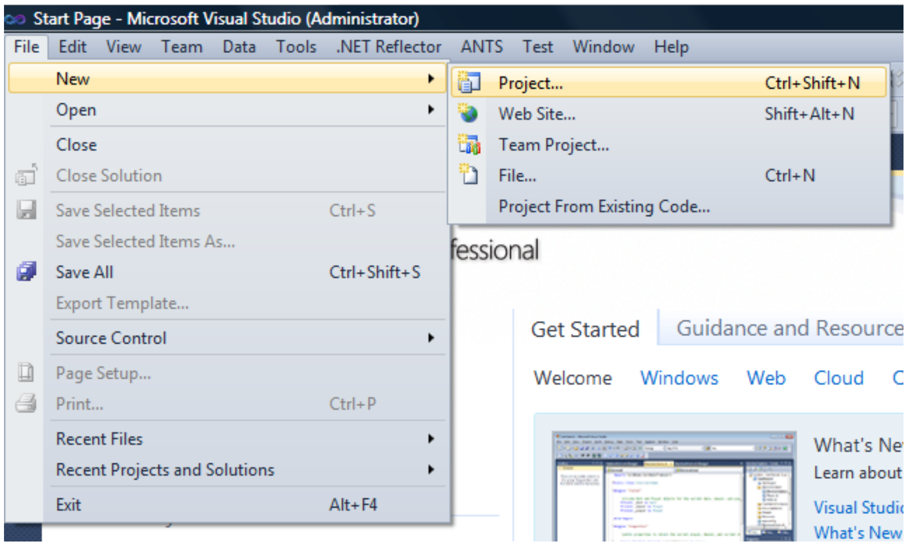
- The content of the folder "..\Software Development Kit\REX SDK\Visual Studio templates\Items\" should be copied to "C:\Users\<current user>\Documents\Visual Studio 2010\Templates\ItemTemplates\Visual C#\"

- The content of the folder "..\Software Development Kit\REX SDK\Visual Studio templates\Projects\" should be copied to "C:\Users\<current user>\Documents\Visual Studio 2010\Templates\ProjectTemplates\Visual C#

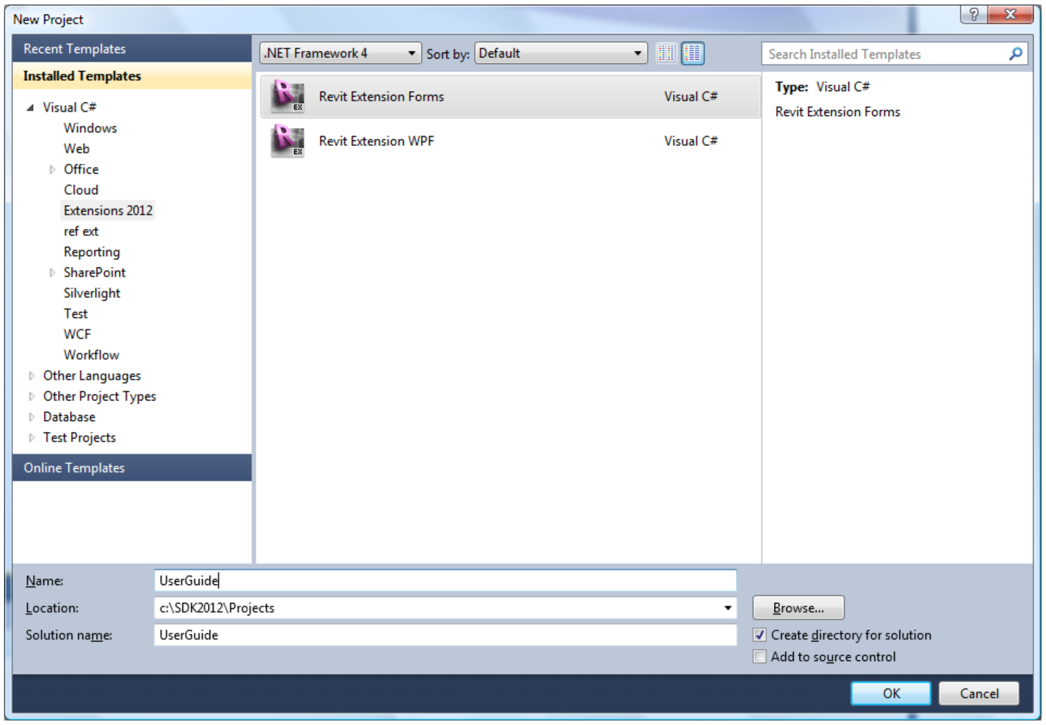
3. PROJECT CREATION

3.1. Visual Studio

After starting visual studio, you need to choose new project (**File / New / Project**).



Then this dialog will appear



Type: Visual C#
Revit Extension Forms

Name: UserGuide
Location: c:\SDK2012\Projects
Solution name: UserGuide
[✓] Create directory for solution
[ ] Add to source control

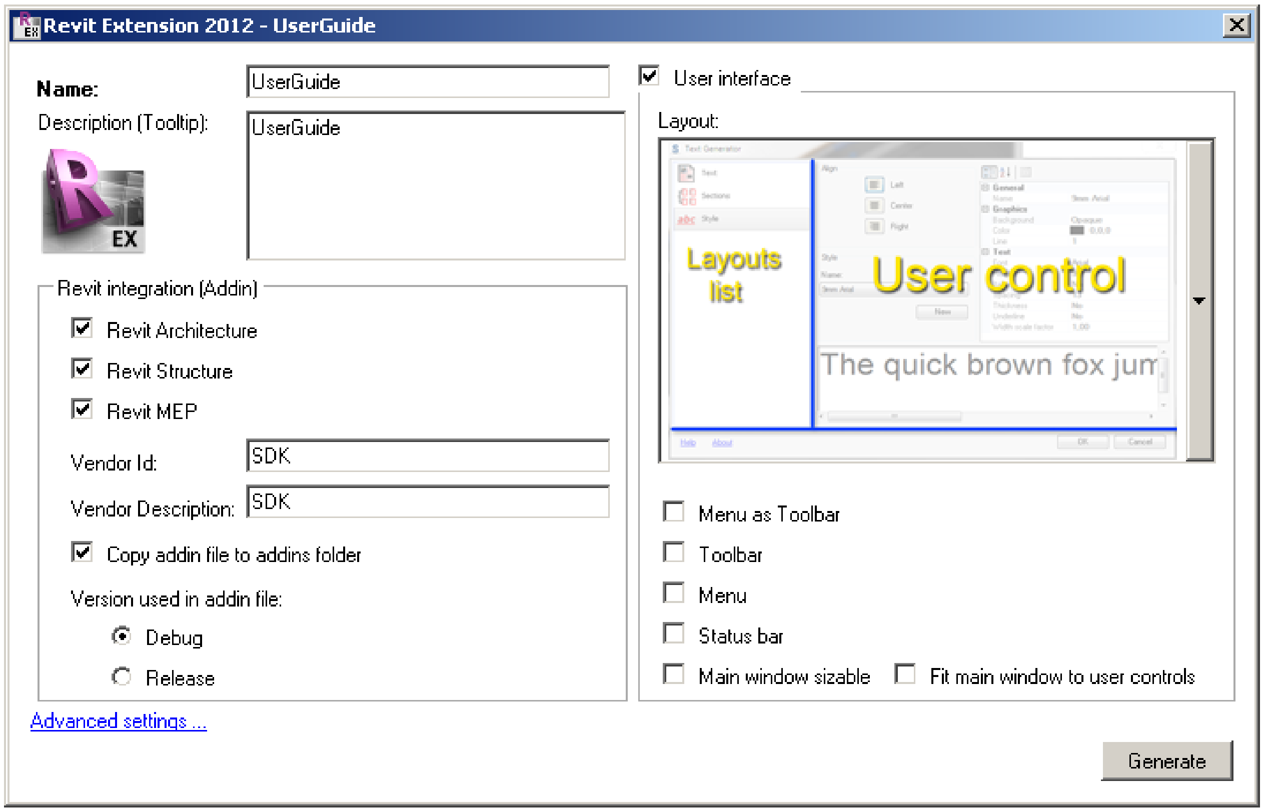
Currently 2 types of project are available:

- Revit Extensions form, to create new extension project based on Winforms technology
- Revit Extensions WPF, to create new extension project based on WPF technology

At this stage the name of the solution and project should be set.
The project name is an important reference and must be unique.

Why is it important to have a unique name?

- Compiled files are stored on a directory with the same name as the C# project.
- The name of the extension is used for the data serialization.


Revit Extension 2012 - UserGuide

Name: UserGuide

Description (Tooltip): UserGuide

Revit integration (Addin)

- [✓] Revit Architecture
- [✓] Revit Structure
- [✓] Revit MEP

Vendor Id: SDK

Vendor Description: SDK

- [✓] Copy addin file to addins folder

Version used in addin file: [✓ Debug] Release

[✓] User interface

Layout:

- [ ] Menu as Toolbar
- [ ] Toolbar
- [ ] Menu
- [ ] Status bar
- [ ] Main window sizable
- [ ] Fit main window to user controls

A set of option could be defined on this dialog to :

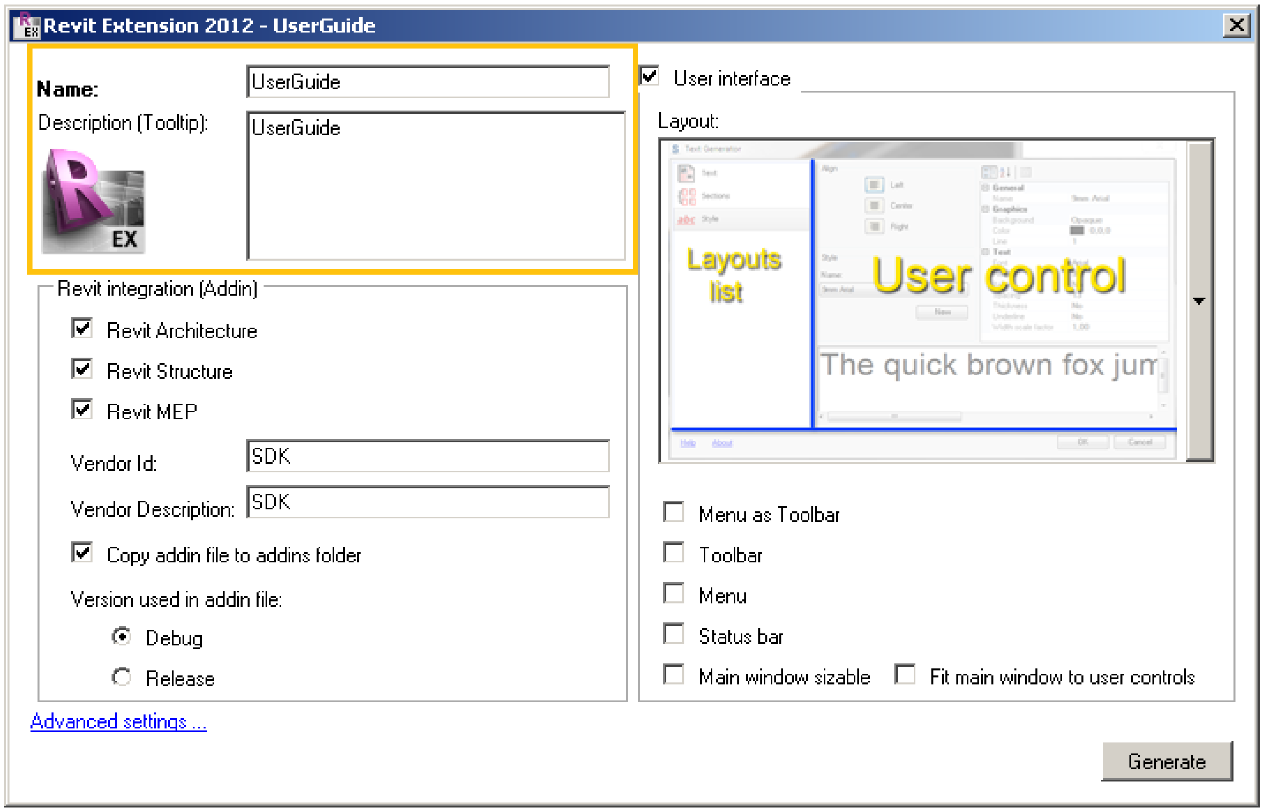
- Define the name of the extension
- Define the adding file used to start extension as an external command
- Configure the UI
- Define some advanced settings to integrate your extension with regular extensions

Advanced options are dedicated for the integration with Revit Extension package delivered to customer under subscription only.

4. MODULE DEVELOPMENT

Below will be described main areas. StepByStep documentation could be read in parallel to this document. This file is located on the PyramidGenerator sample directory.

4.1. Project settings



Revit Extension 2012 - UserGuide

Name: UserGuide

Description (Tooltip): UserGuide

Revit integration (Addin)

- [✓] Revit Architecture
- [✓] Revit Structure
- [✓] Revit MEP

Vendor Id: SDK

Vendor Description: SDK

- [✓] Copy addin file to addins folder

Version used in addin file:

[✓ Debug] Release

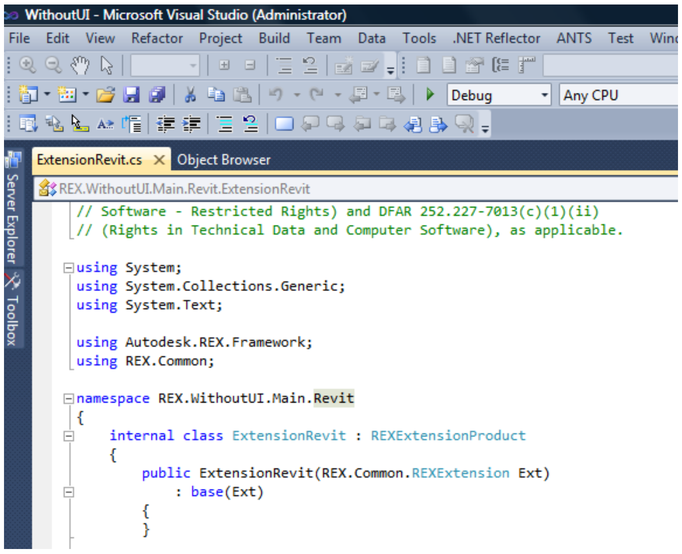
- [✓] User interface

Layout:

- [ ] Menu as Toolbar
- [ ] Toolbar
- [ ] Menu
- [ ] Status bar
- [ ] Main window sizable
- [ ] Fit main window to user controls

During project creation, most important project settings are predefined.

Assembly name, namespace and targeted .Net Framework are set.



WithoutUI - Microsoft Visual Studio (Administrator)
File Edit View Refactor Project Build Team Data Tools .NET Reflector ANTS Test Wind
Debug
Any CPU
ExtensionRevit.cs
Object Browser
Server Explorer
Toolbox
REX.WithoutUI.Main.Revit.ExtensionRevit

```
// Software - Restricted Rights) and DFAR 252.227-7013(c)(1)(ii)
// (Rights in Technical Data and Computer Software), as applicable.

using System;
using System.Collections.Generic;
using System.Text;

using Autodesk.REX.Framework;
using REX.Common;

namespace REX.WithoutUI.Main.Revit
{
    internal class ExtensionRevit : REXExtensionProduct
    {
        public ExtensionRevit(REX.Common.REXExtension Ext)
            : base(Ext)
        {
        }
```

Other settings as the build events as the build events as been set and could be defined by the developer using the Visual Studio common rules.

4.2. Project structure

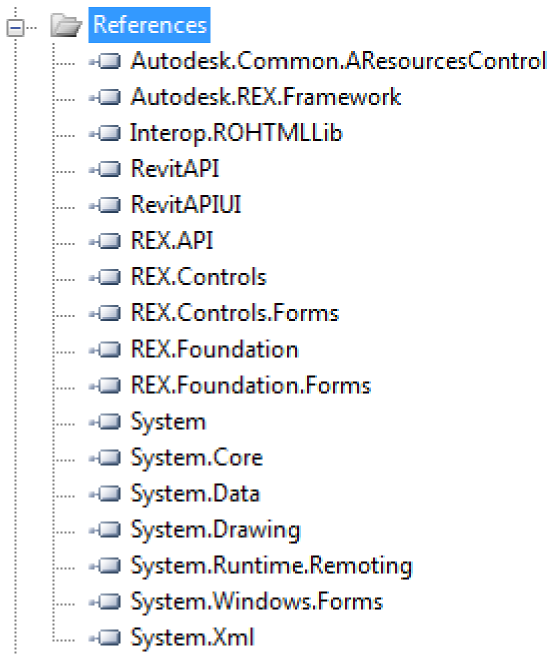
- **References** – Reference project.

  References are resolved automatically during the project creation.

- **Winforms project**

As default, References below are included:

- Revit API And Revit API UI
- REX references
- REX Forms references
- .NET libraries.

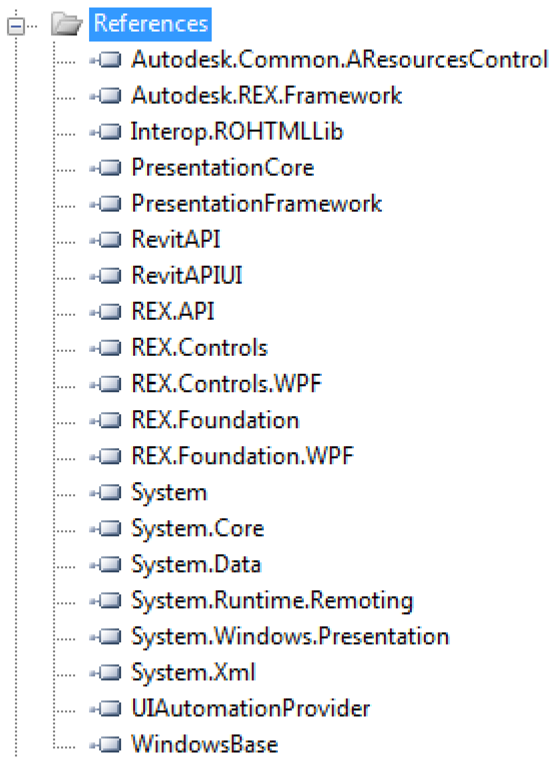


```
References
  Autodesk.Common.AResourcesControl
  Autodesk.REX.Framework
  Interop.ROHTMLLib
  RevitAPI
  RevitAPIUI
  REX.API
  REX.Controls
  REX.Controls.Forms
  REX.Foundation
  REX.Foundation.Forms
  System
  System.Core
  System.Data
  System.Drawing
  System.Runtime.Remoting
  System.Windows.Forms
  System.Xml
```

- **WPF project**

As default, References below are included:

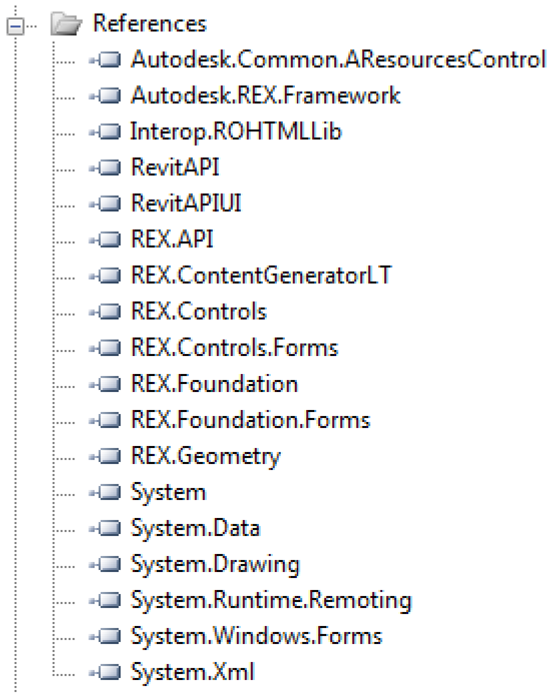
- Revit API And Revit API UI
- REX references
- REX WPF references
- .NET libraries.



```
References
  Autodesk.Common.AResourcesControl
  Autodesk.REX.Framework
  Interop.ROHTMLLib
  PresentationCore
  PresentationFramework
  RevitAPI
  RevitAPIUI
  REX.API
  REX.Controls
  REX.Controls.WPF
  REX.Foundation
  REX.Foundation.WPF
  System
  System.Core
  System.Data
  System.Runtime.Remoting
  System.Windows.Presentation
  System.Xml
  UIAutomationProvider
  WindowsBase
```

- **Content Generator and Geometry**

  Content Generator and Geometry components are not set automatically via the wizard. Developer should add these references by himself.



```
References
  Autodesk.Common.AResourcesControl
  Autodesk.REX.Framework
  Interop.ROHTMLLib
  RevitAPI
  RevitAPIUI
  REX.API
  REX.ContentGeneratorLT
  REX.Controls
  REX.Controls.Forms
  REX.Foundation
  REX.Foundation.Forms
  REX.Geometry
  System
  System.Data
  System.Drawing
  System.Runtime.Remoting
  System.Windows.Forms
  System.Xml
```

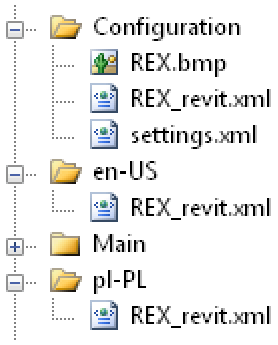
- **Configuration, en-US, pl-PL, and other languages** – On the directory you could find base configurations files and put yours.

The structure is this one:

- **Configuration** - to find settings.xml per default. It's the place to put additional xml configuration files.

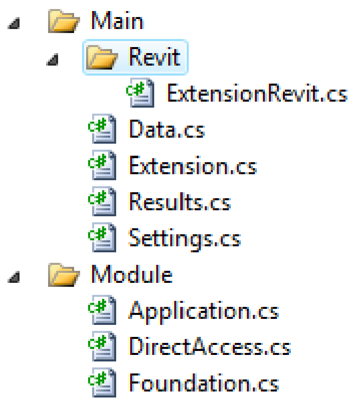
  These files are analyzed by the code on method (Main/Extension.cs/ExtensionSettings/OnParse (…)). Developer could define his own configuration settings for the new module and create particular code for dedicated action.

- **Directories for different languages** - you could find name of the REX_revit.xml file. This file is used only for the integration with Revit Extensions



```
Configuration
    REX.bmp
    REX_revit.xml
    settings.xml
en-US
    REX_revit.xml
Main
pl-PL
    REX_revit.xml
```

- **Main** – main sources



```
Main
    Revit
        ExtensionRevit.cs
    Data.cs
    Extension.cs
    Results.cs
    Settings.cs
Module
    Application.cs
    DirectAccess.cs
    Foundation.cs
```

In this directory, main source files should be located. This is the main location for the developer source code.

These files are created by template and their description is in the next paragraph.

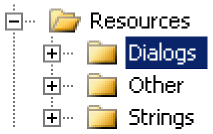
One directories additional folder is created - Revit - to store product dedicated classes and organize properly the project.

On module directory are the main system classes dedicated to a proper initialization.
DirectAccess is the entry point to launch an Extension as a Revit ExternalCommand. Developer could point this class as entry point for his own Ribbon starter – see ExtensionRevitLauncher sample.

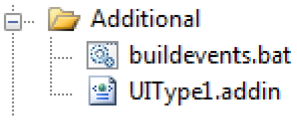
- **Resources** – set of dialogs, controls, bitmaps

Development rules are:

- **Dialogs** for dialogs and user control
- **Strings** – strings
- **Other** – bitmap and other resources



- **Additional** – set of files for configuration. Build events could be used by developer for a post build and the addin file to start the extension as an External Command



4.3. General assumptions

Classes and methods using API should be separated from other classes. Files using Revit API should be placed on Main\Revit directory.

Extension.cs is the main class to manage context and control functions called by other object.

5. EXTENSION FRAMEWORK

- `Main / Extension.cs` – contains classes:

`Extension` – class which should be extended with new user classes and handle all supported calls context. It's the main class of the module. However the best place to achieve changes and addition is on ExtensionRevit class.

- `Main/ Settings` – class manage module settings read from `Settings.xml` file.

- `Main` / **Data.cs** - contains class:

Data – holds all data used in dialogs and additional data which will be saved in the file, in the object, in application object. It applies also to data available and managed by user interface.

- `Main / Results.cs` - contains class

`Results` – optional class similar as Data, used to separate data and results.

- **`Module / Application.cs`** – contains class:

`Application` – control application lifecycle and ensure the communication between Revit and Revit Extensions.

- **`Module / DirectAccess.cs`** - contains class

`DirectAccess` – allows starting module itself. This class is used to call the extension from the Revit Ribbon

- **`Module /Foundation.cs`** - contains class

`Foundation` – allows starting module itself. This class is used for proper initialization.

- **`Revit /ExtensionRevit.cs`** - contains class

`ExtensionRevit` - class which should be extended with new user classes and handle Revit context. It's the main class of the module.

5.1. **Handling module work context**

5.1.1. Introduction

The extension Framework is designed to create some extensions working on the top of different Autodesk product. An Autodesk product means for the extension framework a context.

There are three types of context in module:

- Main context

It's connected with the main application starting the module.

- Module context

Used to module control. It also covers the information about start context
.

- Start context

Part of module context, logically separated and used to control the module, for example sending data from main application to another one, removing data connected with module and managing task in explicit and implicit mode.

For Revit Extensions SDK, the context is Revit.

5.1.2. Main context

`REXContext` class is responsible for main context. Object of this class is used to send information and data between hosting application and module.

On the level of **`REXExtension`** class, access to objects is provided by the context property or using **`ThisApplication.Context2.`**

Most base classes take control on this object by accessing information, data and objects during implementation of the module.

That means:

- Access to API program
- Connection with program in particular context
- Mapping module context settings
- Managing windows, etc.

Below you can see the sample of conditional run of program depending on application from where it was started.

Code region

```
public override void OnCreate()
{
   base.OnCreate();
   if (ThisApplication.Context2.Product.Type == REXInterfaceType.Revit)
   {
   // insert your code here
   }
}
```

5.1.3. Module context

Access to the extension context can be provided by **`ExtensionContext`** property.

Description of Control property of REXExtensionContext.REXControl class:

- **`Language Support`** – it is used to let base class **`REXApplication`** know if module is or is not designed as multi-language. The best idea is to set it in **`Application`** class constructor.

- **`NoUI`** – information for the module not to display user messages in implicit mode.

- **`RunContext`** – contains string for identification of module starting context. Standard context is being mapped to the below enumerator, and for non-standard one to provide settings. **`StandardRunContext`** to **`Other`**. In that case module must be able to identify string set in property.

- **`StandardRunContext`** – contains value of **`RunContext`** property mapped to standard context enumerator.

Code region

```
public enum REXStandardRunContext
   {
    None,
    Other,
    RCad, //ASD
    Robot,
    Revit,
    ESOP,
    CBS,
    OperationClear
    }
```

For this toolkit, only the Revit Context will be used.

- **ShowErrorsDialog –** allows turning on/off error dialogs displayed during module creation and after actions of **OnRun** method.

- **UserInterface –** force module to work in implicit mode (hidden GUI). The best place to set via the source code is in **Application** class constructor.

- **VerifySelection** – information for the module about launching in the verification mode. Particular objects could be verified by application. Module should return true if it's able to handle selection or false if not. If module returns false, at least one position should be added to error collection in **OnRun** method. For example:

Code region

```
public override void OnRun()
{
   System.SystemBase.Errors.AddError("", "", null);
}
```

If error collection is empty module returns true in **Show** or **Create** method.
Sample for handling starting context which aim is to remove all objects generated by module.

Code region

```
public override void OnRun()
{
   base.OnRun();
   if
(ExtensionContext.Control.RunContext==REXConst.RunContext.OperationClear)
   {
      return;
   }
}
```

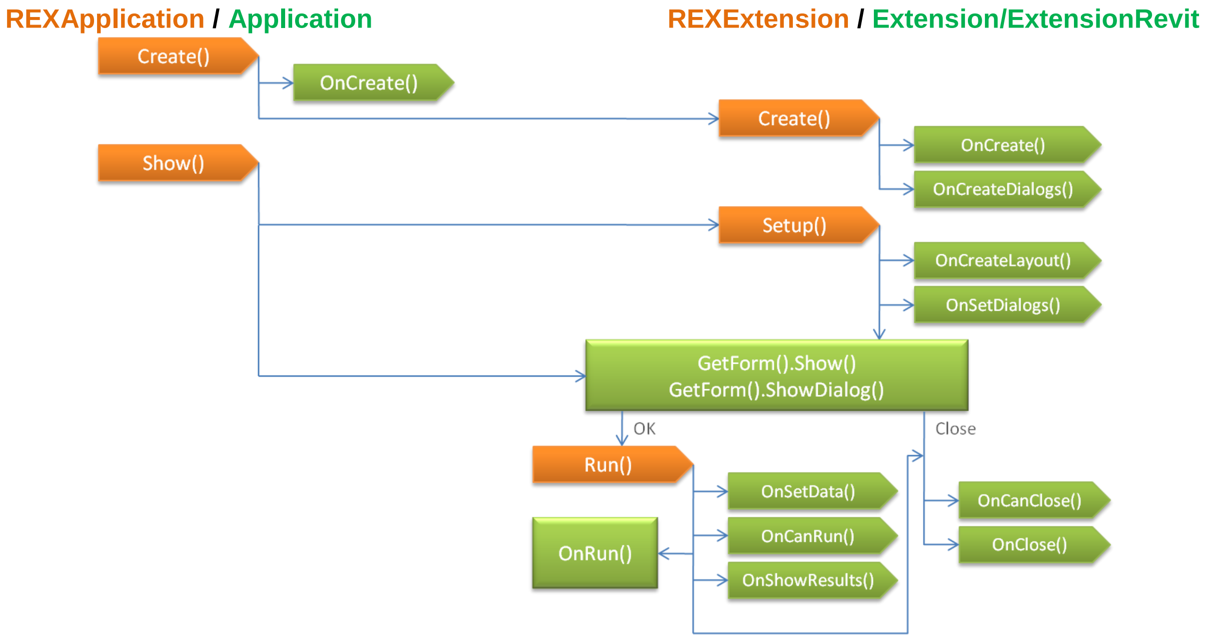
5.1.4. Starting context

Mainly, starting context determines the desired API interface. Standard context can be empty and then module works like a window. If standard context is set it can have standard value (this values are mapped to module context) **ExtensionContext.Control.StandardRunContext**, or non-standard values which are being identified by string **ExtensionContext.Control.RunContext**.

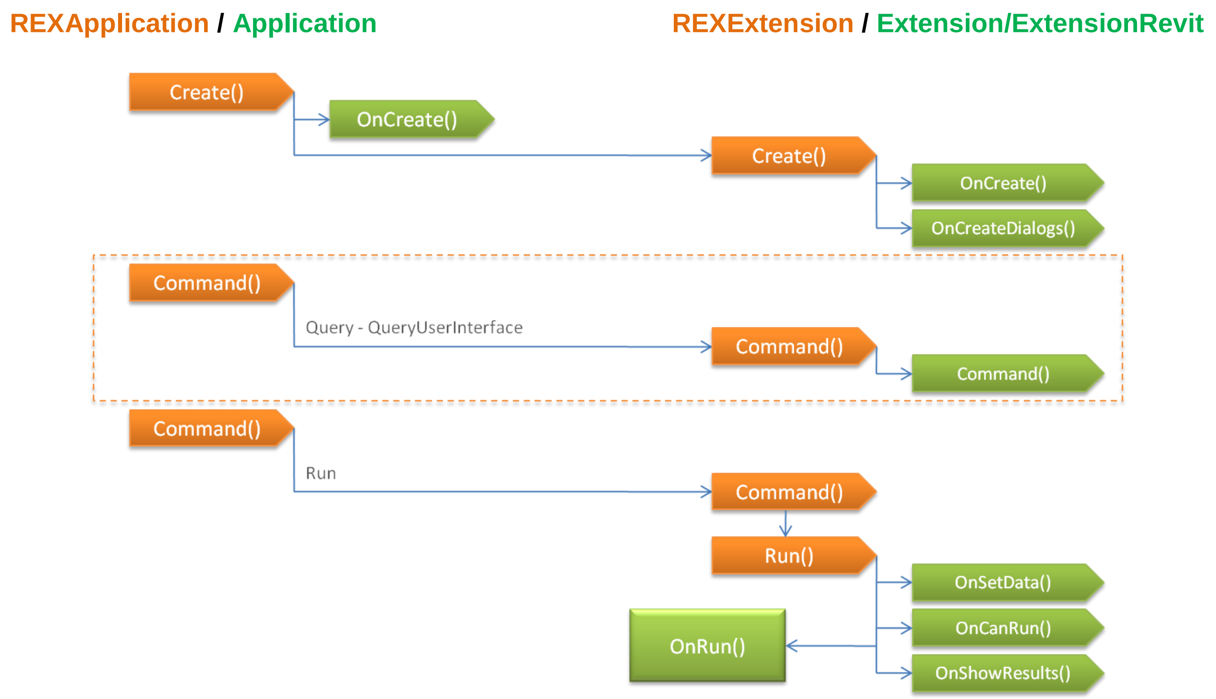
5.2. Method implementation

5.2.1. Method running order

Standard module starts in main context. Orange methods concern base classes and green methods developed by programmer.


REXApplication / Application
REXExtension / Extension/ExtensionRevit
Create()
OnCreate()
Show()
Create()
OnCreate()
OnCreateDialogs()
Setup()
OnCreateLayout()
OnSetDialogs()
GetForm().Show()
GetForm().ShowDialog()
OK
Close
Run()
OnSetData()
OnCanRun()
OnShowResults()
OnRun()
OnCanClose()
OnClose()

Module working scheme in starting context (default one):



If the method **`Command / QueryUserInterface`** returns true, the order of method running is the same as in main context.

It's a developer choice to implement methods highlighted in green in Extension.cs or in ExtensionsRevit.cs.

5.2.2. Application

Method description:

- **`OnCreate`** – this method is called during Application creation and allows developer to enable or disable some functionalities.
- **`GetVersion`** – this method allow developer to define minimal version of the components used.

Code region

```
public override bool OnCreate()
{
   AppExtension.ExtensionContext.Control.LanguagesSupport = true;
      return true;
}

public override string GetVersion(REXVersionType Version)
{
   switch (Version)
      {
              …
```

```
        case REXVersionType.CommercialEngine:
                  return "2.3";
        }
     return "";
}
```

5.2.3. Extension

This module main class is used during development to assure standard behavior.
There is a condition for context inside the class.

It inherits from **`REXExtension`** class. It's the base class for basic extension activities during lifecycle and provides several additional functions.

Main nethods description:

- **`OnCreate`** – this method creates all necessary objects used in explicit and implicit module mode. The creation should be context dependent.

Code region

```
public override void OnCreate()
{
   if (ThisApplication.Context2.Product.Type == REXInterfaceType.Revit)
      Revit.AutoSelectionRestore = true;
   base.OnCreate();
}
```

- **`OnCreateDialogs`** – creates all necessary dialogs and controls used by module in explicit mode.

Code region

```
public override void OnCreateDialogs()
{
     base.OnCreateDialogs();
     SubControlRef = new SubControl(this);
}
```

- **`OnCreateLayout`** – creates and sets the look of main module control (window). The look can be constant or option dependent.

Code region

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

// Example layout
Layout.ConstOptions = (long)REXUI.SetupOptions.HSplitFixed |
(long)REXUI.SetupOptions.VSplitFixed | (long)REXUI.SetupOptions.FormFixed;

Layout.ConstOptions |= (long)REXUI.SetupOptions.List |
(long)REXUI.SetupOptions.ToolMenu | (long)REXUI.SetupOptions.ToolBar;
// Group data
Layout.AddLayout(new REXLayoutItem(REXLayoutItem.LayoutType.Group, "Data",
"", Resources.Strings.Texts.LIST_Data, 0, null, null, 0));

Layout.AddLayout(new REXLayoutItem(REXLayoutItem.LayoutType.Layout, "L1",
"Data", "List, TabDialog, TabView, TabNote",
(long)REXUI.SetupOptions.TabDialog | (long)REXUI.SetupOptions.TabNote |
(long)REXUI.SetupOptions.TabView, SubControlRef, null, 0));


// Group results
Layout.AddLayout(new REXLayoutItem(REXLayoutItem.LayoutType.Group, "Results",
"", Resources.Strings.Texts.LIST_Results, 0, null, null, 0));

Layout.AddLayout(new REXLayoutItem(REXLayoutItem.LayoutType.Layout, "L5",
"Results", "List, Dialog", (long)REXUI.SetupOptions.TabDialog,
SubControlRef));


REXLayoutItem LayoutItem = Layout.GetItem("L1");



System.SetCaption();
}
```

- **`OnSetDialogs`** – copies internal structure of Data object to controls. It is called before showing dialog window.

Code region

```
public override void OnSetDialogs()
{
    base.OnSetDialogs();
    SubControlRef.SetDialog();
    Layout.SelectLayout("L3");
}
```

- **`OnSetData`** – method reverse to above one. It copies data from dialogs to internal structure of Data object. It's started after OK button click or in implicit mode.

Code region

```
public override void OnSetData()
{
    base.OnSetData();
    SubControlRef.SetData();
    if (Data.A <= 30)
        System.SystemBase.Errors.AddError("ErrCalc1", "Error 1", null);
}
```

- **`OnCanRun`** – verifies data and allows taking decision if any operation is started (**OnRun** method) or not.

- **`OnRun`** – takes action dependent on running context and starting context. In case of standard work as a dialog method is started after **`OK`** button click. In implicit mode it's started just after module creation – take a look on Method running order (4.6.1).

If any position is added to **`System.SystemBase.Errors`** collection, the window with error, warning and messages are shown after finishing of this method running. It's possible to disable and hide error dialog by this setting: **`ExtensionContext.Control.ShowErrorsDialog = false;`**

Code region

```
public override void OnRun()
{
    base.OnRun();
    ThisExtension.Progress.Show(GetWindowForParent());
    ThisExtension.Progress.Header = "Test Progress";
    ThisExtension.Progress.Text = "Status";
    ThisExtension.Progress.Steps = 20;
    for (int i = 0; i < Progress.Steps; i++)
    {
        ThisExtension.Progress.Step("Status " + i.ToString());
        Thread.Sleep(100);
    }
    ThisExtension.Progress.Position = 0;
    for (int i = 0; i < Progress.Steps; i++)
    {
        ThisExtension.Progress.Step("Status " + i.ToString());
        Thread.Sleep(100);
    }

    ThisExtension.Progress.Hide();

    Results.C = Data.A + Data.B;
    Results.D = Data.A - Data.B;
  }
```

- **`OnShowResults`** – optional method which allows to show results after **OnRun** method. This method is empty as default.

Code region

```
public override void OnShowResults()
{
    base.OnShowResults();
}
```

- **`OnCanClose`** – method gives ability to take decision if dialog could be closed or not.

Code region

```
public override bool OnCanClose()
{
    base.OnCanClose();
}
```

- **`OnClose`** – method uses to release all resources during module closing.

Code region
```
public override void OnClose()
{
     base.OnClose();
}
```

- **`OnActivateLayout`** – this method is called during activation or inactivation of window layout.

Code region
```
public override void OnActivateLayout(REXLayoutItem LItem, bool Activate)
{
  base.OnActivateLayout(LItem, Activate);
  if (Activate)
  {
    if (LItem.Name == "L1")
      {
        SubControlRef.Focus();
        REXLayoutItem LayoutItem = Layout.GetItem("L1");
        LayoutItem.Visible = false;
        Layout.Update();
      }
      if (LItem.Name == "L4")
      {
        REXLayoutItem LayoutItem = Layout.GetItem("L1");
        LayoutItem.Visible = true;
        Layout.Update();
      }
   }
   else
   {
   }
}
```

- **`OnErrorSelected`** – in case where error dialog is visible and errors from `System.SystemBase.Errors` collection are on this dialog, it's possible to handle particular user action with this method. The user action could be, for example, to choose an option on dialog and this method manage the response to this action.

- **`OnGetText`** – secondary method is generally used for getting texts using module internal methods. In case of resources obfuscation, this method avoids access issues.

Code region
```
{
  if (Name == REXConst.ENG_ResModuleDescription)
    return Resources.Strings.Texts.REX_ModuleDescription;
  if (Name == REXConst.ENG_ResVersionInfo)
    return Resources.Strings.Texts.REX_VersionInfo;
  return Resources.Strings.Texts.ResourceManager.GetString(Name);
}
```

5.3. Settings

Method description:

- **`OnParse`** – allows interpretation of marks from **`Configuration / settings.xml`** file. This file is read automatically by base class during module loading. Developer can put his own markups in setting section of a file:
  ```
    <settings>
  ...
    </settings>
  ```
  The standard structure of the file must be kept.

Code region

```
<?xml version="1.0" encoding="utf-8"?>
<ROXML version="1.0">
  <header>
    <contents type="configuration" version="1.5"/>
    <product type="component" category="library" technology=".net" name="REX"
/>
  </header>
   <settings>
….
   </settings>
</ROXML>
```

5.4. Data

This class is used to preserve internal data and structures of module derived from external program references. The values of all the data should be stored in REX base units – take a look at units handling paragraph.
Class data can be serialized in different contexts:

- File (**`ModeFile`**) – standard **rxd** file reading and writing. It's provided by **`System`** object methods: **`SaveToFile`**, **`LoadFromFile`**.

- Object (**`ModeObject`**) – is used to write data to an object in Revit project.

- Project (**`ModeProject`**) – is used to write data to project properties.

5.4.1. Method description

- **`OnSetDefaults`** – this method is used to set data default values. It's possible to separate the default value for imperial and metric units. This method is called before calling **`OnCreate`** of **`Extension`** class.

Serialized object checking is provided by base class fields. Current serialization version is set in Data class constructor (**`VersionCurrent`**).

Code region

```
public Data(REXExtension Ext): base(Ext)
{
  VersionCurrent = 1;
}
```

The loaded version is placed in **`VersionLoaded`** field.

- **`OnSave`**– saving data during serialization.

Code region

```
protected override void OnSetDefaults(REXUnitsSystemType UnitsSystem)
{
  if (UnitsSystem == REXUnitsSystemType.Imperial)
   {
      A = 10;
      B = 10;
   }
   else if (UnitsSystem == REXUnitsSystemType.Metric)
   {
      A = 20;
      B = 20;
   }
}
```

- **`OnLoad`**– loading data during serialization.

Code region

```
protected override bool OnLoad(ref BinaryReader Data)
{
  if (Mode == DataMode.ModeFile)
  {
  }
  if (Mode == DataMode.ModeProject)
  {
  }
  if (Mode == DataMode.ModeObject)
  {
  }
  if (VersionLoaded >= 1)
  {
    A = Data.ReadDouble();
    B = Data.ReadDouble();
  }
  return true;
}
```

5.4.2. Samples

The serialization sample demonstrates how to serialize data in one Revit family instance.
The FrameGenerator example demonstrates how to save data inside a file and load them later.

5.5. Results

It's an optional class similar to Data used for separating data and results logically. Behaviors are the same as for Data class.

5.6. ExtensionRevit

Inside ExtensionRevit class, developer could find a list of override methods. This class is the main class for development.

5.7. Foundation

The foundation class is a helper to initialize and start properly extension. Developer doesn't have to modify it.

6. GRAPHICS USER INTERFACE (GUI)

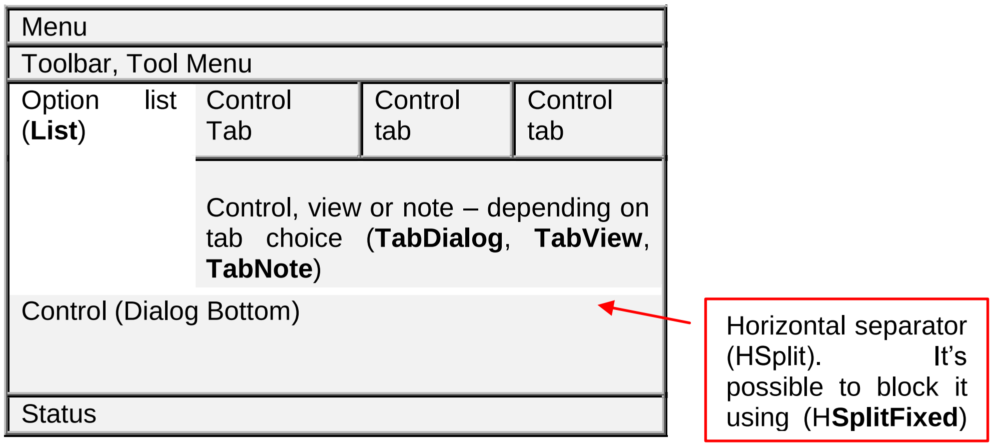
6.1. Introduction

Templates contain automation for look and interaction with the user.
2 technologies are available:

- Winforms
- WPF

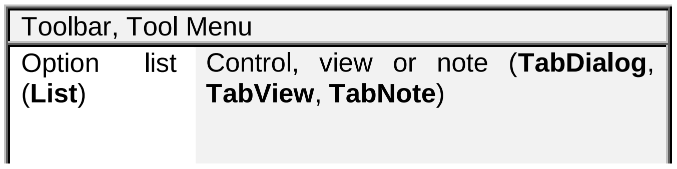
All is set in `OnCreateLayout` method using `REXLayoutItem` and `REXUI` objects.
`Setup Options` flags are used to define the look of the window. These flags are described in a separate paragraph. The look can be constant or different according to the list choice.
Following GUI configurations are possible:

- Full dialog configuration


Menu
Toolbar, Tool Menu
Option list (**List**)
Control Tab
Control tab
Control tab
Control, view or note – depending on tab choice (**TabDialog**, **TabView**, **TabNote**)
Control (Dialog Bottom)
Status
Horizontal separator (HSplit). It's possible to block it using (H**SplitFixed**)

- Using single `Tab` flag and bottom side dialog.

List | TabDialog | Dialog Bottom | Toolbar | Tool Menu | Status
Or List | TabView | …
Or List | TabNote | …


Toolbar, Tool Menu
Option list (**List**)
Control, view or note (**TabDialog**, **TabView**, **TabNote**)

Control (Dialog Bottom)
Status

- Using more than one **Tab** flag and bottom side dialog.

List | TabDialog | TabView | TabNote | Dialog Bottom
Or different **Tab** flag combinations.

Option list (**List**)	Control Tab	Control tab	Control tab
	Control, view or note – depending on tab choice (**TabDialog**, **TabView**, **TabNote**)		
Control (Dialog Bottom)			

- Using more than one **`Tab`** flag without bottom side dialog.

List | TabDialog | TabView | TabNote
Or different **`Tab`** flag combinations.

Option list (**List**)	Control tab	Control tab	Control tab
	Control, view or note – depending on tab choice (**TabDialog**, **TabView**, **TabNote**)		

To set constant look **`Layout.ConstOptions`** property should be set. It's also possible to set the same value to each option and the look will be constant.

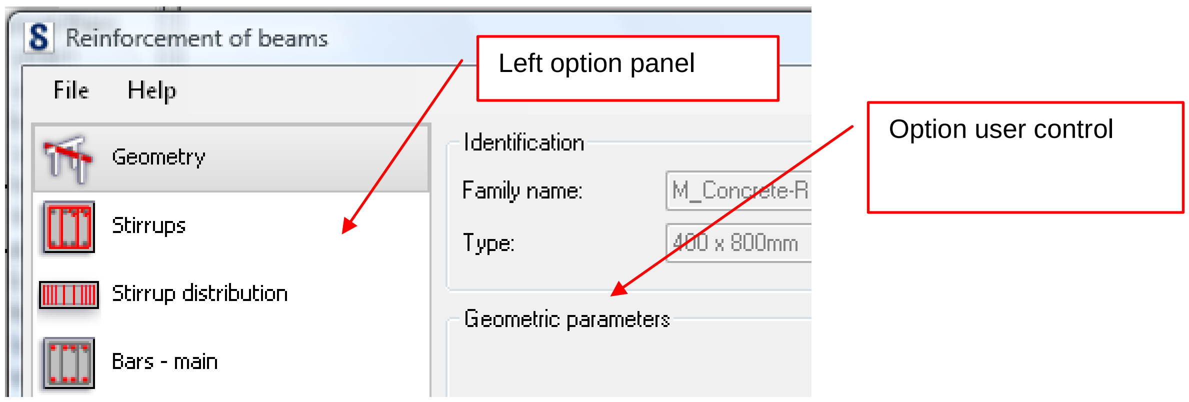
Code region
```
// Example layout
Layout.ConstOptions = (long)REXUI.SetupOptions.HSplitFixed |
(long)REXUI.SetupOptions.VSplitFixed | (long)REXUI.SetupOptions.FormFixed;
```

6.2. Design

The standard functionality of template gives possibility to embed user controls in **`Main Control`**.

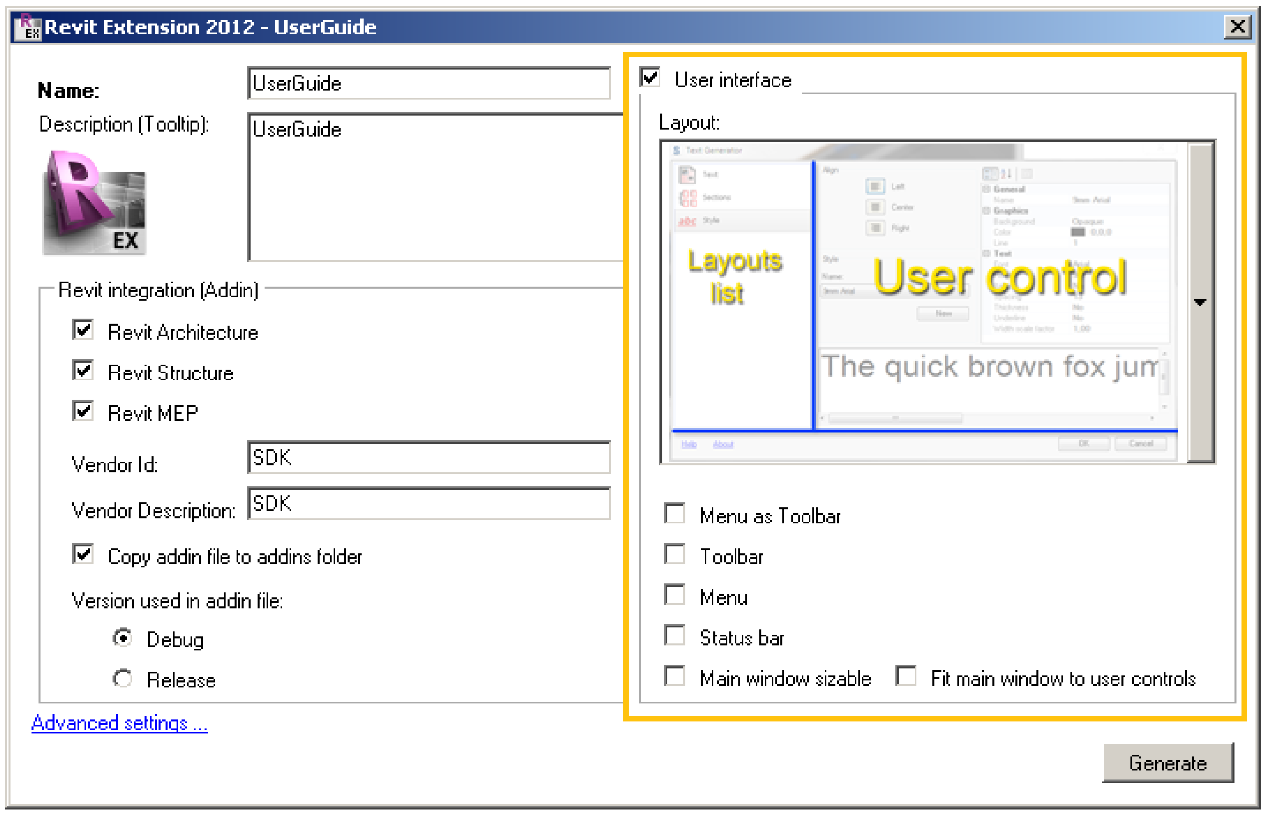
`Main Control` is automatically embedded in main module window (**`Main Form`**). Module can work in three standard modes: modal window, modeless window and control.

The order of designing standard module look:
- Specify all options accessible in left panel, additionally divided in groups.
- Design all user controls in connection with particular options.



Setting window look is done in `OnCreateLayout` method of Extension class.

6.3. Wizard


Revit Extension 2012 - UserGuide

Name: UserGuide

Description (Tooltip): UserGuide

Revit integration (Addin)

- [✓] Revit Architecture
- [✓] Revit Structure
- [✓] Revit MEP

Vendor Id: SDK

Vendor Description: SDK

- [✓] Copy addin file to addins folder

Version used in addin file:

- (●) Debug
- ( ) Release

[✓] User interface

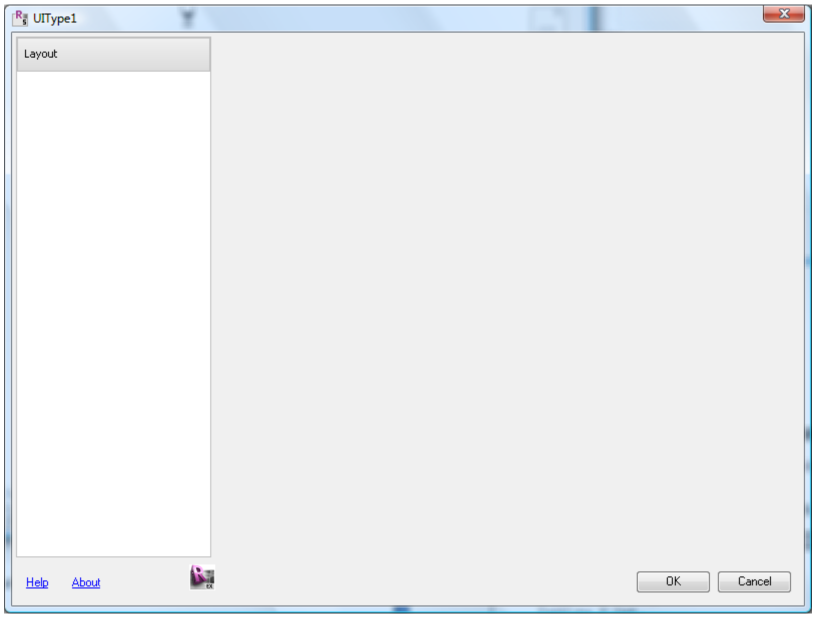
Layout:

- [ ] Menu as Toolbar
- [ ] Toolbar
- [ ] Menu
- [ ] Status bar
- [ ] Main window sizable
- [ ] Fit main window to user controls

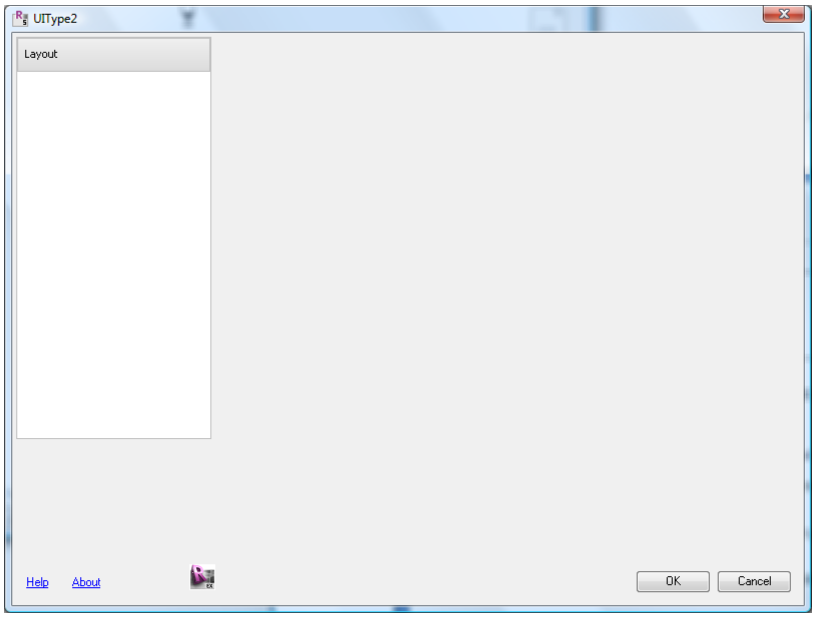
Using the Wizard, all flags will be set automatically to configure the UI. User Interface checkbox value should be in this case true.

The ComboBox layout will allow developers to configure 3 type of UI.

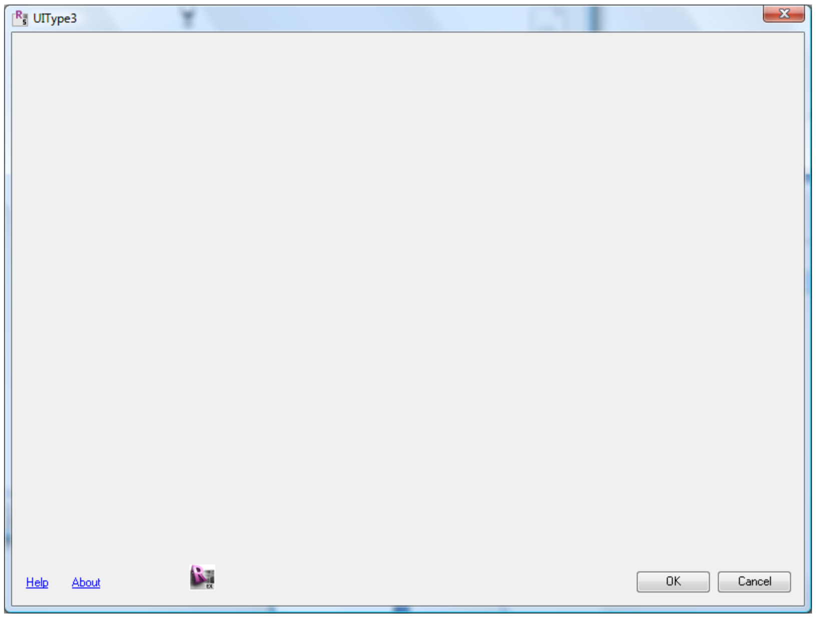
- Type 1


UIType1
Layout
Help
About
OK
Cancel

- Type 2

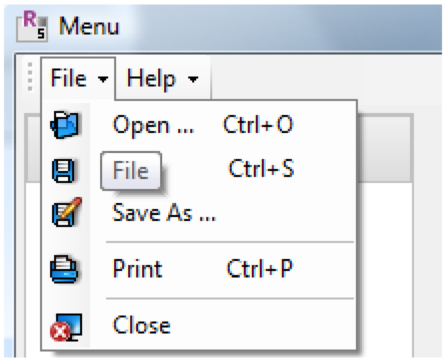

UIType2
Layout
Help
About
OK
Cancel

- Type 3

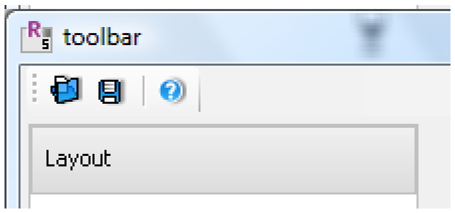


Some additional checkbox will add some specific behavior to the dialog.

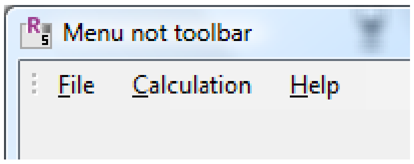
- Menu as toolbar



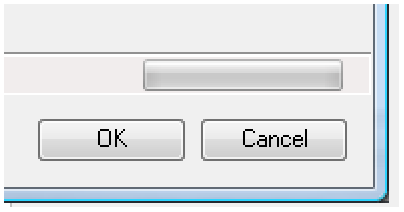
- Toolbar



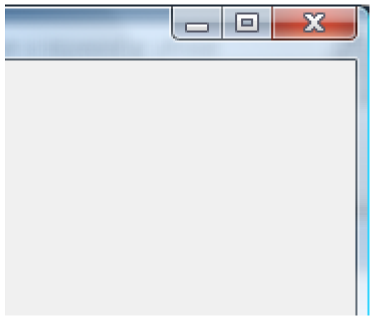
- Menu



- Status bar



- Main window sizable



- Auto size to users controls

6.4. Window look service (Managing layouts)

The classes **`REXUI`** and **`REXLayoutItem`** are responsible for setting window look and for interactions with options.

To set window look use binary flags defined in **`REXUI.SetupOptions`** enumerator:

Code region
```
   public enum SetupOptions
        {
            None = 0,
            List = 1,
            TabDialog = 2,
            TabView = 4,
            TabNote = 8,
            DialogBottom = 16,
            Status = 32,
            Menu = 64,
            ToolBar = 128,
            ToolBarInner = 256,
            ToolMenu = 512,
            ActivateControl = 1024,
            FormFixed = 2048,
            VSplitFixed = 4096,
            HSplitFixed = 8192,
            DisableDocking = 16384,
            ToolbarIncrementForm = 32768,
            AutoSizeForm = 65536,
        }
```

Access to flags is provided in **`Extension`** class by **`REXUI.SetupOptions.`**
Flag description:

- **`List`** – visibility of left option panel.

- **`TabDialog`** – user control visibility in right panel. User control is visible as a tab.
- **`TabView`** – visibility of control with view or control with other user interface in right panel. User control is visible as a tab.
- **`TabNote`** – visibility of control with note in right panel (embedded **`WebBrowser`**). User control is visible as a tab.

Using only **`Tab`** flag causes controls not to be embedded in **`TabControl`** container but directly positioned in right panel without tabs.

- **`DialogBottom`** – visibility of additional dialog located in bottom side if window.
- **`Status`** – status bar visibility.
- **`Menu`** – menu bar visibility.
- **`ToolBar`** – toolbar visibility.
- **`ToolBarInner`** – right panel toolbar visibility.
- **`ToolBarMenu`** – visibility of toolbar with docked menu.
- **`ActivateControl`** – activation of dialog control, view or note after option choice (not implemented in current version)
- **`FormFixed`** – determine if dialog have a constant size or if user can change window size
- **`VSplitFixed`** – vertical split bar blockade
- **`HSplitFixed`** – horizontal split bar blockade
- **`DisableDocking`** – toolbar docking blockade inside window
- **`ToolbarIncrementForm`** – increase height of main window (**`MainForm`**) by menu bar height or toolbar height.
- **`AutoSizeForm`** – automatic window size adjustment to cover all used controls, by default it's necessary to set main window size (**`MainForm`**) manually to cover main control (**`MainControl`**) and user controls (not implemented in current version)

To set Menu and Toolbars according UI designer wishes, **`REXUI.CommandOptions`** enumerator could be used:

Code region
```
public enum CommandOptions
        {
            None = 0,
            ToolBarOpen = 1,
            ToolBarSave = 2,
            ToolBarCalculate = 4,
            ToolBarHelp = 8,
            ToolMenuFile = 16,
            ToolMenuFileOpen = 32,
            ToolMenuFileSave = 64,
            ToolMenuFileSaveAs = 128,
            ToolMenuFilePrint = 256,
            ToolMenuFileClose = 512,
            ToolMenuCalculation = 1024,
            ToolMenuCalculationRun = 2048,
            ToolMenuHelp = 4096,
            ToolMenuHelpRun = 8192,
            ToolMenuHelpAbout = 16384,
            MenuFile = 32768,
            MenuFileOpen = 65536,
            MenuFileSave = 131072,
            MenuFileSaveAs = 262144,
            MenuFilePrint = 524288,
            MenuFileClose = 1048576,
            MenuCalculation = 2097152,
            MenuCalculationRun = 4194304,
            MenuHelp = 8388608,
            MenuHelpRun = 16777216,
            MenuHelpAbout = 33554432,
        }
```

6.5. User controls

Module GUI is based on user controls. **`REXExtensionControl`** template could be added via the add item Visual Studio command,

Code region

```
    public partial class SubControl : REXExtensionControl
{
   public SubControl(REXExtension Ext)
         : base(Ext)
   {
       InitializeComponent();
   }
}
```

6.6. REX Controls

The controls package consists of three assemblies:

- REX.Controls.dll - Contains the common part which is technology independent.
- REX.Controls.Forms.dll - Contains implementation of System.Windows.Forms controls.
- REX.Controls.WPF.dll - Contains implementation of WPF controls.

In order to use the controls of the specified technology it is necessary to add following references to the project:

- REX.Controls.dll
- REX.Controls.(Forms or WPF).dll

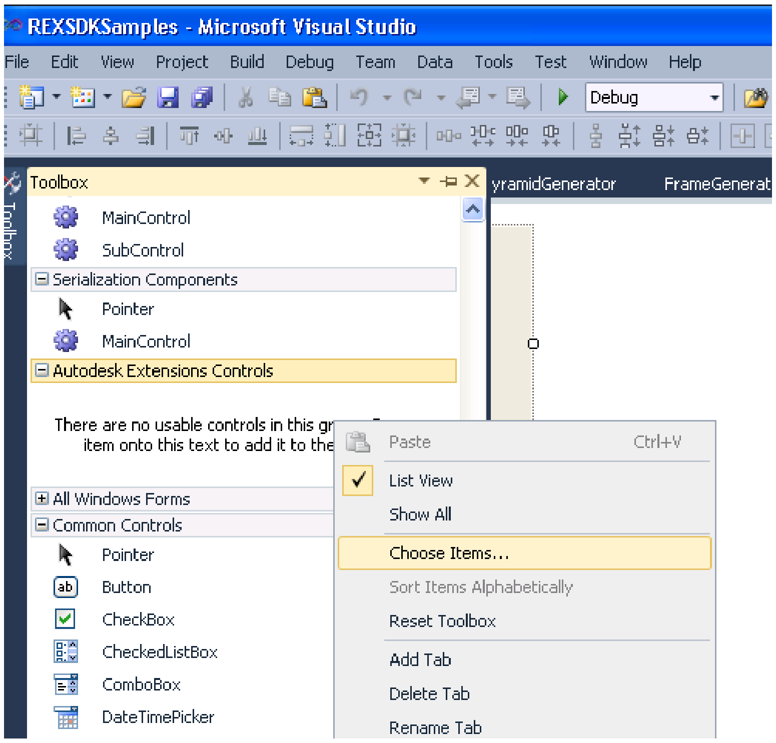
Namespace to use are:

- REX.Controls.Common
- REX.Controls.Forms
- REX.Controls.WPF

This document describes the implementation of the controls based on the Forms technology. The implementation in WPF technology is in most cases the same. All significant differences are pointed out at the ends of controls descriptions.

Sometimes, controls are not included automatically on the Visual Studio toolbox. These controls could be added following common visual studio rules:

- Choose items



- Select REX.Controls.Forms for WinForms technology or REX.Controls.WPF for WPF from C:\Program Files\Common Files\Autodesk Shared\Extensions 2012\Framework\Engine folder.

6.6.1. REXEditBox

The *REXEditBox* control displays and formats the text entered at design time or changed programmatically by the developer.

- **Data type**

The *REXEditBox* supports texts and numeric data. The data type is determined by the `DataType` property.
There are following types available:

- `TEXT`
- `DECIMAL,`
- `IMPERIAL_FRACTIONAL_INCHES,`
- `IMPERIAL_FRACTIONAL_FEET_INCHES`

- **Setting and getting control values**

The value of the control can be specified by:
- Number
- List of numbers
- Text
- List of texts

There is a group of methods responsible for setting and getting values of specified types:
- `SetComplexValue`
- `SetValue`
- `GetValue`
- `GetText`
- `GetComplexDoubleValue`
- `GetComplexStringValue`

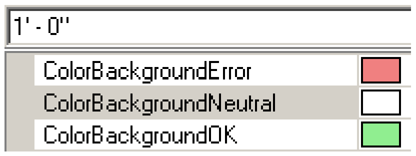
The text value may be also set and get directly by using the `Text` property.

- **State of the control**

Generally the control may be in either of two states:
- `OK` – when the control is validated
- `ERROR` – when validation of the control fails

To learn about the state of the control, use the `GetState` method. It returns the value of the `EControlState` type (`eOK, eEMPTY,eFORMAT, eRANGE_MIN, eRANGE_MAX, eUNKNOWN`).

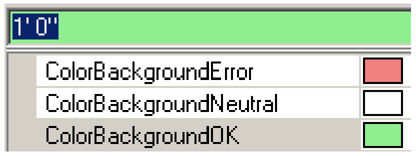
The color of control's background is determined by its state:
If the state is OK and the *REXEditBox* is not in the edit mode the background color will be taken from the `ColorBackgroundNeutral` property.



1' - 0"

ColorBackgroundError	🟥
ColorBackgroundNeutral	⬜
ColorBackgroundOK	🟩

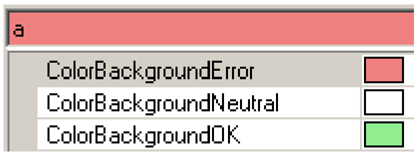
If the state is OK and the *REXEditBox* is in the edit mode the background color will be taken from the `ColorBackgroundOK` property.



1' 0"

ColorBackgroundError	🟥
ColorBackgroundNeutral	⬜
ColorBackgroundOK	🟩

If the state is ERROR the background color will be taken from the `ColorBackgroundError` property (the edit mode doesn't matter in this case).



a

ColorBackgroundError	🟥
ColorBackgroundNeutral	⬜
ColorBackgroundOK	🟩

The way of setting the background will change a little bit when the control is disabled but this will be described later in this document.

There is an additional option related to the state of the *REXEditBox* - the `RememberCorrect` property. If this property is set to true, it allows the user to return to the correct value automatically after validation, otherwise the control will stay in the ERROR state.

- **Numeric data**

There are 3 types of numeric data:

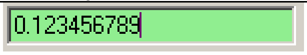
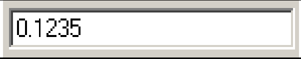
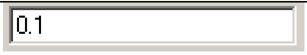
- `DECIMAL`	- decimal
- `IMPERIAL_FRACTIONAL_INCHES`	- fractional inches
- `IMPERIAL_FRACTIONAL_FEET_INCHES`	- feet and fractional inches

The main difference between all types is obviously the formatting style.

- **Decimal format**

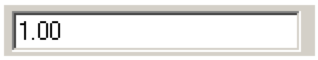
The format of decimal values depends on following properties:

- RoundingIncrement
  `RoundingIncrement` specifies the rounding of the decimal value.

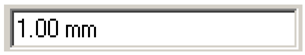
The input value	0.0001	0. 1
 0.123456789	 0.1235	 0.1

- Separator
  `Separator` specifies the decimal separator. By default it is taken from the Windows settings.

- UnitSymbol
  `UnitSymbol` specifies the unit symbol which is added at the end of the formatted value.


1.00

With undefined `UnitSymbol` (empty string)

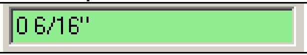
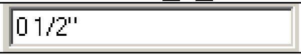
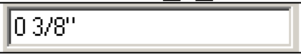

1.00 mm

With `UnitSymbol` equals “mm”

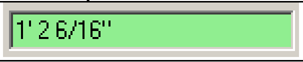
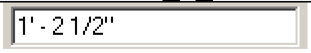
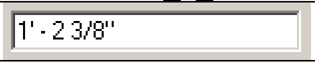
- **Imperial format**

The format of imperial values depends on the `RoundingFractional` property.

Fractional inches:

The input value	ROUNDING_1_2	ROUNDING_1_8
 0 6/16"	 0 1/2"	 0 3/8"

Feet and fractional inches:

The input value	ROUNDING_1_2	ROUNDING_1_8
 1' 2 6/16"	 1' - 2 1/2"	 1' - 2 3/8"

The formatted value is always reduced to the simplest representation no matter which `RoundingFractional` is set (e.g. 2/4" is reduced to ½ even if `ROUNDING_1_8` is set)

- **Value types**

There are two types of numeric values supported by the *REXEditBox*:
- Single value
- Complex value – a list of values

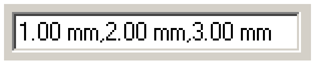
The mode in which the control works is determined by the `Complex` property. If true it supports complex value, otherwise it supports single value.

A number in a complex value is formatted in the same way as a single value. The additional things which may be set are:
- ComplexSeparator

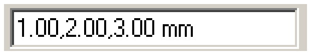
`ComplexSeparator` – Specifies the separator between items of the list. It is also possible to use a space as the items separator - the `ComplexSpaceSeparator` property is responsible for this option.

- ComplexUnitOnEnd

`ComplexUnitOnEnd` – Indicates whether a unit symbol should be placed at the end of the whole formatted string. If the property is set to false a unit symbol is placed next to each item on the list.


1.00 mm,2.00 mm,3.00 mm

`ComplexUnitOnEnd` = false


1.00,2.00,3.00 mm

`ComplexUnitOnEnd` = true

- **Validation**

The entered value may be validated against different type of constraints provided by the control:

- Value range

  The control validates the value against minimum and maximum values specified by developer. There is a bunch of properties responsible for this part:
  - `RangeMax` – Specifies the highest possible value.
  - `RangeMaxCheck` – Indicates whether the value should be validated against the `RangeMax` property.
  - `RangeMin` – Specifies the lowest possible value.
  - `RangeMinCheck` – Indicates whether the value should be validated against the `RangeMin` property.

  In case of complex values each single value is validated when the range is checked.

- Number of elements in the list ("Complex" mode):

  It is possible to limit the number of elements in the list:

- `MaxTokens` – Specifies the highest number of items.
- `TokensRangeMaxCheck` – Indicates whether the number of items should be validated against the `MaxTokens` property.
- `MinTokens` – Specifies the lowest number of items.
- `TokensRangeMinCheck` – Indicates whether the number of items should be validated against the `MinTokens` property.

- User validation:

  The user is able to define its own validation rules by providing the external validation methods. This is obtained through following methods:
  - `SetUserValidation` – for single values
  - `SetUserValidationComplex` – for complex values

  The method has to decide if the specified value is against internal constraints. For example:

Code region
```
private bool Valid(double val, IUnitObject uo)
{
    if (val > 3)
        return true;
    else if (val < 0)
        return true;
    else
        return true;
}
```

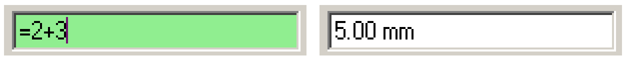
The example above checks if the value is greater than 3 and less than 0. The next example will show how to constrain the complex value in such a way that the sum of its items must be less than 10.

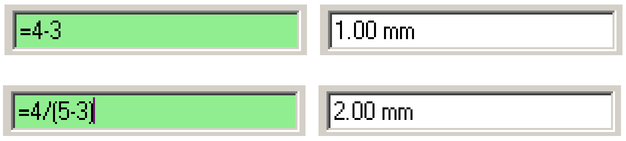
Code region
```
private bool ValidComp(List<double> vals, IUnitObject uo)
{
    double s = 0;
    foreach (double val in vals)
    {
        s += val;
    }
    if (s > 10)
        return false;
    return true;
}
```

- **Formulas**

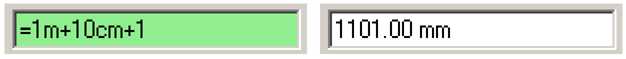
Formulas can also be entered into the *REXEditBox*. If there is "=" at the beginning of the string the formula examination is run by the control. Following operators are supported: +, -,/,*,()

Examples:


=2+3
5.00 mm



Additionally the *REXEditBox* recalculates unit symbols which are not set as the current one (`UnitSymbol`):

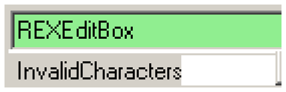


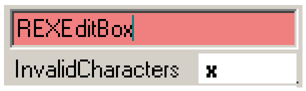
- **Text data**

The REXEditBox can be used as the simple text control (TEXT DataType). In this mode the control provides two options for validation:

- InvalidCharacters

Specifies the list of characters which are invalid:





- User validation:

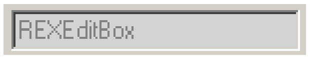
The user is able to define its own validation rules by providing an external validation through the `SetUserTextValidation` method, for example:

Code region
```
private bool Valid(string val, REX.Controls.Common.IUnitObject uo)
{
    if (val.Contains("x"))
        return false;
    return true;
}
```

- **Disabled state**

The background and foreground of the *REXEditBox* may behave in two different ways when the control is disabled:

- Standard – color of the background determined by the `ColorBackgroundStandart` property

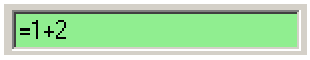


- Hide foreground – color of the background determined by the `ColorBackgroundDisabled` property

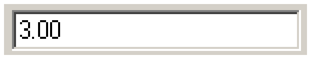
Behaviors described above are determined by the `OnDisabledAction` property

- **Entering the REXEditBox**

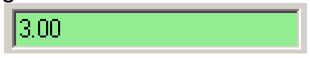
When the *REXEditBox* is entered by the user it takes the current formatted string as a base for analysis. If user inputs string:



It will be formatted as below when user will left the control



If user starts to edit it once again he will get:



Set the `OriginalTextOnEnter` property to true in order to return the original text with the formula.

- **WPF**

Names of properties are different:

ColorBackgroundOK	BackgroundOK
ColorBackgroundError	BackgroundError
ColorBackgroundDisabled	BackgroundDisabledHidden
ColorBackgroundStandard	BackgroundDisabledStandard

In WPF version it is possible to use the `DoubleValue` and `ComplexDoubleValue` to set and get the numerical value of the control (for binding purpose mainly).

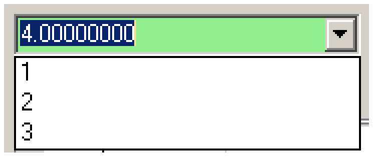
6.6.2. REXComboBox

The REXComboBox control combines a *System.Windows.Forms.ComboBox* control with the *REXEditBox* control.

Most of the features provided by the *REXComboBox* are the same as in the *REXEditBox* and they are described in the *REXEditBox* section. Below is the description of functionalities which are characteristic of a *REXComboBox*.

- **Current Value**

The current value of the *REXComboBox* may be taken from the item collection or input by the user (as in *REXEditBox*).



Use the `GetPropertiesType` method in order to find out in which mode is the control at the moment. This method returns the result of the `EComboPropertiesType` type. There are two modes available:

- List – the current value was taken from the list

- Main – the current value was input by the user

- **Validation of the list**

Items of the list may be validated in the same way as values input to the control. Set the `ListValidation` to true if you want to switch this option on.

***REXEditBox* mode**

It is possible to switch the *REXComboBox* to ordinary *REXEditBox* without adding an additional control. The simplest way to do it is to change the `TextBoxMode` to true. In this case the *REXComboBox* changes to the *REXEditBox* which instance are available through the `EditBox` property.

6.6.3. Controls in the REX environment

The Extensions Framework provides support for the units system defined by the Product context. Units can be divided into three types:

- Base – internal base REX units
- Interface – internal product units (Revit API)
- Display – units set currently in the project (Revit project)

For instance if we have a Revit project with the unit for the "Length" category set to "mm" there are:

- Base – "m"
- Interface – "ft"
- Display – "mm"

Generally controls should be formatted according display units, data should be stored in base units and all operation on Revit API has to be made in interface units.

Extensions Framework provides support for units operations (the `IREXUnits` interface). There is an additional object which helps managing controls. It is `DFM`, a member of the `REXExtension` class:

- **Registration**

In order to use `DFM` for managing the control it is obligatory to register it in the system. It is done by using the `AddUnitObject` method. During definition the user decides about the unit category and power of the specified control.
It is also possible to define the key name which will be used for the control identification but it is not required (the reference of the control may be used directly).

Code region

```
ThisExtension.DFM.AddUnitObject(rexCombo,EUnitType.Dimensions_SectionDim, 1);
ThisExtension.DFM.AddUnitObject(rexEdit, EUnitType.Dimensions_Angle, 1);
```

- **Setting and getting values**

Values can be set to and read from the control in all unit types (base, interface, display). There is a group of dedicated methods:
Set:

- `SetDataBase`
- `SetDataDisplay`
- `SetDataInterface`
- `SetComplexDataBase`
- `SetComplexDataDisplay`
- `SetComplexDataInterface`

Get:

- `GetDataBase`
- `GetDataDisplay`
- `GetDataInterface`
- `GetComplexDataBase`
- `GetComplexDataDisplay`
- `GetComplexDataInterface`

- **Validation**

DFM supports also validation mechanisms of the specified control:
Range:

- `SetBaseMaxValue`
- `SetInterfaceMaxValue`
- `SetBaseMinValue`
- `SetInterfaceMinValue`

User validation:

- `SetUserValidFunction`
- `SetUserValidFunctionComplex`

Code region

```
ThisExtension.DFM.SetBaseMinValue(rexEditBox1,true,0);
ThisExtension.DFM.SetBaseMaxValue(rexEditBox1, true, 10);
ThisExtension.DFM.SetDataBase(rexEditBox1, 2);
double interfaceVal = ThisExtension.DFM.GetDataInterface(rexEditBox1);
```

- **IUnitObject**

`DFM` is based on the `IUnitObject` and it is possible to use it for any type of the control which implements this interface (e.g. custom control). The interface was created based on the *REXEditBox* so please refer to it in case of doubts about required functionality.

6.6.4. REXUnitComboBox and REXUnitEditBox

The *REXUnitComboBox* and the *REXUnitEditBox* extends the functionality of the *REXEditBox* and the *REXComboBox* accordingly. They provide functionality of controls combined with the REX unit system. They are alternative solution for `DFM`.

- **Unit settings**

The controls are customized by following settings:

`UnitType`
    Specifies the category of the unit.
`Power`
    Specifies the power of the control.
`UnitEngine`
    Specifies the unit engine taken from the REX environment. It is obligatory to set it; otherwise the control will work as regular one.

Code region

```
        rexUnitEditBox.UnitEngine = ThisExtension.Units.UnitsBase;
        rexUnitEditBox.Power = 2;
        rexUnitEditBox.UnitType = EUnitType.Dimensions_SectionDim;
```

- **Methods**

All properties and methods inherited from base controls (*REXEditBox*, *REXComboBox*) are set in “display” units. In order to set values which are described in the “interface” or the “base” unit, dedicated methods have to be used:

Set:

- `SetDataBaseValue`
- `SetDataInterfaceValue`
- `SetComplexDataBaseValue`
- `SetComplexDataInterfaceValue`

Get:

- `GetDataBaseValue`
- `GetDataInterfaceValue`
- `GetComplexDataBaseValue`
- `GetComplexDataInterfaceValue`

Validation:

- `SetBaseMaxValue`
- `SetInterfaceMaxValue`
- `SetBaseMinValue`
- `SetInterfaceMinValue`
- `GetBaseMaxValue`
- `GetInterfaceMaxValue`
- `GetBaseMinValue`
- `GetInterfaceMinValue`

Code region

```
      rexUnitEditBox.SetBaseMinValue(0);
      rexUnitEditBox.SetBaseMaxValue(10);
      rexUnitEditBox.SetDataBaseValue(2);
double interfaceVal =
rexUnitEditBox.GetDataInterfaceValue(REXInterfaceType.Revit);
```

- **WPF**

In WPF version it is possible to use the `DoubleValueBase` and `ComplexDoubleValueBase` to set and get the numerical value of the control (for binding purpose mainly).

6.6.5. REXEditControlEngine

The *REXEditControlEngine* class represents the engine of the *REXEditBox* control. It can be used in user's edit control or as the independent standalone engine.

- **Standalone engine**

Usage of engine is exactly the same as usage of the *REXEditBox* control but without the user interface:
Example:

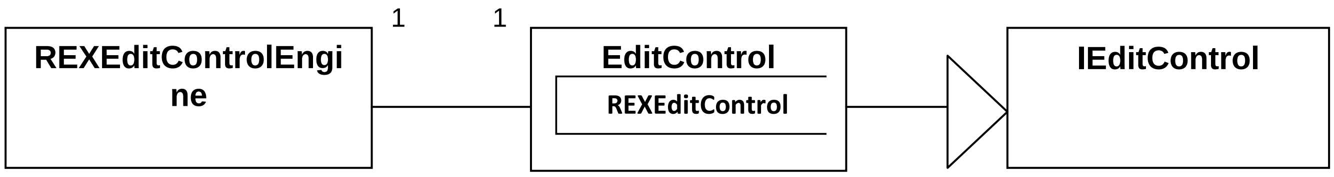
Code region

```
REXEditControlEngine<Color> engine = new REXEditControlEngine<Color>(null);
engine.DataType = REX.Controls.Common.EControlType.DECIMAL;
engine.UnitSymbol = "cm";
engine.SetValue(2);
string txt = engine.Text;
engine.SetValue("=1+2m");
double val = engine.GetValue();
```

- **<u>Using the REXEditControlEngine in user's control</u>**

If the user wants to use the *REXEditControlEngine* in his custom control, he can use provided mechanisms to synchronies the engine with the control. The only conditions that have to be satisfied are:

- Implementation of the *IEditControl* interface



- Ensure exchange of information between engine and the control (text changing, edition start, edition end, setting values, getting values etc.)

- **<u>IEditControl</u>**

  - `GetEnable`
    Informs about the state of the control (whether the control is enabled).
  - `SetText`
    Sets the text content of the control.
  - `GetText`
    Gets the current text content of the control.
  - `SetBackground`
    Sets the background of the control.
  - `GetBackground`
    Gets the background of the control.

- **<u>Information exchange</u>**

The user has to remember about synchronization of properties of the control with properties of the engine. The best practice is to use the engine as a data container and provide a façade to its properties:

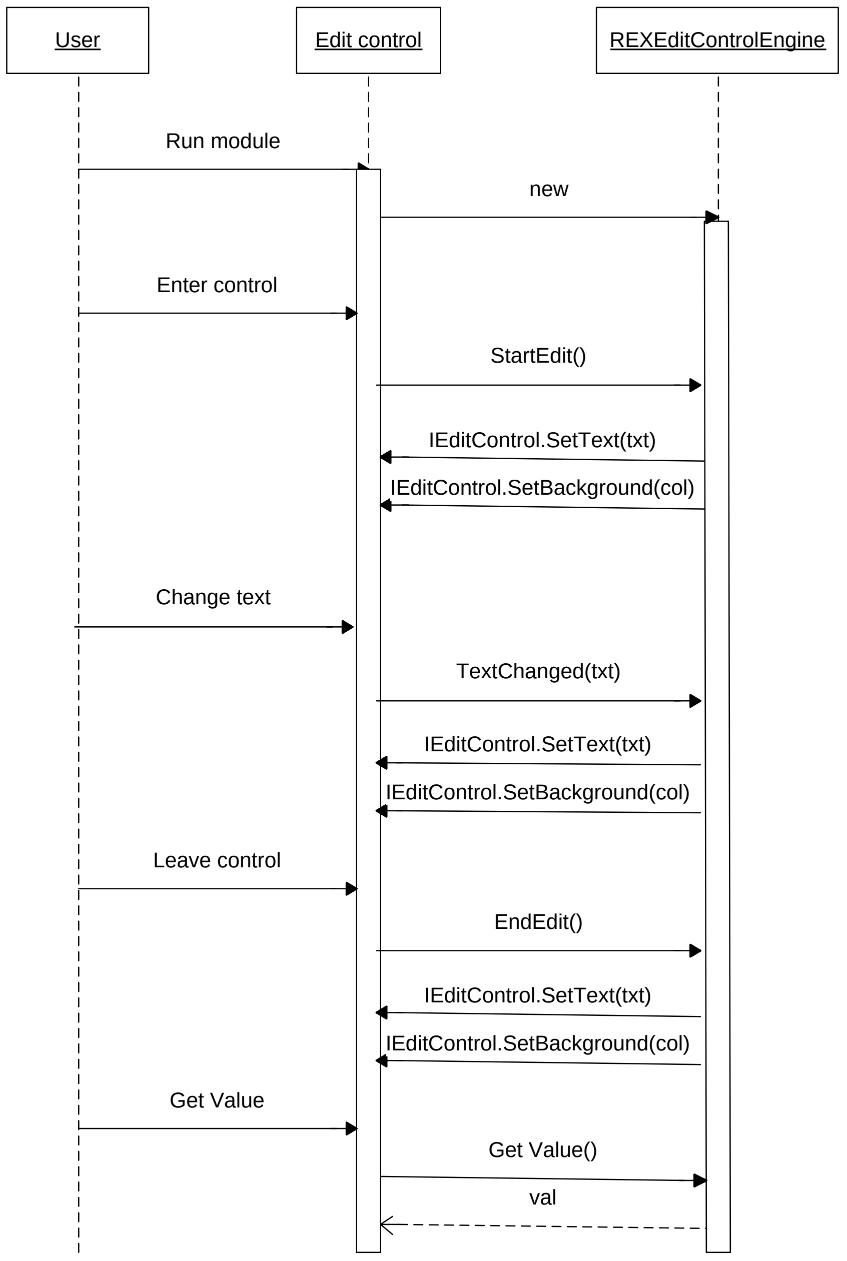
Code region

```
public Double RangeMin
{
    set
    {
        editControl.RangeMin = value;
    }
    get
    {
        return editControl.RangeMin;
    }
}
```

In case of events the control has to inform the engine about changes. For example when user enters the control, the `StartEdit` method should be called.
As a result the engine goes into edition mode, changes control's background and sets an appropriate text.

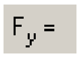
The diagram underneath shows example of the control and the engine cooperation:

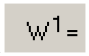
1. User launches extension
2. The control is created
3. The control initializes its own engine (*REXEditControlEngine*) and pass own reference to it (as implementation of the *IEditControl* interface)
4. User enters the control.
5. The control starts the edition mode of the engine (the `StartEdit` method).
6. The engine response by setting the background and text of the control (through the *IEditControl* interface).
7. User enters a text inside the control.
8. The control sends information about text change to the engine (through the `TextChange` method).
9. The engine validates entered text and sets the background and the text in the control (through the *IEditControl* interface)
10. User leaves the control.
11. The control ends the edition mode of the engine (the `EndEdit` method).
12. The engine response by setting the background and text of the control (through the `IEditControl` interface).
13. User finishes its work with the module. The module gets the current value of the control.
14. The control gets the value which is stored inside the engine.


User
Edit control
REXEditControlEngine
Run module
new
Enter control
StartEdit()
IEditControl.SetText(txt)
IEditControl.SetBackground(col)
Change text
TextChanged(txt)
IEditControl.SetText(txt)
IEditControl.SetBackground(col)
Leave control
EndEdit()
IEditControl.SetText(txt)
IEditControl.SetBackground(col)
Get Value
Get Value()
val

6.6.6. REXIndexLabel

The *REXIndexLabel* control is dedicated for labels where a lower and an upper index have to be presented:

- The lower index 

- The upper index 

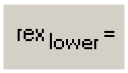
- **Properties**

  - `Text`
    Specifies the content of the control.
  - `LeftAlign`
    Specifies the text alignment in the control (true if left alignment; otherwise right alignment).
  - `IndexCoeff`
    Specifies the proportion between index and its parent.

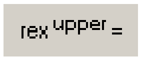
- **Syntax**

In order to apply lower or upper index the `Text` property have to be input in the specific format:

- In order to draw small letters it is necessary to place them in curly brackets:

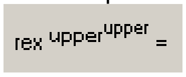


`Text` = “rex{lower}”



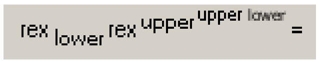
`Text` = “rex{^upper}”

It is also possible to make embedded structures:



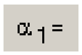
`Text`=”rex{^upper{^upper}}”

And mixed ones:



`Text` = “rex{lower} rex{^upper{^upper{lower}}}

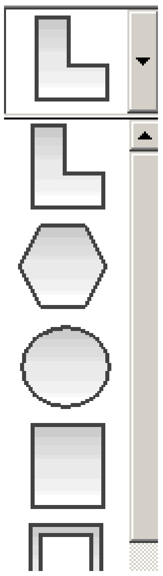
In order to apply “Symbolic” font to the specified character, the @ mark has to be used:



`Text` = “@a{1}”

6.6.7. REXImageComboBox

The *REXImageComboBox* represents the ComboBox with images as its items.



- **REXImageComboBoxItem**

The main part of the *REXImageComboBox* is the item represented by *REXImageComboBoxItem*. Instances of the *REXImageComboBox* are added directly to the `Items` collection. It is also possible to add a simple text directly to the *REXImageComboBox*. Main properties of the *REXImageComboBox*:

`Text`
Represents the text of the item.

`Image`
Represents the image source of the item.

`VisibleOnList`
Indicates whether the item should be visible on the list. If the item is not visible on the list it is still possible to select it directly from the code.

`Tooltip`
Represents the tooltip of the item.

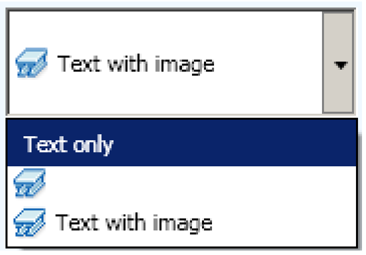
Example:

Code region
```
REXImageComboBoxItem item = new REXImageComboBoxItem()
{ VisibleOnList = true, Image = Resources.Image1};
rexImageComboBox1.Items.Add(item);
rexImageComboBox1.Items.Add("Simple text");
```

- **WPF and Forms**

The main difference between Forms and WPF implementation is behavior when the text and the image are set at the same time:

- In case of Forms *REXImageComboBox* the image is presented without text (the text can be used in this mode as a Tooltip –the `UseTextAsToolTip` property).
- In case of WPF *REXImageComboBox* the text is presented next to the image and `Tooltip` can be set separately:



7. UNITS

Three types of units are used during work with extension:
- base units (internal Extensions)
- interface units (Revit API)
- User units (displayed on dialogs according user settings).

- All internal values which need to be saved to objects should keep the value in REX base units. The unit system is metric.
- Interface units concern Revit data exchanged with Extension. During getting data from API interfaces of Revit the data must be recalculated from interface units to base units. During setting data to API interface of Revit the data must be recalculated from base units to interface units.
- The situation is analogical when displaying values on dialogs. During displaying value, data must be recalculated from basic units to user units. During getting data from dialogs, the units must be recalculated from user units to base units to obtain a correct formatting in Revit UI.

REX Engine and base classes provide support for unit handling. Application settings are used to set user units during start-up of module. Interface units are defined for different application context.

REX units are divided into categories. Every category has definition which is based on unit group.
Unit group: **`UG_Length`**

Code region
```
<unitgroup name="UG_Length">
  <unit name="mm" coefficient="0.001" />
  <unit name="cm" coefficient="0.01" />
  <unit name="m" coefficient="1" />
  <unit name="km" coefficient="1000" />
  <unit name="in" coefficient="0.0254" />
  <unit name="ft" coefficient="0.3048" />
  <unit name="yd" coefficient="0.9144" />
  <unit name="mile" coefficient="1609.344" />
</unitgroup>
```

The category is based on groups with an independent definition for imperial and metric unit.

Code region

```
<category name="UC_Length" power="-1" topower="6">
   <definition system="metric" precision="0.1" exponential="false">
       <unitset group="UG_Length" unit="in" power="1" />
   </definition>
   <definition system="imperial" precision="0.1" exponential="false">
       <unitset group="UG_Length" unit="ft" power="1" />
   </definition>
</category>
```

7.1. Unit groups

Group Name /Unit	Coefficient /Formula	Return formula
UG_Length		
mm	0.001	
cm	0.01	
m	1	
km	1000	
in	0.0254	
ft	0.3048	
yd	0.9144	
mile	1609.344	
UG_Force		
N	1	
daN	10	
kN	1000	
MN	1000000	
kG	9.80665	
T	9806.65	
b	4.448222	
kip	4448.222	
UG_Angle		
Rad	1	
Deg	0.01745329251994	
Grad	0.01570796326795	
UG_Temperature		
DegC	1	
DegF	(%x - 32)/1.8	1.8 * %x + 32
K	%x – 273	%x + 273
UG_Mass		
G	0.001	
kg	1	
t	1000	
lb	0.453592	
oz	0.02834952	
ton	907.184	
UG_Time		
s	1	
min	60	
h	3600	
day	86400	
week	604800	
year	31536000	
UG_Power		
W	1	
kW	1000	
MW	1000000	
hp	745.700	
UG_Energy		

Group Name /Unit	Coefficient /Formula	Return formula
J	1	
daJ	10	
kJ	1000	
MJ	1000000	
cal	4.186800	
UG_Frequency		
Hz	1	
kHz	1000	
MHz	1000000	
UG_Percent		
%	1	
UG_Stress		
Pa	1	
hPa	100	
kPa	1000	
MPa	1000000	
psf	47.88026	
psi	6894.75789	
ksi	6894757.89	
bar	100000	
UG_CostTimeWork		
w-h	1	
UG_CostTimeEquipment		
m-h	1	
UG_Currency		
$	1	
L	1	
EUR	1	
Zł	1	

7.2. Base categories

Name	Base Unit	Revit
UCG_Dimensions		
UC_Length	m	X
UC_Angle	Rad	X
UC_SectionDim	m	X
UC_StructureDim	m	X
UC_SectionChar	m	X
UC_SectionDiam	m	X
UC_Displacement	m	X
UCG_Forces		
UC_Force	N	X
UC_ForceDistLen	N/m	X
UC_ForceDistSurf	N/m2	X
UC_ForceDistVol	N/m3	X
UC_Moment	N*m	X
UC_Stress	N/m2	X
UC_Pressure	N/m2	X
UC_MomentDistLen	(N*m)/m	X
UCG_MassWeight		
UC_Speed	m/s	X
UC_Frequency	Hz	X
UCG_Energy		
UC_Energy	J	
UC_Power	J	X
UCG_Temperature		
UC_Temp	DegC	X

7.3. Unit type

Extensions Framework provides support for units operations (the `IREXUnits` interface).
`EUnitType` enumeration list all type of unit available in Extension Framework and user could be used for internal calculation.
Only based categories listed previously are supported in Revit with all behaviors listed below.

7.4. Unit handling

REXUnits class is a layer between REX Engine and a module. Access to recalculation functions from the **`Extension`** class is following: **`System.Units`**.

Methods description:

- **`FormatDisplayValue`** – enables value formatting according current user settings. It complies precision and doesn't add unit name.

Code region
```
public string FormatDisplayValue(EUnitType UnitType, double Value);
public string FormatDisplayValue(string UnitName, double Value);
public string FormatDisplayValue(EUnitType UnitType, double Value, double
Power);
public string FormatDisplayValue(string UnitName, double Value, double
Power);
```

- **`Display`** – enables value recalculation from base unit to user unit. It's used mainly to set value in GUI.

Code region
```
public double Display(double BaseValue, EUnitType UnitType);
public double Display(double BaseValue, string UnitName);
public double Display(double BaseValue, EUnitType UnitType, int Power);
public double Display(double BaseValue, string UnitName, double Power);
```

- **`Base`** – enables value recalculation from user unit to base unit. It's used mainly to get value from GUI.

Code region
```
public double Base(double DisplayValue, EUnitType UnitType);
public double Base(double DisplayValue, string UnitName);
public double Base(double DisplayValue, EUnitType UnitType, int Power);
```

- **`Interface`** – enables value recalculation from base unit to application interface unit. It's used to set value in program API interface.

Code region
```
public double Interface(double BaseValue, EUnitType UnitType,
REXInterfaceType InterfaceType);
public double Interface(double BaseValue, string UnitName, REXInterfaceType
InterfaceType);
public double Interface(double BaseValue, string UnitName, string
InterfaceName);
public double Interface(double BaseValue, EUnitType UnitType, int Power,
REXInterfaceType InterfaceType);
public double Interface(double BaseValue, string UnitName, double Power,
REXInterfaceType InterfaceType);
public double Interface(double BaseValue, string UnitName, double Power,
string InterfaceName);
```

- **`Base`** – enables value recalculation from application interface unit to base unit. It's used to get value from program API interface.

Code region
```
public double Base(double InterfaceValue, EUnitType UnitType,
REXInterfaceType Interface);
public double Base(double DisplayValue, string UnitName, double Power);
public double Base(double InterfaceValue, string UnitName, REXInterfaceType
Interface);
public double Base(double InterfaceValue, string UnitName, string
InterfaceName);
public double Base(double InterfaceValue, EUnitType UnitType, double Power,
REXInterfaceType Interface);
public double Base(double InterfaceValue, string UnitName, double Power,
REXInterfaceType Interface);
public double Base(double InterfaceValue, string UnitName, double Power,
string InterfaceName);
```

- **`DisplayName`** – methods to get GUI unit name.

Code region
```
public string DisplayName(EUnitType UnitType);
public string DisplayName(string UnitName);
public string DisplayName(EUnitType UnitType, bool ForceReturn);
public string DisplayName(string UnitName, bool ForceReturn);
public string DisplayName(string UnitName, double Power);
public string DisplayName(EUnitType UnitType, int Power, bool ForceReturn);
public string DisplayName(string UnitName, double Power, bool ForceReturn);
```

- **`BaseName`** –methods to get the unit name for base units.

Code region
```
public string BaseName(string UnitName);
```

- **`BaseFromInterface`** – Method to Convert units from interface units to base units

Code region
```
public double BaseFromInterface(double InterfaceValue, EUnitType UnitType);
public double BaseFromInterface(double InterfaceValue, string UnitName,
double Power);
```

- **`Calculate`** – method to calculate the specified value from specified unit to another from one category.

Code region
```
public double Calculate(double Value, double Power, EUnitType UnitType,
string FromDefinition, string ToDefinition);
public double Calculate(double Value, double Power, string CategoryName,
string FromDefinition, string ToDefinition);
```

- **`CalculateFormBase`** – method to calculate the specified value from base to definition.

Code region
```
public double CalculateFromBase(double Value, double Power, string
ToDefinition);
```

- **`CalculateToBase`** – Method to calculate the specified value from definition to base.

Code region
```
public double CalculateToBase(double Value, double Power, string
FromDefinition);
```

- **`DisplayText`** –This method will return a full formatted value with unit name as text.

Code region
```
public string DisplayText(double DisplayValue, EUnitType UnitType, bool
Unit);
public string DisplayText(double DisplayValue, EUnitType UnitType, int Power,
bool Unit);
public string DisplayText(double DisplayValue, string UnitName, int Power,
bool Unit);
```

- **`DisplayTextFromBase`** – enables value recalculation from base unit to user unit. It's used mainly to set value in GUI. This method will return value with unit name as text.

Code region
```
public string DisplayTextFromBase(double BaseValue, EUnitType UnitType, bool
Unit);
public string DisplayTextFromBase(double BaseValue, EUnitType UnitType, int
Power, bool Unit);
public string DisplayTextFromBase(double BaseValue, string UnitName, int
Power, bool Unit);
```

8. SERIALIZATION

8.1. Data serialization

SDK classes ensure support for serialization of **`Data`** and **`Results`** class. Serialization can be provided for three types of objects (**`Data`** class description).

8.2. Serialization in Revit

The **`REXParameters`** class is responsible for serialization to Revit program object instances. This class is based on classes like **`Element`**. **`SaveToHost`** and **`LoadFromHost`** methods start serialization in **`Data`** and **`Results`** classes.

Description of **`REXParameters`** class main methods:

- **`SaveToHost`** – starts serialization, save to Revit element instance.

Code region

```
public bool SaveToHost(Autodesk.Revit.DB.Element Element);
public bool SaveToHost(Autodesk.Revit.DB.Element Element,
REXSystem.DataOperationType OperationType);
public bool SaveToHost(Autodesk.Revit.DB.Element Element,
REXSystem.DataOperationType OperationType, bool SaveData);
public bool SaveToHost(Autodesk.Revit.DB.Element element, string name, object
obj);
public bool SaveToHost(Autodesk.Revit.Element Element, string Name, Stream
Data);
```

- **`LoadFromHost`** – starts serialization, reads from element instance.

Code region

```
public object LoadFromHost(Autodesk.Revit.DB.Element element, string name);
public bool LoadFromHost(Autodesk.Revit.DB.Element Element,
REXSystem.DataOperationType OperationType, bool Header);
public bool LoadFromHost(Autodesk.Revit.DB.Element Element, string Name,
Stream Data);
public bool LoadFromHost(Autodesk.Revit.DB.Element Element);
public bool LoadFromHost(Autodesk.Revit.DB.Element Element, bool Header);
public bool LoadFromHost(Autodesk.Revit.DB.Element Element,
REXSystem.DataOperationType OperationType);
public bool LoadFromHost(Autodesk.Revit.DB.Element Element,
REXSystem.DataOperationType OperationType, bool Header);
```

- **`ClearHost`** – removes module data from element instance.

Code region

```
public bool ClearHost(Autodesk.Revit.DB.Element Element);
```

- **`SetHostId`** – sets parent object ID in child object. Additionally stores information in subobject about using it by current module.

Code region
```
public int SetHostId(Autodesk.Revit.DB.Element HostElement,
Autodesk.Revit.DB.Element Element);
```

- **GetHostId** – gets main object ID.

Code region
```
public int GetHostId(Autodesk.Revit.DB.Element Element);
```

- **GetHost** – gets the reference to main object.

Code region
```
public Autodesk.Revit.DB.Element GetHost(Autodesk.Revit.DB.Element Element);
```

- **SaveToProject** – starts serialization, save to Revit project.

Code region
```
public bool SaveToProject(REXSystem.DataOperationType OperationType);
```

- **LoadFromProject** – starts serialization, reads from project.

Code region
```
public bool LoadFromProject(REXSystem.DataOperationType OperationType);
```

- **RemoveFromProject** – removes Extension data from project.

Code region
```
public bool RemoveFromProject(REXSystem.DataOperationType OperationType);
```

8.3. File serialization

REXSystem class is responsible for writing data on a file. To access this class from **Extension** (**REXExtension**), the property **System** can be used. Methods **LoadFromFile** and **SaveToFile** start serialization in **Data** and **Results** classes.
These methods enable any implicit save to default file and to the file set as a parameter. It's also possible to save file using standard save dialog with manual setting of the path and file name.

8.4. Handling progress window

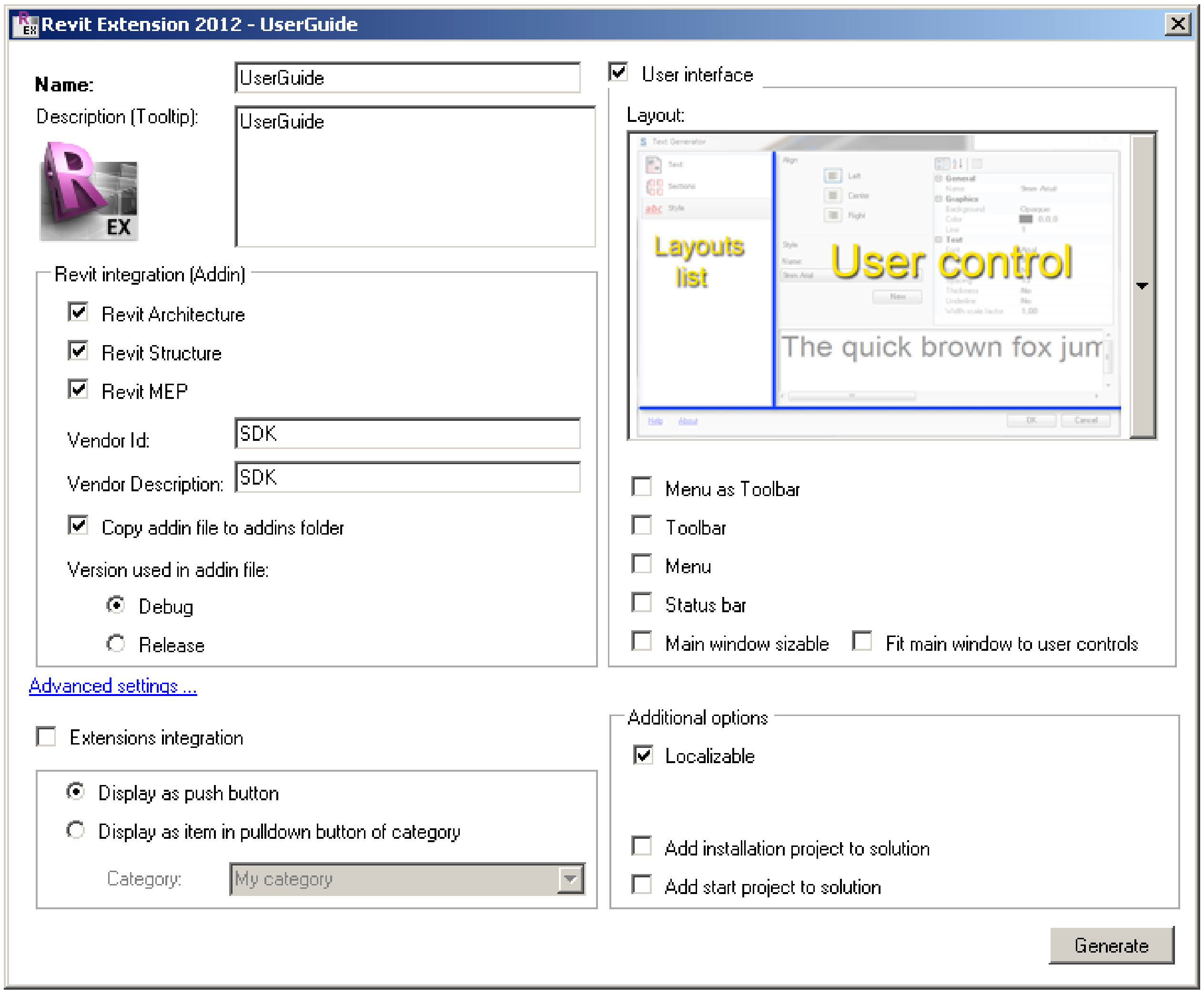
REX Framework provides common progress window. It's fully configurable and is accessible via Extension object.
Following methods are exposed:

Code region

```
void Hide();
void Show(int Steps);
void Show(object Parent);
void Show(object Parent, int Steps);
void Step();
void Step(string Text);
void Step(string Header, string Text);
void StepBack();
void StepBack(string Text);
void StepBack(string Header, string Text);
```

9. MULTI-LANGUAGE EXTENSION CREATION

As default the multi-language support is disable. It's possible to set this mode via the template wizard.


Revit Extension 2012 - UserGuide

Name: UserGuide
Description (Tooltip): UserGuide

Revit integration (Addin)
- [✓] Revit Architecture
- [✓] Revit Structure
- [✓] Revit MEP

Vendor Id: SDK
Vendor Description: SDK
- [✓] Copy addin file to addins folder

Version used in addin file:
- (●) Debug
- ( ) Release

Advanced settings ...

- [ ] Extensions integration
- (●) Display as push button
- ( ) Display as item in pulldown button of category

Category: My category

- [✓] User interface

Layout:

- [ ] Menu as Toolbar
- [ ] Toolbar
- [ ] Menu
- [ ] Status bar
- [ ] Main window sizable
- [ ] Fit main window to user controls

Additional options
- [✓] Localizable
- [ ] Add installation project to solution
- [ ] Add start project to solution

Generate

To work with many languages, appropriate directory tree is created according .NET rules.

Files containing translated resources should be placed in subdirectories named as culture in .NET (`CultureName`).

Property „Localizable" should be set true for every form and control language dependent. In such case all texts from controls are copied to resource file - ***.resx**.

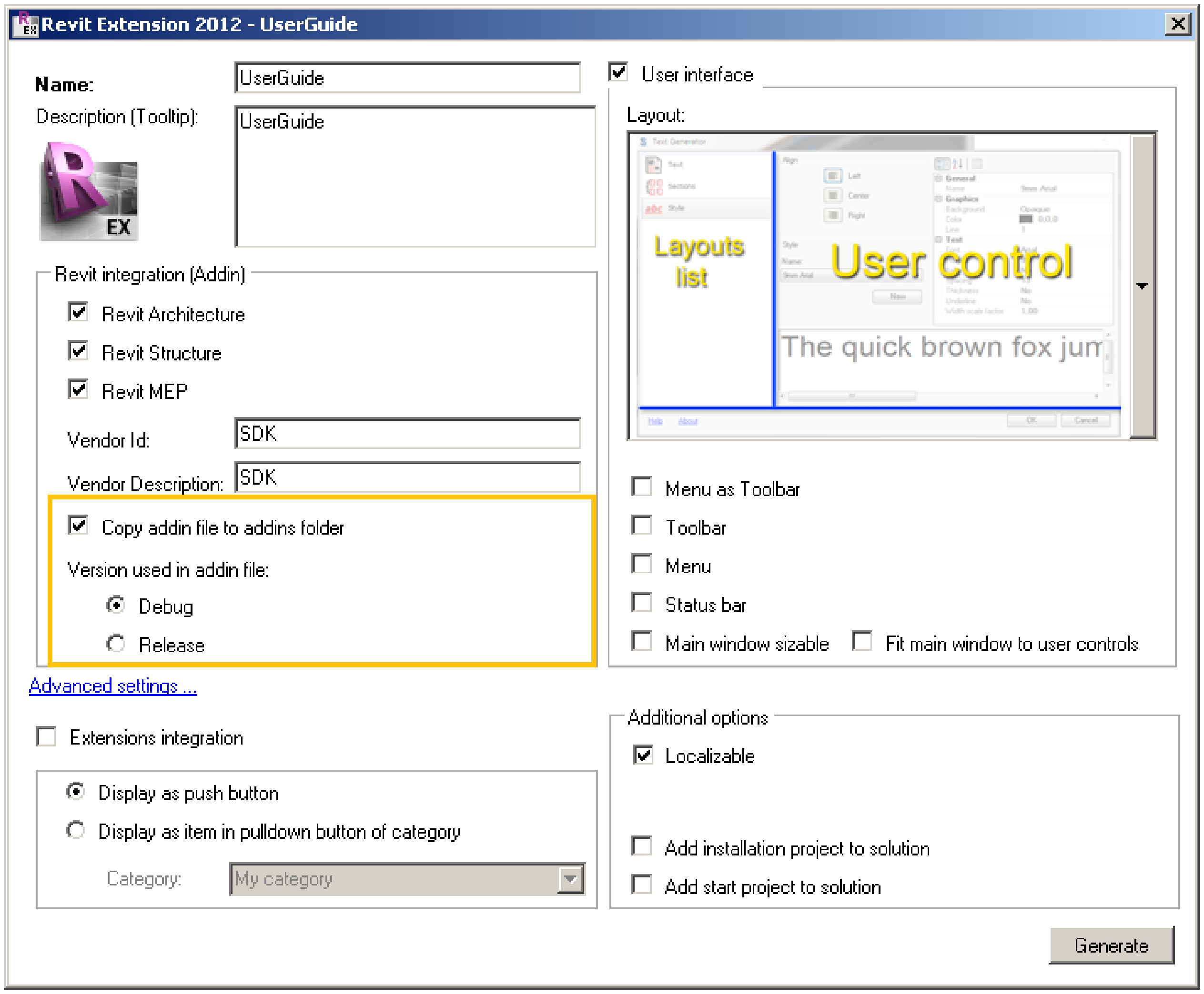
Sample structure of multi-language Extension:

UserExtension /	main module directory
en-US /	
UserModule.Resources.dll	language dependent resources file
UserModule.fr-FR.chm	module help file
de-DE /	
UserModule.Resources.dll	
UserModule.fr-FR.chm	
pl-PL /	
UserModule.Resources.dll	
UserModule.fr-FR.chm	
fr-FR /	
UserModule.Resources.dll	
UserModule.fr-FR.chm	
UserModule.dll	module file
UserModule.Png	image file linked with Extension

10. MODULE DEBUGGING

For debugging using VS 2010 it's necessary to set breakpoints in particular places of source and start main application.
Starting application can be set in project properties or it's possible to attach to it using menu command **Debug/Attach to process**.

Extension wizard generates an addin file for an easy start and debugging with Revit.



Revit Extension 2012 - UserGuide

Name: UserGuide

Description (Tooltip): UserGuide

Revit integration (Addin)

- [✓] Revit Architecture
- [✓] Revit Structure
- [✓] Revit MEP

Vendor Id: SDK

Vendor Description: SDK

- [✓] Copy addin file to addins folder

Version used in addin file:

- [✓ Debug] Release

Advanced settings ...

- [ ] Extensions integration
- [✓ Display as push button] Display as item in pulldown button of category

Category: My category

- [✓] User interface

Layout:

- [ ] Menu as Toolbar
- [ ] Toolbar
- [ ] Menu
- [ ] Status bar
- [ ] Main window sizable
- [ ] Fit main window to user controls

Additional options

- [✓] Localizable
- [ ] Add installation project to solution
- [ ] Add start project to solution

Generate

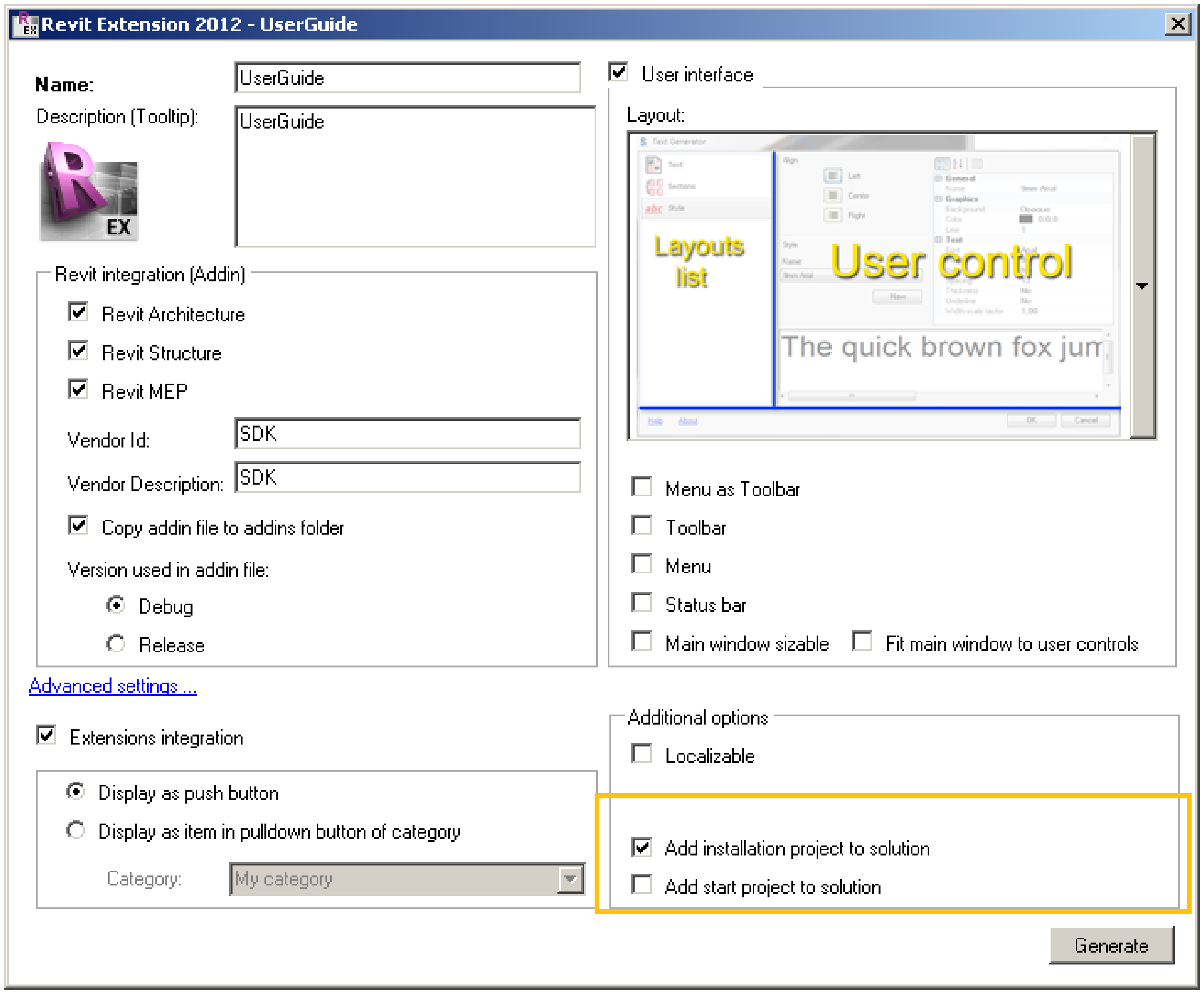
The addin file is copied per default in C:\Users\<current user>\AppData\Roaming\Autodesk\Revit\Addins\2012.

Thanks to this, Extension is fully configured and ready to be launched inside Revit

It's important to remember to copy compiled module files to module directory after compilation if the post build automation is removed.

File „**buildevents.bat”** can be used to manage copying too. This file is created during module creation and put in module directory. It's possible to modify it to copy all compiled files from output directory to module directory. This file is always launched after compilation.

11. INSTALLATION


Revit Extension 2012 - UserGuide

Name: UserGuide
Description (Tooltip): UserGuide

Revit integration (Addin)
[✓] Revit Architecture
[✓] Revit Structure
[✓] Revit MEP
Vendor Id: SDK
Vendor Description: SDK
[✓] Copy addin file to addins folder
Version used in addin file:
(●) Debug
( ) Release

Advanced settings ...

[✓] Extensions integration
(●) Display as push button
( ) Display as item in pulldown button of category
Category: My category

[✓] User interface
Layout:

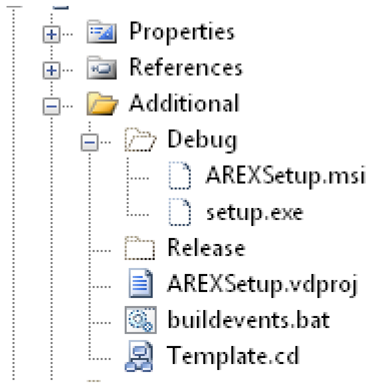
[ ] Menu as Toolbar
[ ] Toolbar
[ ] Menu
[ ] Status bar
[ ] Main window sizable [ ] Fit main window to user controls

Additional options
[ ] Localizable
[✓] Add installation project to solution
[ ] Add start project to solution

Generate

To deploy Extension, developer can use Extension Setup project included in the solution. This project is configured to install extension in External 2012 folder.

After build, installation is located in “Additional”Folder.


```
Properties
References
Additional
    Debug
        AREXSetup.msi
        setup.exe
    Release
    AREXSetup.vdproj
    buildevents.bat
    Template.cd
```

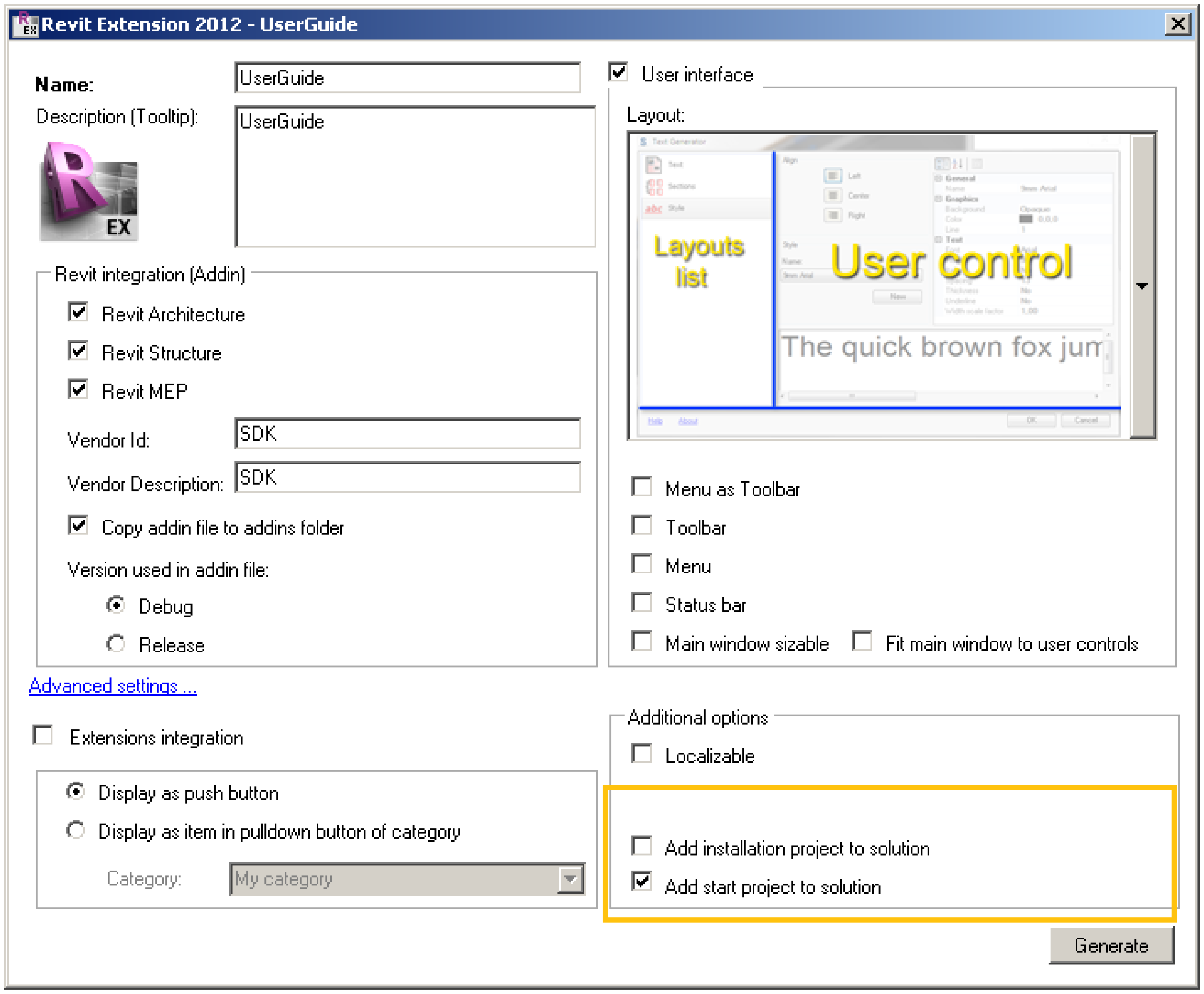
Developer can also right click on ExtensionSetup project and install or uninstall module from Visual Studio.

A manual or by a bat file copy paste of Extensions file is possible too.

Important note and limitation, the target machine should have Revit Extensions installed.

12. EXTENSION STARTER

The wizard allows you to generate a starter project.



Revit Extension 2012 - UserGuide

Name: UserGuide

Description (Tooltip): UserGuide

Revit integration (Addin)

- [✓] Revit Architecture
- [✓] Revit Structure
- [✓] Revit MEP

Vendor Id: SDK

Vendor Description: SDK

- [✓] Copy addin file to addins folder

Version used in addin file: [✓ Debug] Release

Advanced settings ...

- [ ] Extensions integration

[✓ Display as push button] Display as item in pulldown button of category

Category: My category

- [✓] User interface

Layout:

- [ ] Menu as Toolbar
- [ ] Toolbar
- [ ] Menu
- [ ] Status bar
- [ ] Main window sizable
- [ ] Fit main window to user controls

Additional options

- [ ] Localizable
- [ ] Add installation project to solution
- [✓] Add start project to solution

Generate

This project set as start project in your solution will give you the ability to test your dialog based application without Revit.

13. CONTENT GENERATOR

The REX.ContentGenerator is a component that provides tools for content management.

It allows converting Revit families to REX basic types and the other way round.
Advantages of the *REX.ContentGenerator*:

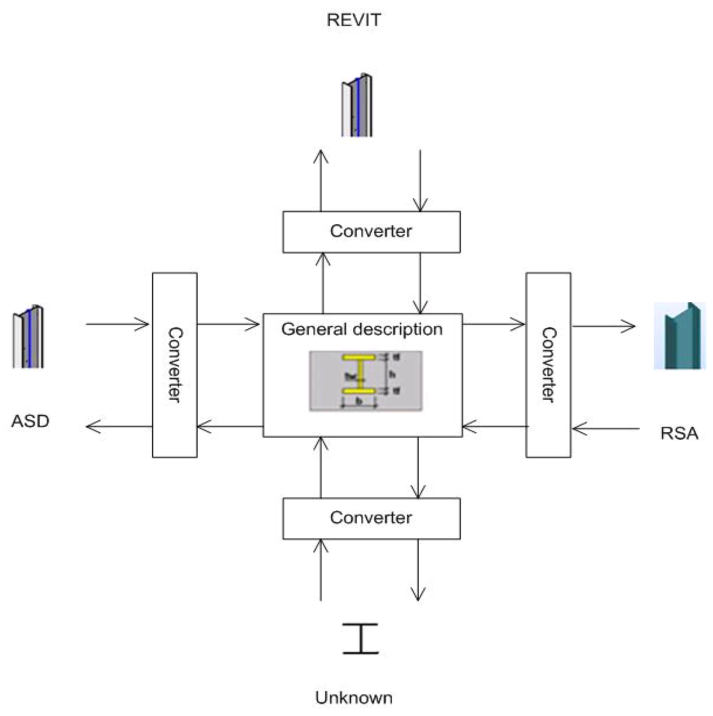
- Generation of Revit families.
- Interpretation of Revit families.
- Creating elements based on Extensions databases.
- Searching elements inside Extensions databases.
- Extensions databases browser.
- A mapping mechanism.

Before using of *REX.ContentGenerator* it is necessary to add following references to the project (with *Copy Local* = false property):

- REX.ContentGeneratorLT.dll
- REX.Geometry.dll

13.1. Architecture

Architecture of the *REX.ContentGenerator* is based on a `REXFamilyType` object which is the center point of the system.
This object is a data container independent of any product. Converters are facades to different products and have abilities to create a `REXFamilyType` from an internal representation as well as create products' elements based on the specific `REXFamilyType`.
Converters are separated from each other and they don't know anything about themselves.
All communication between them is ensured by the `REXFamilyType` object.



Elements are divided into:

- Certified (made with *REX.ContentGenerator*)
- Uncertified (not made by *REX.ContentGenerator*)

13.2. **Classes**

13.2.1. REXFamilyType

REXFamilyType is the center point of the *REX.ContentGenerator* component. It is a base class for all classes which represents different types of content. There are 8 main categories in *REX.ContentGenerator*:

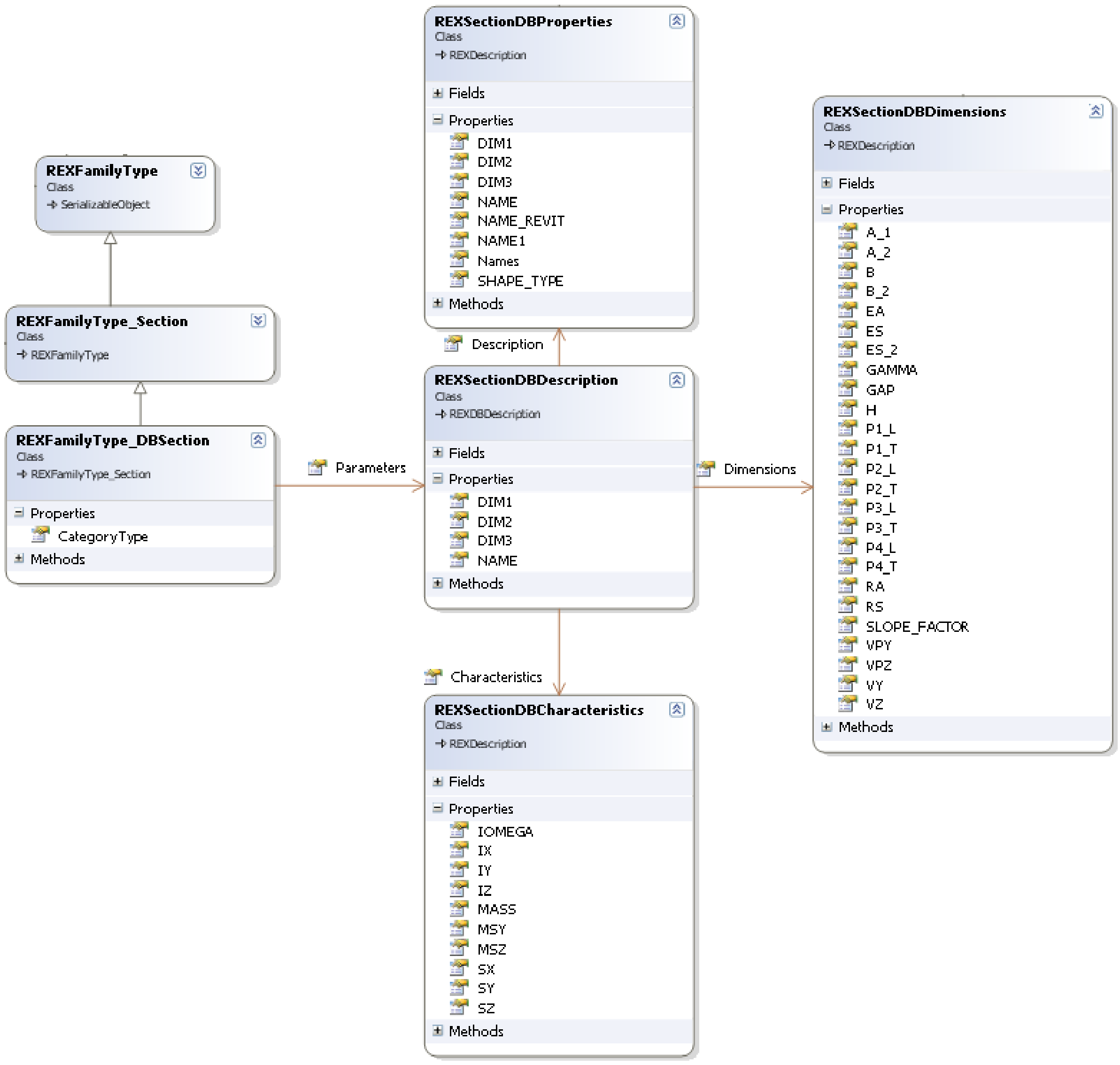
- Database sections
- Parametric sections
- Material

Properties

`ElementType`	The type of the element (beam, column etc.).
`Certified`	Indicates whether the element is certified.
`FamilyName`	The family name of the element. This name shouldn’t contain “ ’ ” in the end.
`Name`	The type name of the element. This name shouldn’t contain “ ’ ” in the end.
`Material`	The material of the element.
`Mapped`	If the element was mapped.
`UniqueID`	The unique ID of the element.
`CategoryType`	The category of the element.
`Description`	The object which describes parameters of the specific *REXFamilyType* in details.

13.2.2. REXFamilyType_DBSection

The section from a database is represented by the `REXFamilyType_DBSection` class. Its `Description` is of the `REXSectionDBDescription` type (there is a `Parameter` property which gives direct access to already casted `Description`).



13.2.3. REXSectionDBDescription

- **`Dimensions`** – describes dimensions of the section (according database convention: height, width etc.)

- `H`	The section's height.
- `B`	The section's width.
- `EA`	The web thickness.
- `ES`	The flange thickness.
- `RA`	The fillet radius (web).
- `RS`	The fillet radius (flange).

- `GAP`	The distance between sections in compound sections.
- `VY`	The extreme distance from the local Z axis of the point on the positive side of the Y axis.
- `VPY`	The extreme distance from the local Z axis of the point on the negative side of the Y axis.
- `VPZ`	The extreme distance from the local Y axis of the point on the negative side of the Z axis.
- `GAMMA`	The angle of rotation of the section.
- `B_2`	The AISC k value for design.
- `ES_2`	The AISC design value k.
- `P1_L`	The length of a slab P2 (cross-shaped section).
- `P1_T`	The thickness of a slab P1 (cross-shaped section)
- `P2_L`	The length of a slab P2 (cross-shaped section)
- `P2_T`	The Thickness of a slab P2 (cross-shaped section).
- `P3_L`	The length of a slab P3 (cross-shaped section).
- `P3_T`	The thickness of a slab P3 (cross-shaped section).
- `P4_L`	The length of a slab P4 (cross-shaped section).
- `P4_T`	The thickness of a slab P4 (cross-shaped section).
- `SLOPE_FACTOR`	The slope factor.
- `A_2`	The additional angle.
- `A_1`	The additional angle.

- **`Description`** – describes properties of the section (NAME, DIM1, DIM2, DIM3 etc.)

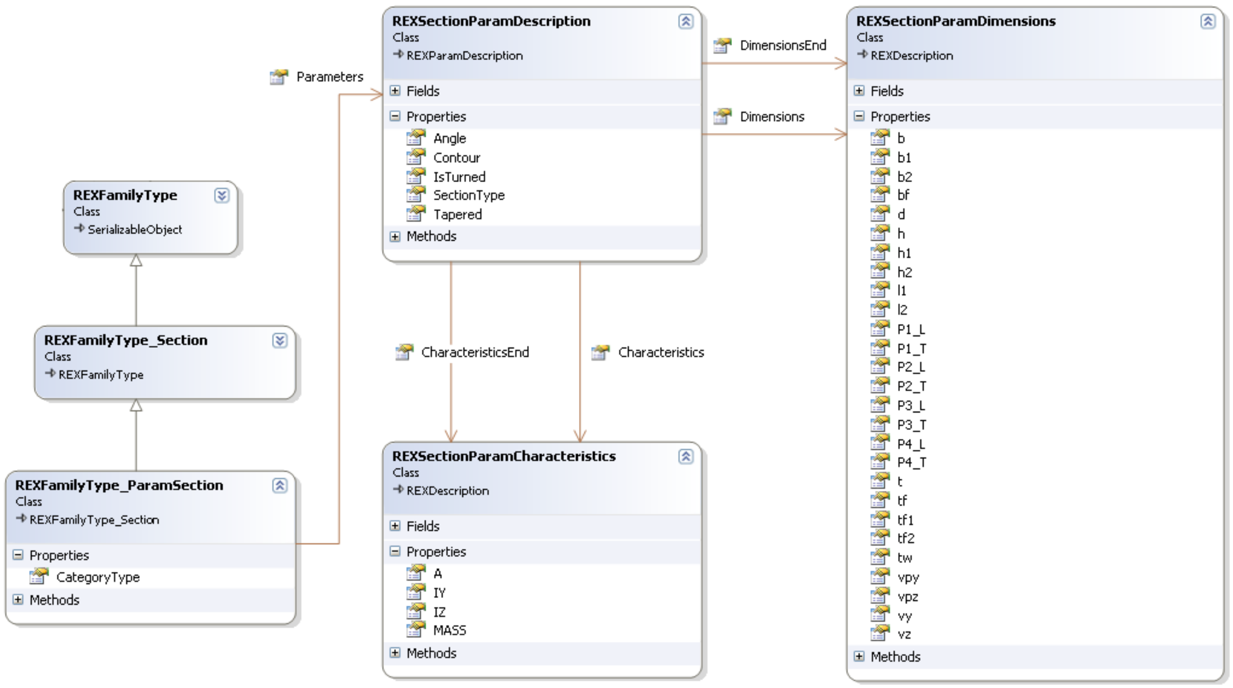
- `NAME_REVIT`	The Revit name. It is taken as a key for the identification process in a database when the element is not certified.
- `NAME`	The name. It is taken as key for the identification process in a database (with combination of: DIM1, DIM2, DIM3).
- `NAME1`	The section name.
- `DIM1`	The first numeric component of the section label. It is taken as a key for the identification process in a database (with combination of: NAME, DIM2, DIM3).
- `DIM2`	The second numeric component of the section label. It is taken as key for the identification process in a database (with combination of: NAME, DIM1, DIM3).
- `DIM3`	The third numeric component of the section label. It is taken as a key for the identification process in a database (with combination of: NAME, DIM1, DIM2).
- `SHAPE_TYPE`	The number of the section shape type.
- `Names`	The list of optional names.

- **`Characteristics`** – describes characteristics of the section (according to the database convention: moment of inertia, cross-sectional area etc.)

- `SX`	The cross-sectional area.
- `SY`	The reduced section area for XY-shear deformation calculations - considering the influence of shear forces along Y axis.
- `SZ`	The reduced section area for XZ-shear deformation calculations - considering the influence of shear forces along Z axis.
- `IX`	The torsional constant.
- `IY`	The moment of inertia (Iy).
- `IZ`	The moment of inertia (Iz).
- `IOMEGA`	The warping constant (for thin-walled sections).
- `MASS`	The weight per length unit of a section.
- `MSY`	The plastic section modulus (bending) about the Y axis.
- `MSZ`	The plastic section modulus (bending) about the Z axis.

13.2.4. REXFamilyType_ParamSection

The parametric section is represented by the `REXFamilyType_ParamSection` class. Its `Description` is of the `REXSectionParamDescription` type (there is a `Parameters` property which gives direct access to the already casted `Description`).

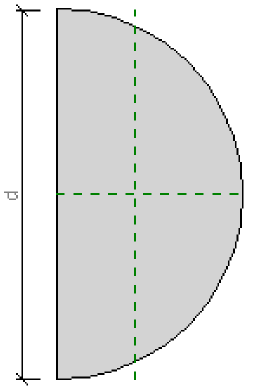
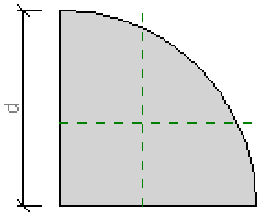
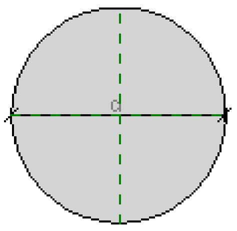
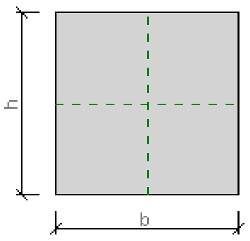
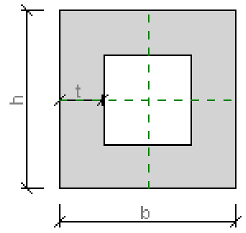
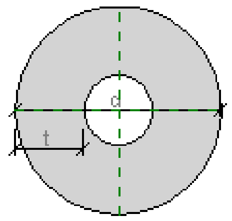
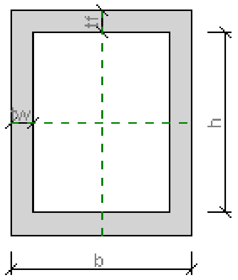
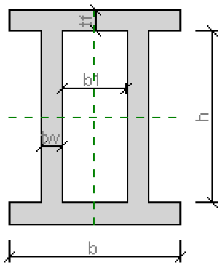
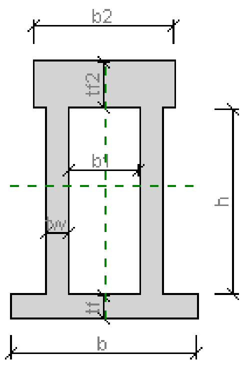
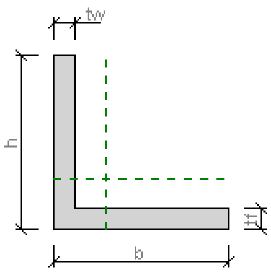
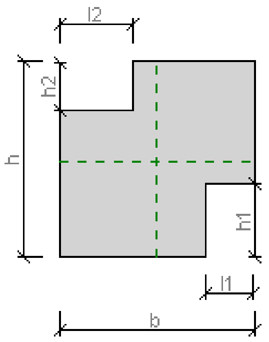
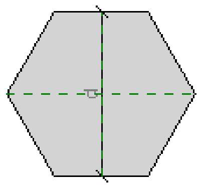
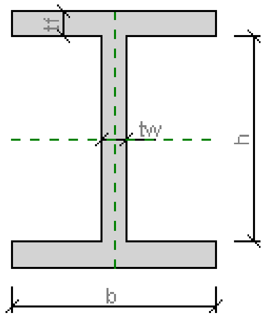
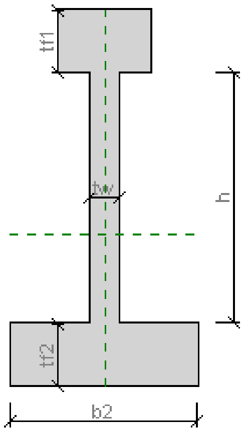
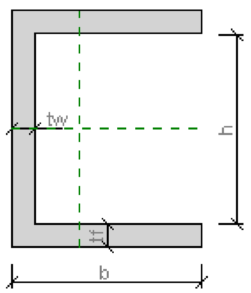
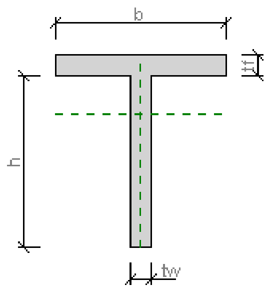


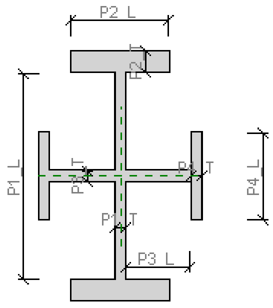
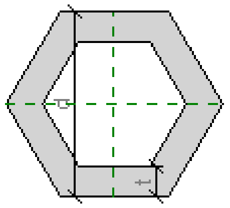
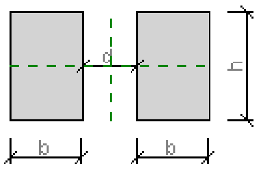
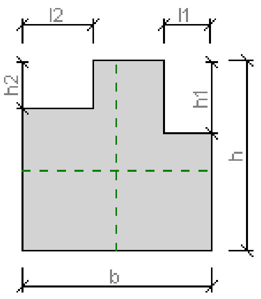
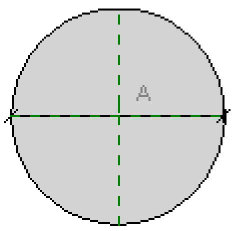
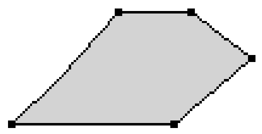
13.2.5. REXSectionParamDescription

- Properties:

- `Angle`	The angle of the section.
- `Contour`	The contour (only for UNKNOWN type).
- `IsTurned`	Indicates whether the section is turned.
- `Tapered`	Indicates whether the section is tapered.
- `SectionType`	The type of the section.
- `Dimensions(End)`	Dimensions of the section (on end).

- Section types:

			
ROUNDH	**ROUNDQ**	**ROUND**	**RECT**
			
RECT_HOLLOW	**TUBE**	**BOX1**	**BOX2**
			
BOX3	**L**	**Z**	**POLYG**
			
I	**IASYM**	**C**	**T**

 P2 L P2 T P1 L P1 T P4 T P4 L P3 T P3 L			
CROSS	**POLYG_HOLLOW**	**DRECT**	**TASYM**
			
CABLE	**UNKNOWN**		

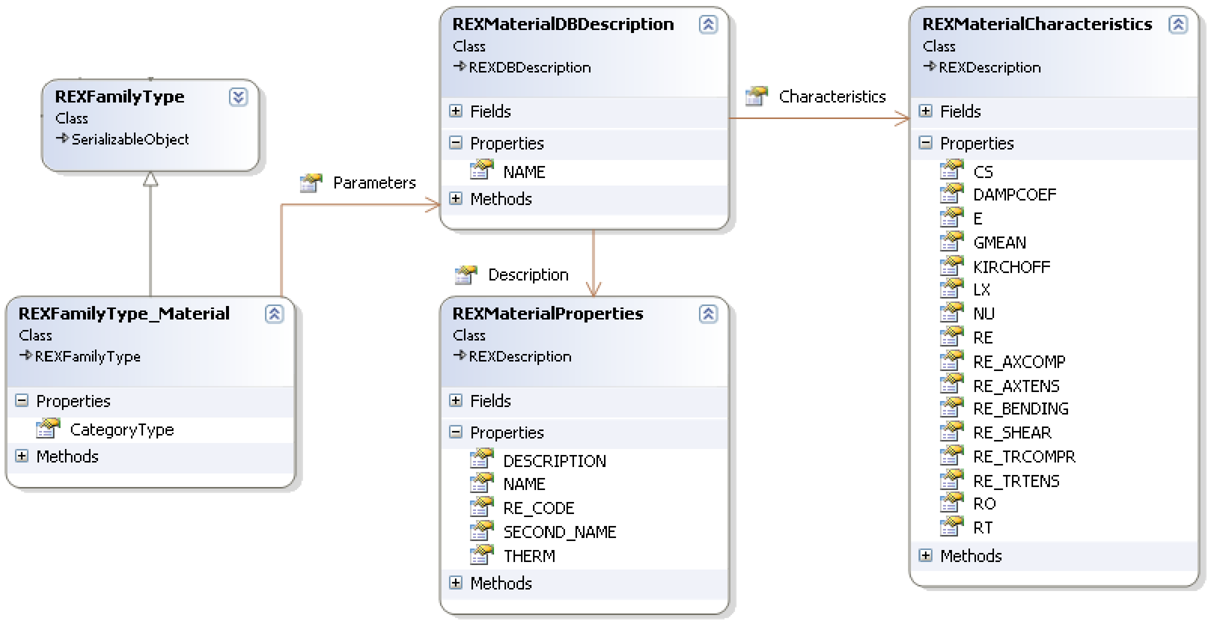
- **Characteristics(End)** Characteristics of the section (end)

- IY	The moment of inertia about the Y axis.
- IZ	The moment of inertia about the Z axis.
- A	The Cross-section area.
- MASS	The nominal weight per unit length.

- Methods**:**
  - `CalculateMainAxisAndCharacteristics`
    Calculates the main axis and characteristics of the section.
  - `CalculateCharacteristics`
    Calculates characteristics of the section.
  - `CalculateMainAxisParams`
    Calculates the main axis of the section.
  - `GetContour`
    Returns the contour of the section.

13.2.6. REXFamilyType_Material

The material is represented by the `REXFamilyType_Material` class. Its `Description` is of the `REXMaterialDBDescription` type (there is a `Parameter` property which gives direct access to already casted `Description`).



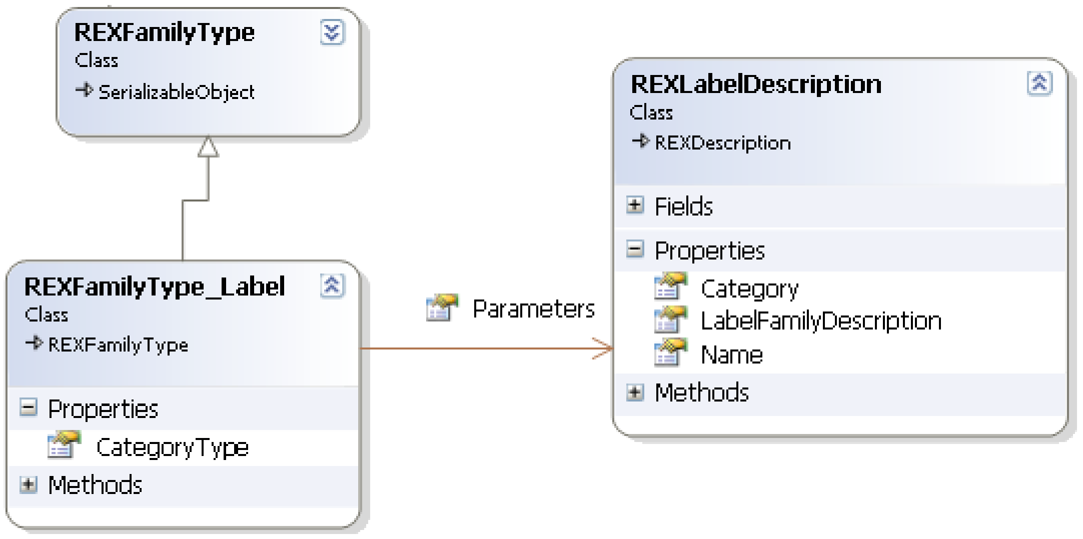
13.2.7. REXMaterialDBDescription

- Description The material description.
  - NAME The material name. It is taken as a key for the identification process in a database.
  - SECOND_NAME The material's name (used for a steel to give an equivalent according to Eurocode).
  - DESCRIPTION The detailed material description.
  - THERM The material type /for steel: thermal treatment-1 /for timber: natural-0, glued-1, glued KertoS specific for Kerto code-2, glued KertoQ specific for Kerto code-3, glued KertoS-4, glued KertoQ-5.
  - RE_CODE The resistance.

- *Characteristics* The characteristics of the material.
  - E Young's modulus
  - NU Poisson's ratio.
  - LX The thermal expansion coefficient.
  - RO The unit weight.
  - RE The design strength.
  - RE_AXCOMP The axial compression resistance (for timber).the reduction factor for shear for steel / Burning velocity (mm/min) (data specific for timber design according to CB71).
  - RT The design tensile strength.
  - RE_BENDING The bending strength (for timber).
  - RE_AXTENS The axial tension resistance (for timber).</
  - RE_TRTENS The transversal tension resistance (for timber).
  - RE_TRCOMPR The transversal compression resistance (for timber).
  - RE_SHEAR The shear strength (for timber).
  - DAMPCOEF The damping coefficient.
  - GMEAN The average modulus G.
  - KIRCHOFF The shear modulus G.

13.2.8. REXFamilyType_Label

A label represents any object of any type. This type was created for mapping purpose.
It is described by name and category of the element. The label is represented by the `REXFamilyType_Label` class.
Its Description is of the `REXLabelDescription` type (there is a `Parameter` property which gives direct access to already casted `Description`).



- **<u>`REXLabelDescription`</u>**
  - `Category` The category.
  - `Name` The name.
  - `LabelFamilyDescription` Gets or sets the label family description. Label represents any type of content. In the LabelFamilyDescription object can be placed element which is represented by the label (for mapping mainly - to draw sections).

13.3. Converters

The element which decides about interpretation of the *REXFamilyType* in specific context is converter. There are two defined converters inside *REX.ContentGenerator* component:

`RVTFamilyConverter`	Responsible for the Revit context
`REXFamilyConverter`	Responsible for the REX context (databases mainly)

13.3.1. RVTFamilyConverter

The `RVTFamilyConverter` is a façade to the Revit document. It allows to:

- Gets the Revit element based on the `REXFamilyType` object.
- Gets the `REXFamilyType` object based on the Revit element.

- Methods:
  - `InitGlobal` Initializes lists of existing families from the Active Document.
  - `GetFamily` Gets the *REXFamilyType* object from the input element.
  - `GetElement` Gets the Revit element created based on the *REXFamilyType* object.
  - `GetElement` Gets the Revit element created based on the *REXFamilyType* object.
  - `GetElements` Gets the list of Revit elements created based on the list of

- *REXFamilyTypes*:
  - `GetCertifiedCategoryOfElement` Gets the category of the certified element. If the element is not certified the UNKNOWN result will be returned.
  - `Close` Saves data to the Revit document, clears collections, releases objects.
  - `GetExistingNames` Returns the list of names existing for the specific *REXFamilyType* object.

- `Settings`:
  - `SetLanguage` Sets language for created families (names of families will be translated according to this) - “en-US”, “pl-PL” etc. (string lang);
  - `Clean` Cleans temporary files.
  - `Update` If elements should be updated every time.
  - `IgnoreFamilyNameLanguage` If languge should be taken to consideration during element search (language set by SetLanguage function). If set to false, there will be the same elements in the project but of different family names (e.g. one in english - *I-sections with wide flanges* and one in french - *En I à ailes larges*).
  - `CreateNewElement` Indicates whether a new family should be created if the element isn’t found among existing families.
  - `Labels` Indicates whether labels should be taken to consideration when mapping.
  - `BufferElements` Indicates whether elements should be buffered after investigation – using RevitId. If false it will be analyzed each time.

13.3.2. REXFamilyConverter

The `REXFamilyConverter` is a façade to REX databases:

- Gets the database records based on the `REXFamilyType` object.
- Gets the `REXFamilyType` object based on the database record.

- `GetDBList` Gets the list of database's records adequate for the input object.
- `GetFamily` Gets the *REXFamilyType* object based on the specified database's record.
- `GetFamilies` Gets the list of all elements in the selected database.
- `GetFamilyTemplate` Gets the dedicated template of the *REXFamilyType* (with initial data).
- `GetEmptyFamilyOfAlias` Gets the empty family of alias name. It doesn't have any data inside despite indentification and it can be used to take proper data from database.
- `GetFamilyDefault` Gets the default *REXFamilyType* of the specific type. It has all data filled with some default values (e.g. some default material with parameter is taken)
- `GetNameForAlias` Gets the certified name for the alias.
- `FindFamily` Finds the *REXFamilyType* in the specific list.
- `Close()` Clears collections, releases COM objects. It should be called before the end of the converter usage.
- `GetFamilyDBSection` Gets the section of the specific name (based on data from the database).
- `GetFamilyMaterial` Gets the material of the specific name (based on data from the database).
- `GetDatabaseDirectory` Gets the directory for database of specific category.
- `GetUserDatabaseDirectory` Gets the directory for the user database of specific category.
- `GetDatabasePath` Gets the path for specific database.

- `Settings`
  - `MultiSearch` Indicates whether multiply results of database search should be returned (if false - when first result is found algorithm will stop).
  - `DatabaseAccess` The database access.
  - `MultiThread` Indicates whether multithread operations are required.
  - `ReleaseDBAfterEachAction` Indicates whether the database should be released after each action (otherwise it will be kept in memory).
  - `GetParamsForDisplayName` Gets parameters for the specific display name of specific category.
  - `GetParamsForField` Gets parameters for the specific name of specific category.
  - `GetResource` Gets the resource for specific name of specific category.
  - `SupportUserDatabases` Indicates whether user’s databases are supported.

13.4. Filters

Filters are used for selecting these elements from a database which respect some additional constraints. There are two types of filters:

- Constraint of the single parameter (`REXContentParamFilter`)
- Logic filter (`REXContentLogicFilter`)

- **`REXContentParamFilter`**

`REXContentParamFilter` describes a single constraint for a single parameter (e.g. height of a section).

- Properties:
  - `More` Gets or sets a value indicating whether a parameter is to be larger than the constraint value.
  - `Less` Gets or sets a value indicating whether a parameter is to be smaller then the constraint value.
  - `Equal` Gets or sets a value indicating whether a parameter is to be equal to the constraint value.
  - `ParamName` Gets or sets the name of the parameter which is constrained.
  - `ConstraintValue` Gets or sets the constraint value.

- **`REXContentLogicFilter`**

`REXContentLogicFilter` allows to define a compound constraint.

- Properties:
  - `Filters` The list of filters (among them might be `REXContentParamFilters` and `REXContentLogicFilters`.
  - `LogicOperator` The logic operator.

13.5. Custom factories

The goal of the custom factory is to provide ability to use own categories which aren't included in the *REX.ContentGenerator* component.

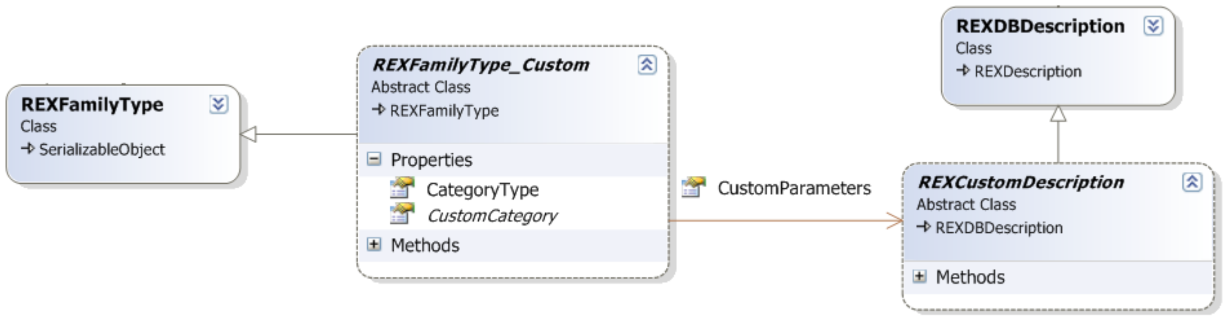
Implementation of a custom category demands following steps:
- Definition of common, consistent name.
- Implementation of a `REXFamilyType_Custom` class.
- Preparation of family templates.
- Preparation of configuration files (xml).
- Preparation of databases (xml) – if necessary.
- Registration of new factories inside REX and RVT converters.

- *Definition of the common, consistent name*

The custom category is identified by its unique name. It has to be consistent and common in all places (code, configuration files, databases). It has to be different to:
- DBSection
- ParamSection
- Material

- *Implementation of a REXFamilyType_Custom class*

  The custom architecture is based on the `REXFamilyType` object (like whole component). For this purpose there were created two abstract classes:
  `REXFamilyType_Custom`
  `REXCustomDescription`



13.5.1. REXFamilyType_Custom

- The `REXFamilyType_Custom` is an abstract class.
- All custom `REXFamilyTypes` have to be derived from it.
- It has to have parameterless constructor defined (it is used as generic class inside *REX.ContentGenerator* and new element is created by a `T()` constructor).
- It has to initialize its description of the type derived from `REXCustomDescription` class.

- <u>Properties:</u>
  - `CustomCategory` The name of the category for identification (e.g. "CustomSection") – has to be implemented

- `CustomParameters` The description of `REXFamilyType` (derived from `REXCustomDescription` class)

- Methods:
  - `Clone` Creates a new object that is a copy of the current instance – has to be implemented

13.5.2. REXCustomDescription

- `REXCustomDescription` is an abstract class.
- All custom `REXDescriptions` have to be derived from it.
- It has to have parameterless constructor defined (it is used as generic class inside *REX.ContentGenerator* and new element is created by a `T()` constructor).

- Methods:
  - `Clone` Creates a new object that is a copy of the instance – has to be implemented
  - `GetTypeName` Returns the name for the current type (e.g. this name will be set in Revit as FamilySymbol name)

Parameters of the custom object are taken from `REXCustomDescription`. *REX.ContentGenerator* iterates threw all properties of the class and gets or sets appropriate values. `REXCustomDescription` may have a flat structure (all properties are defined directly in it):

Code region
```
class FootingDescription:REXCustomDescription
{
       public double h { get; set; }
       public double b { get; set; }
       public double w { get; set; }
}
```

It can be also organized in more complex way with embedded objects which classes are derived from `REXDescription`:

Code region
```
class FootingDescription:REXCustomDescription
{
        public FootingDimensions Dimensions { get; set; }
}
class FootingDimensions:REXDescription
{
        public double h { get; set; }
        public double b { get; set; }
        public double w { get; set; }
}
```

- *Preparation of family templates*.

The custom factory is based on Revit `FamilyInstance`, `Family` and `FamilySymbol` classes. Other types aren't taken to consideration. The *REX.ContentGenerator* creates new `FamilySymbols` using rfa files.

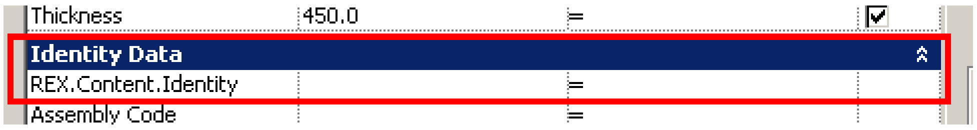
REX.ContentGenerator doesn't make them by itself and they have to be provided by user. The idea is to prepare family which will be a template for families of the same type.

For instance if a user want to create footing category which would contain rectangle and trapezoid shape, he would have to prepare two template files:

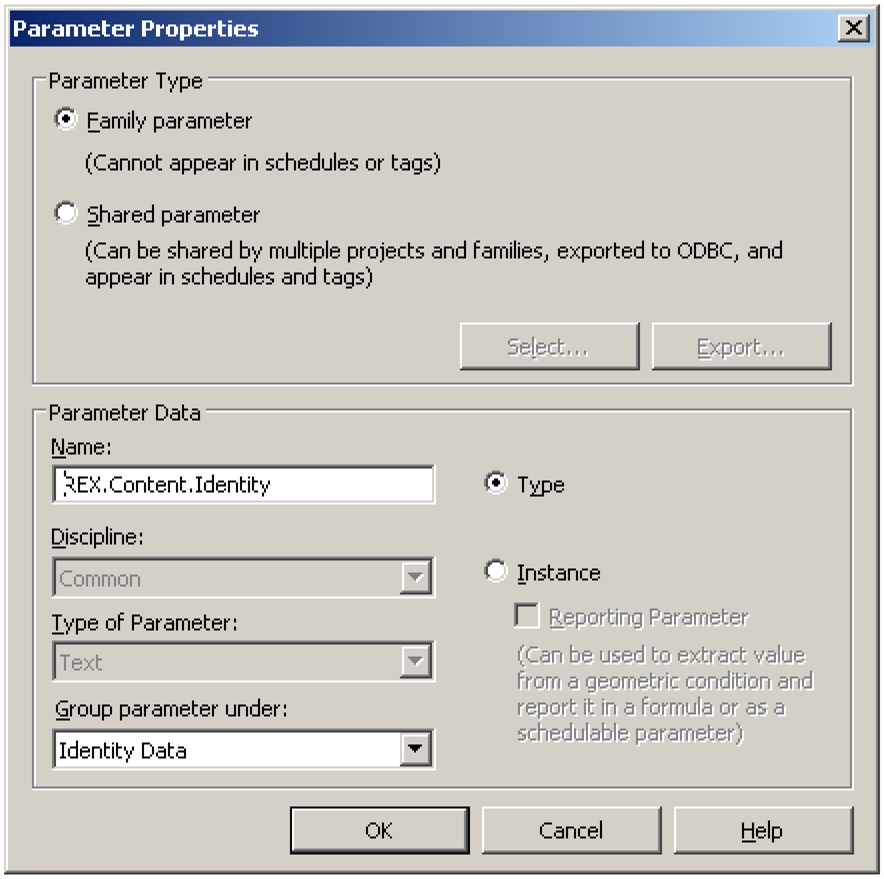
- Template_Rectangle.rfa
- Template_Trapezoid.rfa

Templates can be prepared in a free way. The standard of *REX.ContentGenerator* demands only two things:

- There have to be added **REX.Content.Identity** parameter:



Thickness	450.0	=	✓
Identity Data			
REX.Content.Identity		=	
Assembly Code		=	



Parameter Properties

Parameter Type

[✓ Family parameter]
(Cannot appear in schedules or tags)

Shared parameter
(Can be shared by multiple projects and families, exported to ODBC, and appear in schedules and tags)

Select... Export...

Parameter Data

Name: REX.Content.Identity

[✓ Type] Instance

Discipline: Common

Reporting Parameter
(Can be used to extract value from a geometric condition and report it in a formula or as a schedulable parameter)

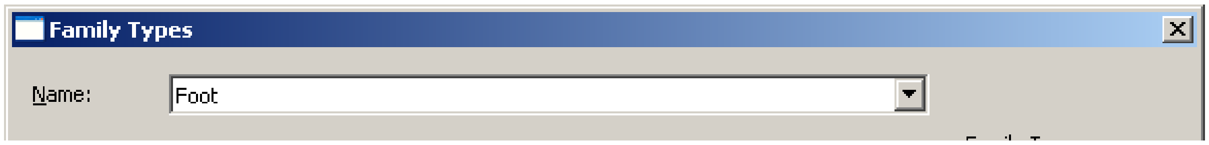
Type of Parameter: Text

Group parameter under: Identity Data

OK Cancel Help

All identity data is stored inside this parameter and it has to be added. Otherwise the system will work incorrectly.

- There have to be defined one default type inside an rfa file. This type will be loaded to the project and used as a pattern for the new element.



Name: Foot

- *Preparation of configuration files*

There are two configuration files which have to be prepared:

- Parameters.xml – This file provides definition for all parameters (names, units, powers, functions etc.)
- Templates.xml – This file defines relations between templates and types (identified by parameters defined in Parameters.xml file)

<u>Parameters</u>

Each parameter has to have provided definition. Otherwise it will be impossible to convert it in a correct way. The example of parameters file is presented below:

Code region
```
<?xml version="1.0" encoding="utf-8"?>
<Categories>
  <Category name="CustomFooting">
    <Field name="b" unit="UC_SectionDim" power="1" display="Length" />
    <Field name="w" unit="UC_SectionDim" power="1" display="Width"  />
    <Field name="h" unit="UC_SectionDim" power="1" display="Thickness"  />
    <Field name="name" unit="UC_Text" visible="true" identity="true" 
display="name" />
    <Field name="FootingType" unit="UC_Text" template="true" identity="true" 
database="type" />
  </Category>
</Categories>
```

- Elements

`Category` - Contains list of parameters defined for a specific category

Attributes:

`name` Indicates the name of the custom category

`Field` - Contains definition for the specified parameter

Attributes:

- `name` The name of the property defined in REXCustomDescription
  In case of complex structure (with embedded REXDescription types) the whole root has to be defined here:
  `<Field name="Dimensions.b" unit="UC_SectionDim" power="1" display="Length" database="b"/>`
- `unit` The unit category of the parameter according REX categories

Name	Base Unit
UCG_Dimensions	
UC_Length	m
UC_Angle	Rad
UC_SectionDim	m
UC_StructureDim	m
UC_SectionChar	m
UC_SectionDiam	m

Name	**Base Unit**
UC_Displacement	m
UC_SecDimDistLen	m2/m
UC_DispDistLen	m/m
UC_AngleLen	Rad/m
UCG_Forces	
UC_Force	N
UC_ForceDistLen	N/m
UC_ForceDistSurf	N/m2
UC_ForceDistVol	N/m3
UC_SurfForce	m2/N
UC_Moment	N*m
UC_Stress	N/m2
UC_Pressure	N/m2
UC_MomentDistLen	(N*m)/m
UCG_MassWeight	
UC_Mass	kg
UC_MassDens	kg/m
UCG_Time	
UC_Time	s
UC_Speed	m/s
UC_Accel	m/s2
UC_Flux	m2/s
UC_Frequency	Hz
UC_FrequencyPolar	s-1
UC_SpeedPolar	Rad/s
UC_AccelPolar	Rad/s2
UCG_Energy	
UC_Energy	J
UC_Power	J
UC_PowerTime	J*h
UC_PowerFlux	J*m2
UC_ImpactRes	J/m2
UCG_Temperature	
UC_Temp	DegC
UC_TempGradient	DegC/m
UCG_Other	
UC_Percent	%
UC_Coefficient	
UC_Number	
UC_Text	
UCG_Cost	
UC_CostTimeWork	w-h
UC_CostTimeMachine	m-h
UC_CostCurr	$ (L, EUR, zł)
UC_CostCurrDivLen	$/m
UC_CostCurrDivArea	$/m2
UC_CostCurrDivVol	$/m3
UC_CostCurrDivMass	$/m
UC_CostCurrDivTimeWork	$/w-h
UC_CostCurrDivTimeMachine	$/m-h

- `power` The power of the parameter.
- `display` The name of the parameter in Revit (thanks to this parameter *REX.ContentGenerator* knows that property `w` should be mapped to the Revit parameter “Width”). If the “display” attribute is not set the “name” attribute is taken directly.

- `database` The name of the parameter in the database (thanks to this parameter *REX.ContentGenerator* knows that property `FootingType` should be mapped to the database parameter "type").

- `template` This attribute indicates parameters which are taken to consideration during template search inside the document. If already created `FamilySymbol` has the same parameters (which are identified by the template attribute) as input one, then it can be taken as template and duplicated. Beside these parameters there are checked: `FamilyName`, `TypeName` and `DBName` (if empty or null they aren't taken to consideration). Template parameters don't have to be added to the rfa file. They are stored in the REX.Content.Identity parameter. This attribute is also taken to consideration in Templates.xml which will be described later.
-
- `identity` This attribute indicates parameters which are taken to consideration during element search inside the document. If already created `FamilySymbol` has the same parameters (which are identified by the identity attribute) as input one, then it is treated as the same one. Beside these parameters there are checked: `FamilyName`, `TypeName` and `DBName` (if empty or null they aren't taken to consideration). Identity parameters don't have to be added to the rfa file. They are stored in the REX.Content.Identity parameter.

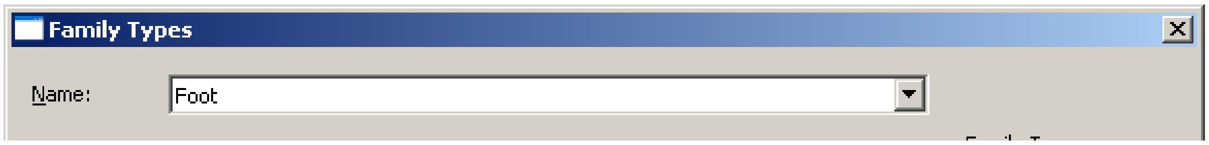
Templates

REX.ContentGenerator has to have defined relations between rfa files and custom types:

Code region
```xml
<?xml version="1.0" encoding="utf-8"?>
<Categories>
  <Category name="CustomFooting">
    <Family file ="footing.rfa" type="Foot" description ="My footing”"
FootingType="rect" />
  </Category>
</Categories>
```

Elements

`Category`	Contains list of parameters defined for the specified category	
	Attributes:	
	`name`	indicates the name of the custom category
`Family`	contains definition for the specified type	
	Common attributes:	
	`file`	the template file rfa
	`type`	the type name defined in rfa file



`description`	the family name of new created element (it can be also defined in REXFamilyType as FamilyName)

Template attributes

Template attributes indicates for what kind of element specific template is assigned. This attributes are taken from Parameters.xml file (parameters with `template="true"` attribute). In the example above `FootingType` is a such parameter.

Preparation of databases

REX.ContentGenerator provides its own format of databases which can be used by a user

Code region
```
<?xml version="1.0" encoding="utf-8"?>
<Database category="CustomFooting">
  >
  <Units>
    <Unit type="UC_SectionDim" symbol ="mm"/>
  </Units>
  <Families>
    <Type name="Type1" b="5000" h ="3000" w="2000" type="rect" />
    <Type name="Type2" b="1000" h ="1000" w="1000" type="rect" />
  </Families>
</Database>
```

Elements

`Units` contains definition for units used in the specified database. At the moment following units can be used:

Group Name /Unit	Coefficient /Formula
UG_Length	
mm	0.001
cm	0.01
m	1
km	1000
in	0.0254
ft	0.3048
yd	0.9144
mile	1609.344
UG_Force	
N	1
daN	10
kN	1000
MN	1000000
kG	9.80665
T	9806.65
lb	4.448222
kip	4448.222
UG_Angle	
Rad	1
Deg	0.01745329251994
Grad	0.01570796326795
UG_Temperature	
DegC	1
DegF	(%x - 32)/1.8
K	%x – 273
UG_Mass	

Group Name /Unit	Coefficient /Formula
G	0.001
kg	1
t	1000
lb	0.453592
oz	0.02834952
ton	907.184
UG_Time	
s	1
min	60
h	3600
day	86400
week	604800
year	31536000
UG_Power	
W	1
kW	1000
MW	1000000
hp	745.700
UG_Energy	
J	1
daJ	10
kJ	1000
MJ	1000000
cal	4.186800
UG_Frequency	
Hz	1
kHz	1000
MHz	1000000
UG_Percent	
%	1
UG_Stress	
Pa	1
hPa	100
kPa	1000
MPa	1000000
psf	47.88026
psi	6894.75789
ksi	6894757.89
bar	100000
UG_CostTimeWork	
w-h	1
UG_CostTimeEquipment	
m-h	1
UG_Currency	
$	1
L	1
EUR	1
zł	1

Attributes

`type`	the unit category according the REX standard
`symbol`	the symbol of the unit (m, cm, m etc)

`Families`	contains all types defined in the database.
`Type`	describes specific type with all parameters characteristic for it.

Registration of new factories inside REX and RVT converters

When all elements are prepared it is necessary to register custom factories in dedicated converters.

`RVTFamilyConverter`

Custom factory is registered by RegisterCustomFactory method:

Code region

```
rvtConverter.RegisterCustomFactory<CustomFamily, CustomFamilyDescription>(
CustomFooting.CustomCategoryName, @"..\Parameters.xml", @"..\Templates.xml",
@"..\footingFamilies", BuiltInCategory.OST_StructuralFoundation);
```

`REXFamilyConverter`

Custom factory is registered by RegisterCustomFactory method:

Code region

```
rex.RegisterCustomFactory<CustomFooting,CustomFootingDescription>(
CustomFooting.CustomCategoryName, @"…\Parameters.xml");
```

In case of RVTFamilyConverter as well as REXFamilyConverter Parameters file can be reloaded by `LoadCustomParametersFile` in theirs settings.

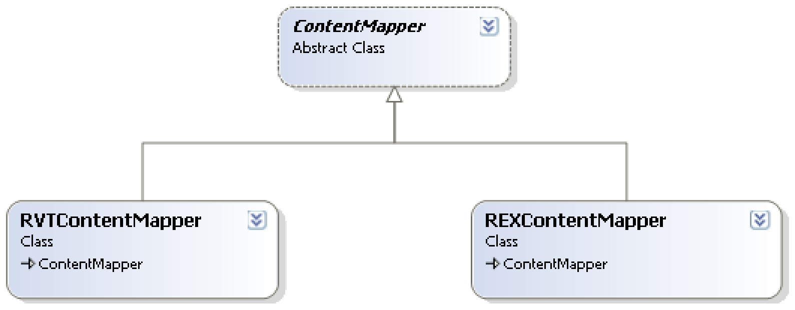
13.6. Mapping

The idea of mapping is to provide ability to map the element to a database element or a label defined by a user. The mapping system provides a dialog where user is able to define mapping relations. The access to the mapping system is assured by a `ContentMapper` objects in `RVTFamilyConverter` and `REXFamilyConverter`.
The mapping system is based on two files:

- Global – there is one file in shared directory where global settings are saved
- User – any map loaded to the system

REX.ContentGenerator investigates a user file first. If element isn’t found (or no user file was loaded), global settings will be taken to consideration.

13.6.1. RVTContentMapper and REXContentMapper



The `RVTContentMapper` and `REXContentMapper` classes provide interface to the mapping system.
`RVTContentMapper` is responsible for mapping Revit elements to `REXFamilyTypes`.
`REXContentMapper` is responsible for mapping `REXFamilyTypes` to `REXFamilyTypes`.
These two converters allows to:

- Manage user's labels (add, remove, etc.).
- Manage map files (save, load, etc.).
- Manage mapping relations (define relations, get relations, etc.).
- Manage mapping dialog (customize, launch, etc.)

Properties**:**

- `DialogSettings`	Settings of Content Map Dialog

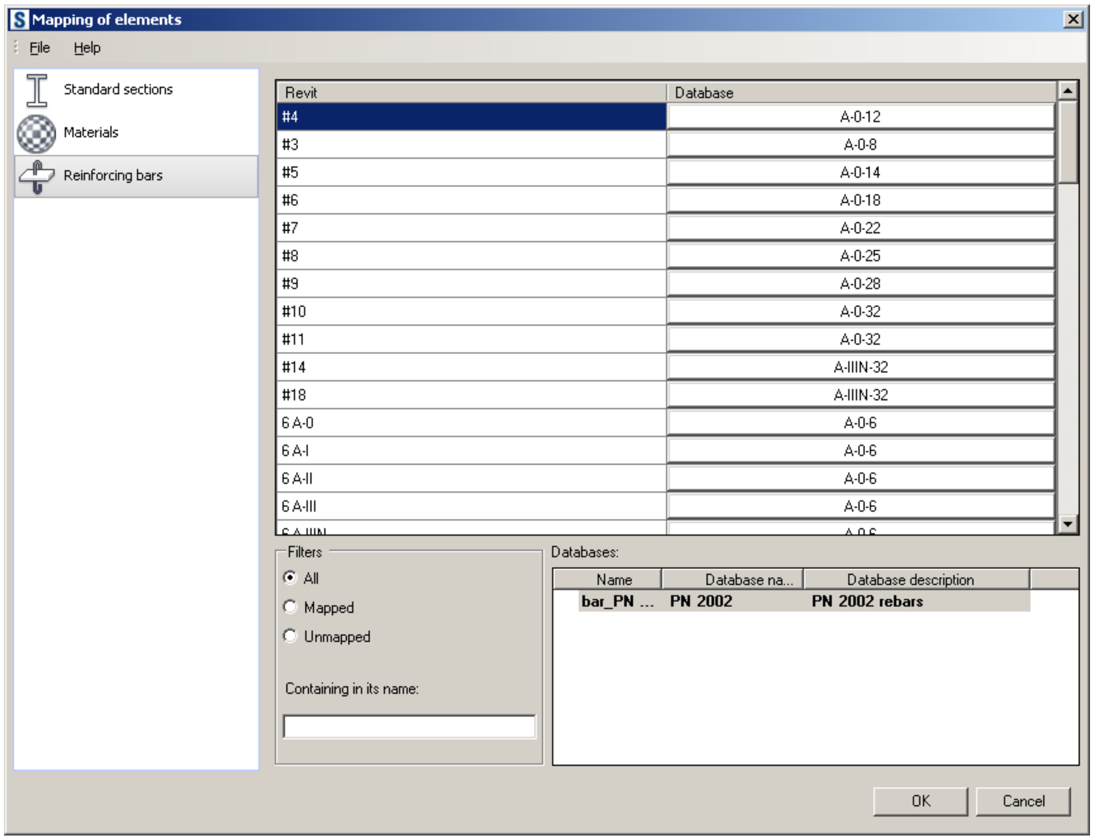
Methods:

- `GetMapElement`	Gets the map element for specific element (Revit or REXFamilyType).
- `SetMapElement`	Sets map element for specific element (Revit or REXFamilyType).
- `LaunchMapDialog`	Runs the Content Map dialog.
- `CreateMapControl`	Return the Content Map control.
- `ApplyMapControl`	Applies changes made in the Content Map control.
- `RefreshMapControl`	***Refreshes the map control due to settings set in the ContentMapper.***
`CloseMapControl`	***Closes the control (but not destroys it).***
- `LoadUserMap`	Loads the user file to the system.
- `ReloadGlobalMap`	Reloads the global settings (reads global file).
- `ResetUserMap`	Clears a user map.
- `SaveGlobalMap`	Saves current settings to the global file.
- `SaveUserMap`	Saves current settings to the user file.
- `SetDefaults`	Sets defaults (without launching dialogs).
- `GetLabels`	Returns the list of labels for specific category.
- `AddLabel`	Adds label to a list.

Dialog

REX.ContentGenerator provides a dialog where a user is able to define mapping relations. The dialog may be launched as fixed dialog or control which can be embedded in the user's dialog.
Two categories are supported in current version:

- Standard Sections
- Materials



Revit	Database
#4	A-0-12
#3	A-0-8
#5	A-0-14
#6	A-0-18
#7	A-0-22
#8	A-0-25
#9	A-0-28
#10	A-0-32
#11	A-0-32
#14	A-IIIN-32
#18	A-IIIN-32
6 A-0	A-0-6
6 A-I	A-0-6
6 A-II	A-0-6
6 A-III	A-0-6

Filters
[✓ All] Mapped Unmapped
Containing in its name:

Databases:

Name	Database na...	Database description
bar_PN ...	PN 2002	PN 2002 rebars

Layouts

Each layout is built in the same way and consists of 4 elements:

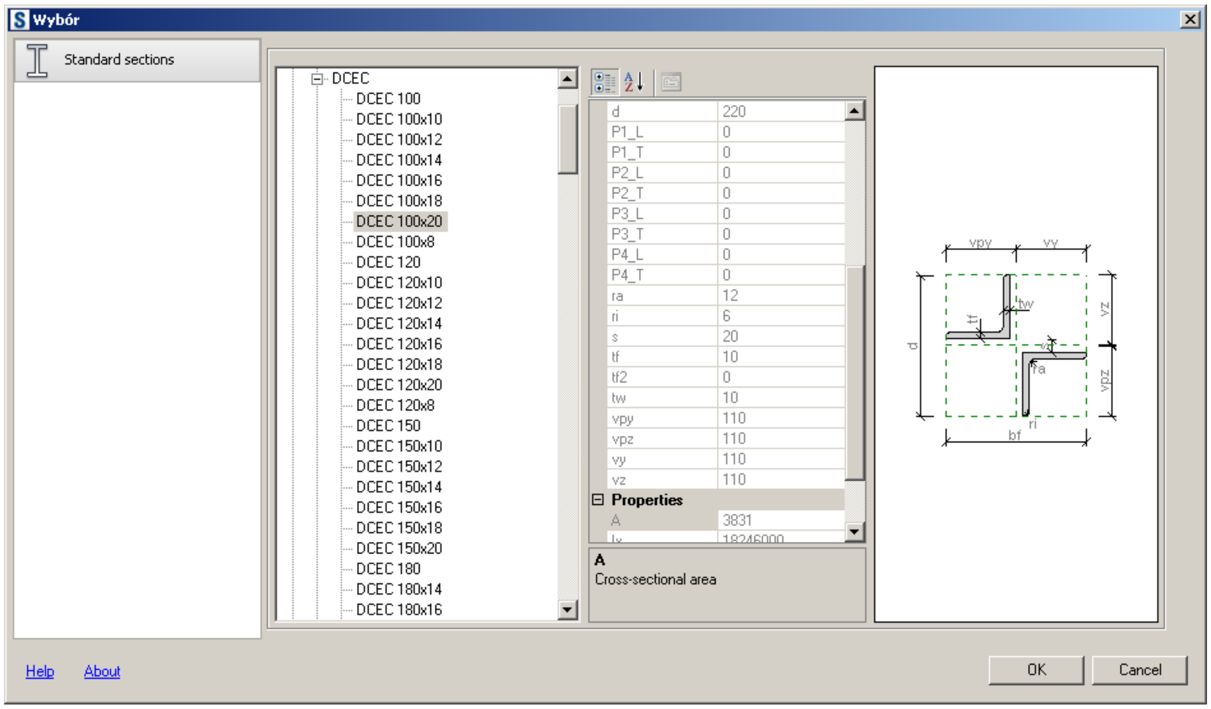
- Mapping table
- Filters
- Databases list
- Menu

Mapping table

Mapping table consists of two columns:

- Revit (or Database)
- Databases

In the first column there is a list of elements which are to be mapped. In the database column user defines how appropriate element should be mapped. After clicking specific database element the selection dialog appears.



DCEC
DCEC 100
DCEC 100x10
DCEC 100x12
DCEC 100x14
DCEC 100x16
DCEC 100x18
DCEC 100x20
DCEC 100x8
DCEC 120
DCEC 120x10
DCEC 120x12
DCEC 120x14
DCEC 120x16
DCEC 120x18
DCEC 120x20
DCEC 120x8
DCEC 150
DCEC 150x10
DCEC 150x12
DCEC 150x14
DCEC 150x16
DCEC 150x18
DCEC 150x20
DCEC 180
DCEC 180x14
DCEC 180x16

d	220
P1_L	0
P1_T	0
P2_L	0
P2_T	0
P3_L	0
P3_T	0
P4_L	0
P4_T	0
ra	12
ri	6
s	20
tf	10
tf2	0
tw	10
vpy	110
vpz	110
vy	110
vz	110
Properties	
A	3831
Ix	18246000

A
Cross-sectional area

User selects new element and validates his choice by pushing "OK" button.
If element wasn't mapped it is "Not assigned".

Filters

Elements can be filtered with three options:

- All	all elements are visible.
- Mapped	only mapped elements are visible.
- Unmapped	only unmapped elements are visible.

Databases

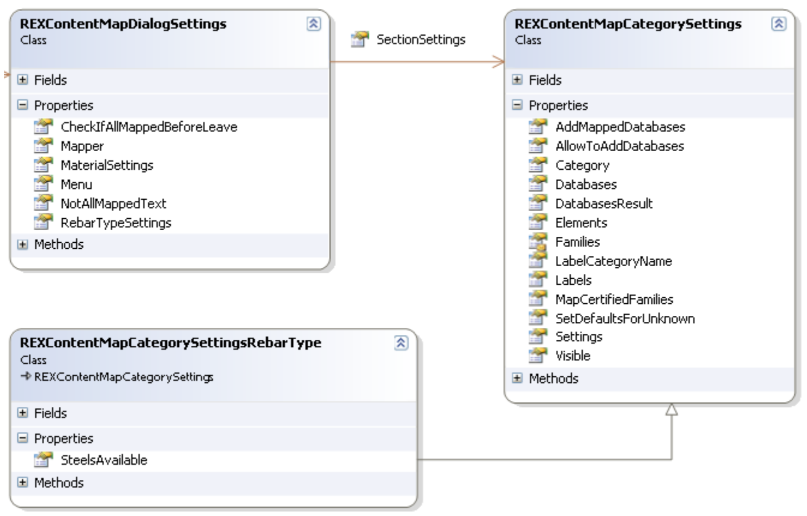
The set of databases which should be taken to consideration during mapping is presented on a special list. Databases can be added or deleted.

Menu

File menu contains following options:

- *Open*	allows for opening the specified mapping file.
- *Save*	allows for saving the current mapping settings to the current file.
- *SaveAs*	allows for saving the current mapping settings to the file.
- *Save as defaults*	allows for saving the current mapping settings to the global settings.
- *Reset defaults*	allows for loading the global settings.

13.6.2. REXContentMapDialogSettings



The dialog can be customized by `REXContentMapDialogSettings` in `RVTContentMapper` (`DialogSettings` property) object.

Properties:

- `MaterialSettings`	Settings for the material category.
- `SectionSettings`	Settings for the section category.
- `Menu`	If a menu should be shown in the dialog.
- `List`	If a list should be shown in the dialog.
- `ShowWhenNoElementsToMap`	If a dialog should be shown when there are no elements to map.
- `CheckIfAllMappedBeforeLeave`	If dialog should be validated if not all elements are mapped.
- `NotAllMappedText`	The text of the message when not all Revit elements has map element assigned.

13.6.3. REXContentMapCategorySettings

The settings of the specific category is described by `REXContentMapCategorySettings` (`MaterialSettings, SectionSettings` in `REXContentMapDialogSettings`).

Properties:

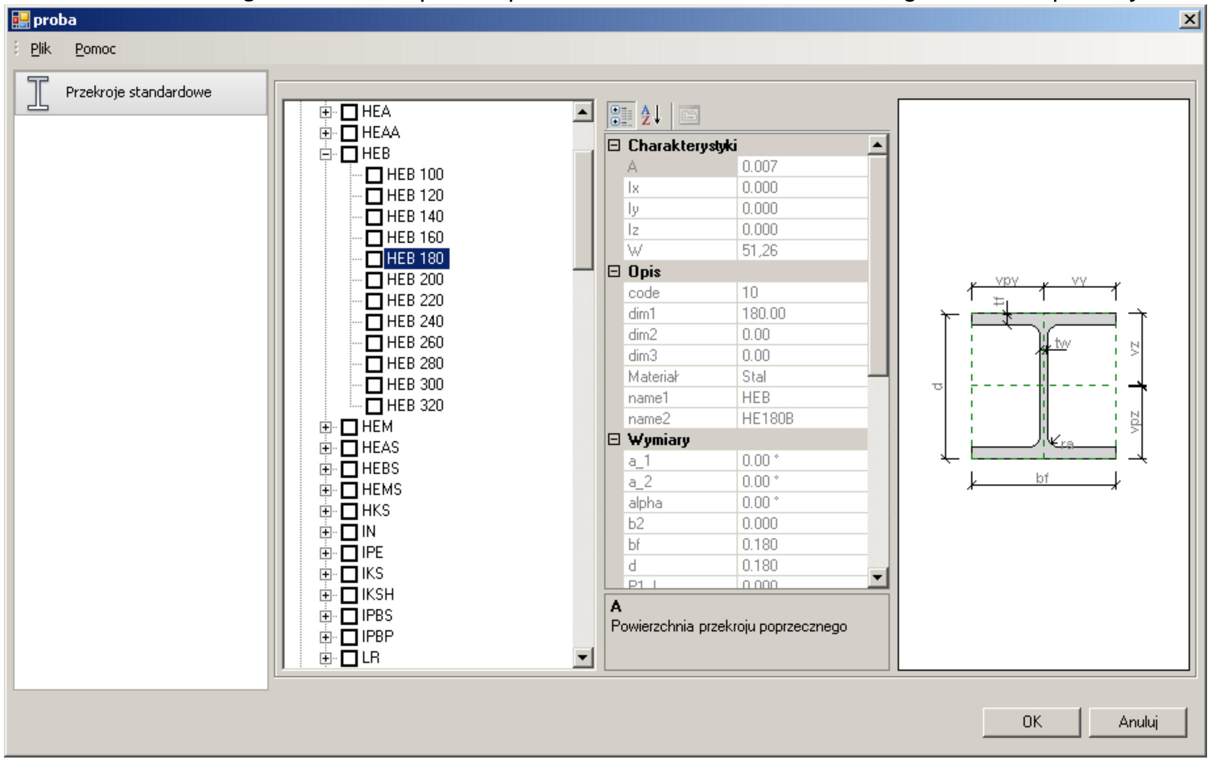
- `AddMappedDatabases`	If database associated with specific element and which doesn't exist on database list should be added automatically
- `AllowToAddDatabases`	If databases can be added by user.
- `Category`	The category of elements.
- `Databases`	A list of initial databases.
- `DatabasesResult`	A list of result databases.
- `Elements`	A list of elements to map.
- `Visible`	If specific category should be visible on the dialog.
- `MapCertifiedFamilies`	If certified elements should be taken to consideration during mapping (otherwise they will be omitted).
- `SetDefaultsForUnknown`	If default should be set for the unknown element.
- `Labels`	The list of labels.
- `LabelCategoryName`	The name for user labels category.

13.7. **Database explorer**

DContentGenerator module is an extension which is used as the database browser. It works in two modes:

- Single selection
- Multiply selection

REX.Contentgenerator component provides several classes to manage it in a simple way.



Charakterystyki	
A	0.007
Ix	0.000
Iy	0.000
Iz	0.000
W	51,26
Opis	
code	10
dim1	180.00
dim2	0.00
dim3	0.00
Materiał	Stal
name1	HEB
name2	HE180B
Wymiary	
a_1	0.00 °
a_2	0.00 °
alpha	0.00 °
b2	0.000
bf	0.180
d	0.180
P1_l	0.000

A
Powierzchnia przekroju poprzecznego

13.7.1. REXContentDialogManager

The `REXContentDialogManager` allows for 3 main actions on the Content database browser:

- `ShowContentDialog` Shows DRevitContentGenerator with input settings.
- `DisposeContentDialog` Disposes DRevitContentGenerator module.
- `SetApplicationInstance` Loads DRevitContentGenerator (if this function isn't called it will be called automatically in `ShowContentDialog` method).

13.7.2. REXContentDialogSettings

The DRevitContentGenerator is customized by `REXContentDialogSettings` object.

- `Caption`	Dialog caption.
- `Categories`	List of Settings for specific categories.
- `DialogIcon`	Dialog icon.
- `DialogMode`	Mode of dialog (generator, selection,multiselection).
- `Progress`	If progress should be visible.
- `ShowList`	If list should be visible.
- `MaterialSettings`	Settings for materials.

- `SectionDB`	Settings for database sections.
- `SectionParamSettings`	Settings for parametric sections.
- `RegionalSettings`	If regional settings should be used.

Specific category is customized by REXContentDialogCategorySettings object.

- `Category`	The category.
- `Databases`	The list of user databases (if regional settings aren't used).
- `SelectedElement`	The selected element (treated as input and output).
- `SelectedElements`	Selected elements (treated as input and output).
- `Visible`	If the category should be visible.

13.8. Examples

13.8.1. Get databases directory

Example: Get directory of databases.

There are two types of databases:

Regular databases – databases which are installed with the product in AllUsers folder (e.g. "C:\Documents and Settings\All Users\Application Data\Autodesk\Structural\Common Data\2012\Data").

Users databases – databases modified by the user located in the User folder (e.g. "C:\Documents and Settings\user1\Application Data\Autodesk\Structural\Common Data\2012")

It is possible to obtain these locations by code:

Code region

```
//Initializing the rex converter.
REXFamilyConverter rex = new REXFamilyConverter();

//Getting to databases.
String regularDBFolder = rex.GetDatabaseDirectory(ECategoryType.SECTION_DB);
String userDBFolder = rex.GetUserDatabaseDirectory(ECategoryType.SECTION_DB);
```

In case of methods where directory is optionally not required (GetDBList, GetFamilyDBSection, GetFamilyMaterial) ContentGenerator uses these two folders. The “regular” databases are taken by default, “users” are taken to consideration only when the option “SupportUserDatabases” (in converters settings) is chosen.

The “SupportUserDatabases” flag is also very important when an uncertified element in Revit is examined. If user databases are supported, RVTFamilyConverter will also try to find an appropriate element (due to elements name) among their records, otherwise it will try to find it only in “regular” databases.

Code region

```
//Initializing the rex converter.
REXFamilyConverter rex = new REXFamilyConverter();
RVTFamilyConverter rvt = new RVTFamilyConverter();

//Getting to databases.
rex.Settings.SupportUserDatabases = true;
rvt.Settings.SupportUserDatabases = true;
```

13.8.2. Get a family symbol based on the database profile

Example: Get a Revit FamilySymbol based on the "HEA100" section from the "Rcatpro" database.

There have to be made two main steps to achieve the goal:

First: Get the `REXFamilyType` object based on the database record (using `REXFamilyConverter`).

Second: Get the FamilySymbol based on the `REXFamilyType` obtained in the first step (using `RVTFamilyConverter`).

There are several ways to make the first step:

1) Get the `REXFamilyType` object based on section keys.

There are 4 key columns which identifies the section in a database: NAME, DIM1, DIM2, DIM3. It is possible to get the required section by defining them in the database description object:

Code region

```
//Initializing the rex converter.
REXFamilyConverter rex = new REXFamilyConverter();

//Creating the database description for the database query.
REXSectionDBDescription dbdesc = new REXSectionDBDescription("Rcatpro");
dbdesc.Directory = rex.GetDatabaseDirectory(ECategoryType.SECTION_DB);
dbdesc.Description.NAME = "HEA";
dbdesc.Description.DIM1 = 100;
dbdesc.Description.DIM2 = 0;
dbdesc.Description.DIM3 = 0;

//Getting the REXFamilyType from the database.
REXFamilyType fam = rex.GetFamily(dbdesc, EElementType.BEAM, EMaterial.STEEL,
ECategoryType.SECTION_DB);
REXFamilyType_DBSection famDBSection = fam as REXFamilyType_DBSection;
```

or using the GetFamilyDBSection function:

Code region

```
//Initializing the rex converter.
REXFamilyConverter rex = new REXFamilyConverter();

//Database identification data.
string dbName = "Rcatpro";
string name = "HEA";
```

```
double dim1 = 100;
double dim2 = 0;
double dim3 = 0;
string directory = rex.GetDatabaseDirectory(ECategoryType.SECTION_DB);


//Getting the REXFamilyType from the database.
REXFamilyType_DBSection                    famDBSection                    =
rex.GetFamilyDBSection(EElementType.BEAM, EMaterial.STEEL, name, dim1, dim2,
dim3, dbName, directory);
```

2) Get the `REXFamilyType` object based on the section name.

Each database section can be identified by the list of specific names. These names are directly defined in the database (NAME1, NAME_REVIT etc). Additionally combinations of keys are taken to consideration (e.g. NAME+"x"+ DIM1+"x"+ DIM2).

Code region

```
//Initializing the rex converter.
REXFamilyConverter rex = new REXFamilyConverter();

//Database identification data.
string alias = "HEA 100";
string directory = rex.GetDatabaseDirectory(ECategoryType.SECTION_DB);
string dbName = "Rcatpro";


//Getting the REXFamilyType from the database.
REXFamilyType_DBSection                    famDBSection                    =
rex.GetFamilyDBSection(EElementType.BEAM,   EMaterial.STEEL,   alias,   dbName,
directory);
```

3) Get the `REXFamilyType` object by filling all properties.

In this case a user has to fill each field of the `REXFamilyType_DBSection` object. This way isn't recommended.

The second step is obtained by using the `RVTFamilyConverter:`

Code region

```
//Initializing the RVT converter.
RVTFamilyConverter rvt = new RVTFamilyConverter(m_CommandData, true);
//Getting Revit element.
Autodesk.Revit.Element el=rvt.GetElement(famDBSection);
FamilySymbol famSymbol = el as FamilySymbol;
```

The whole algorithm:

Code region

```
//Initializing the rex converter.
REXFamilyConverter rex = new REXFamilyConverter();
RVTFamilyConverter rvt = new RVTFamilyConverter(m_CommandData, true);

//Database identification data.
string dbName = "Rcatpro";
string name = "HEA";
double dim1 = 100;
double dim2 = 0;
double dim3 = 0;
string directory = rex.GetDatabaseDirectory(ECategoryType.SECTION_DB);
//Getting the REXFamilyType from the database.
REXFamilyType_DBSection                     famDBSection                    =
rex.GetFamilyDBSection(EElementType.BEAM, EMaterial.STEEL, name, dim1, dim2,
dim3, dbName, directory);

//Getting Revit element.
Autodesk.Revit.Element el = rvt.GetElement(famDBSection);
FamilySymbol famSymbol = el as FamilySymbol;

rex.Close();
rvt.Close();
```

13.8.3. Generate a whole database

Example: Generate the whole “Rcatpro” database in Revit.

There have to be made two main steps to achieve the goal:

First: Get the list of `REXFamilyType` objects based on the “Rcatpro” database (using `REXFamilyConverter`).

Second: Generate list of Revit FamilySymbols based on the list obtained in the first step.
In the first step it is enough to use the `GetFamilies` method from the `REXFamilyConverter` object.

Code region –

```
//Initialize the converter.
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);
REXFamilyConverter rex = new REXFamilyConverter();
//Families.
string path = @"C:\Documents and Settings\All Users\Application
Data\Autodesk\Structural\Common Data\2011\Data\Prof\Rcatpro.xml";
List<REXFamilyType> families = rex.GetFamilies(path,
ECategoryType.SECTION_DB);
```

The second step can be done in two ways:
One by one generation:

Code region

```
//Generation in Revit
foreach (REXFamilyType ft in families)
{
    REXFamilyType_DBSection dbSect = ft as REXFamilyType_DBSection;
    REXFamilyType_DBSection dbSectColumn = new
REXFamilyType_DBSection(dbSect);
    dbSectColumn.ElementType = EElementType.COLUMN;

    //Generation of the column in Revit
    rvt.GetElement(dbSectColumn);
}
```

The whole collection generation:

Code region

```
List<REXFamilyType> familiesColumns = new List<REXFamilyType>();
//Generation in Revit
foreach (REXFamilyType ft in families)
{
    REXFamilyType_DBSection dbSect = ft as REXFamilyType_DBSection;
    REXFamilyType_DBSection dbSectColumn = new
REXFamilyType_DBSection(dbSect);
    dbSectColumn.ElementType = EElementType.COLUMN;

    familiesColumns.Add(dbSectColumn);
}
rvt.EventElementGenerated +=new ElementGenerated(rvt_EventElementGenerated);
rvt.GetElements(familiesColumns, ECategoryType.SECTION_DB);

static void rvt_EventElementGenerated(string id)
        {

        }
```

In this case the whole list is built first and it is input into the `GetElements` function. A user is alerted about generation action by the `EventElementGenerated`.
The whole algorithm:

Code region

```
//Initialize the converter.
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);
REXFamilyConverter rex = new REXFamilyConverter();

//Families.
string path = rex.GetDatabasePath(ECategoryType.SECTION_DB,"Rcatpro");
List<REXFamilyType> families = rex.GetFamilies(path,
ECategoryType.SECTION_DB);

//Generation in Revit
foreach (REXFamilyType ft in families)
{
    REXFamilyType_DBSection dbSect = ft as REXFamilyType_DBSection;
    REXFamilyType_DBSection dbSectColumn=new REXFamilyType_DBSection(dbSect);
    dbSectColumn.ElementType = EElementType.COLUMN;

    //Generation of the column in Revit
    rvt.GetElement(dbSectColumn);
}
rex.Close();
rvt.Close();
```

13.8.4. Generate all HEB sections with height bigger than 0.4 meters

Example: Generate all sections of “HEB” subtype and height bigger than 0.4 meters from the “Rcatpro” database.

There have to be made two main steps to achieve the goal:
First: Get the list of `REXFamilyType` objects based on the “Rcatpro” database (using `REXFamilyConverter`).

Second: Generate the list of Revit FamilySymbols based on the list obtained in the first step.
In the first step it is enough to use the `GetFilteredFamilies` method from the `REXFamilyConverter` object.

Code region –

```
//Initialize the converter.
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);
REXFamilyConverter rex = new REXFamilyConverter();

//Filter
REXContentParamFilter filterNAME = new REXContentParamFilter("NAME", "HEB",
false, false, true);
REXContentParamFilter filterH = new REXContentParamFilter("H", 0.4, true,
false, false);
REXContentLogicFilter filterLogic = new
REXContentLogicFilter(REXContentLogicFilter.ELogicOperator.And, filterNAME,
filterH);

//Families.
string path = @"C:\Documents and Settings\All Users\Application
Data\Autodesk\Structural\Common Data\2011\Data\Prof\Rcatpro.xml";
List<REXFamilyType> families = rex.GetFamilies(path, filterLogic,
ECategoryType.SECTION_DB);
```

Sections are filtered due to the `REXContentParamFilter` and the `REXContentLogicFilter` objects. There is no limitation for embedded filters. The most important is to remember that the constraint value has to be defined in base units and `ParamName` has to be the same as the name of the specific property (in the current example there has to be used: `NAME, H`).
The second step can be done in two ways:
One by one generation:

Code region

```
//Generation in Revit
foreach (REXFamilyType ft in families)
{
    REXFamilyType_DBSection dbSect = ft as REXFamilyType_DBSection;
    REXFamilyType_DBSection            dbSectColumn            =           new
REXFamilyType_DBSection(dbSect);
    dbSectColumn.ElementType = EElementType.COLUMN;

    //Generation of the column in Revit
    rvt.GetElement(dbSectColumn);
}
```

The whole collection generation:

Code region

```
List<REXFamilyType> familiesColumns = new List<REXFamilyType>();
//Generation in Revit
foreach (REXFamilyType ft in families)
{
    REXFamilyType_DBSection dbSect = ft as REXFamilyType_DBSection;
    REXFamilyType_DBSection            dbSectColumn            =           new
REXFamilyType_DBSection(dbSect);
    dbSectColumn.ElementType = EElementType.COLUMN;

    familiesColumns.Add(dbSectColumn);
}

rvt.EventElementGenerated +=new ElementGenerated(rvt_EventElementGenerated);
rvt.GetElements(familiesColumns, ECategoryType.SECTION_DB);

static void rvt_EventElementGenerated(string id)
{

}
```

In this case the whole list is built first and it is input into the `GetElements` function. A user is alerted about generation action by the `EventElementGenerated`.
The whole algorithm:

Code region

```
//Initialize the converter.
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);
REXFamilyConverter rex = new REXFamilyConverter();

//Filter
REXContentParamFilter  filterNAME  =  new  REXContentParamFilter("NAME",  "HEB",
false, false, true);
REXContentParamFilter  filterH  =  new  REXContentParamFilter("H",  0.4,  true,
false, false);
REXContentLogicFilter              filterLogic              =              new
REXContentLogicFilter(REXContentLogicFilter.ELogicOperator.And,   filterNAME,
filterH);
```

```
//Families.
string   path   =   @"C:\Documents   and   Settings\All   Users\Application
Data\Autodesk\Structural\Common Data\2011\Data\Prof\Rcatpro.xml";
List<REXFamilyType>  families  =  rex.GetFilteredFamilies(path,  filterLogic,
ECategoryType.SECTION_DB);

//Generation in Revit
foreach (REXFamilyType ft in families)
{
    REXFamilyType_DBSection dbSect = ft as REXFamilyType_DBSection;
    REXFamilyType_DBSection dbSectColumn=new REXFamilyType_DBSection(dbSect);
    dbSectColumn.ElementType = EElementType.COLUMN;

    //Generation of the column in Revit
    rvt.GetElement(dbSectColumn);
}
rex.Close();
rvt.Close();
```

13.8.5. Get database records fitting a specific Revit element

Example: Get a list of database records which fits to the specific Revit Element.

There have to be made two main steps to achieve the goal:

First:	Get the `REXFamilyType` object based on the Revit element (using `RVTFamilyConverter`).
Second:	Get the list of database records based on the `REXFamilyType` obtained in the first step.

In the first step it is enough to use the `GetFamily` method from the `RVTFamilyConverter` object.

Code region

```
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);
REXFamilyType fam = rvt.GetFamily(famInstance, ECategoryType.SECTION_DB);
```

The `GetFamily` method returns `REXFamilyType` object which may be:

- Certified – Elements made by the *REX.ContentGenerator* which contains all keys (NAME, DIM1, DIM2, DIM3) and parameters (dimensions, characteristics etc.)
- Not certified – Elements which were not made by the *REX.ContentGenerator*

In the second step the `GetDBList` method should be used:

Code region

```
REXFamilyConverter rex = new REXFamilyConverter();
List<REXDBDescription>   dbList   =   rex.GetDBList(dbSection,   directory,
ECategoryType.SECTION_DB);
```

In the example below the list of selected elements is investigated. For each element the adequate list of database records is found:

Code region

```
//Initialize the converter.
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);
REXFamilyConverter rex = new REXFamilyConverter();

//database directory
 string directory = @"C:\Documents and Settings\All Users\Application
Data\Autodesk\Structural\Common Data\2011\Data\Prof";
//Investigation of selected FamilyInstances
foreach (Element el in
commandData.Application.ActiveUIDocument.Selection.Elements)
{
    FamilyInstance famInstance = el as FamilyInstance;
    if (famInstance != null
       && (famInstance.StructuralType ==StructuralType.Beam
       || famInstance.StructuralType == StructuralType.Brace
       || famInstance.StructuralType == StructuralType.Column))
    {
        //Geting REXFamilyType object from the current element.
        REXFamilyType fam =
rvt.GetFamily(famInstance,ECategoryType.SECTION_DB);
        if (fam != null)
        {
            //The database section.
            REXFamilyType_DBSection dbSection=fam as REXFamilyType_DBSection;
            if (dbSection != null)
            {
                //The list of records.
                 List<REXDBDescription> dbList=rex.GetDBList(dbSection,
directory, ECategoryType.SECTION_DB);
                continue;
            }
        }
    }
rex.Close();
rvt.Close();
```

In the example above the whole directory is searched and the result contains the list of records from different databases. Additionally if an element is found the algorithm doesn't stop and continues an investigation. If a user wants to stop the algorithm when the first record is detected it is enough to set appropriate flag in the `REXFamilyConverter` object:

Code region

```
REXFamilyConverter rex = new REXFamilyConverter();
rex.Settings.MultiSearch = false;
```

To limit list of investigated databases there is overloaded the `GetDBList` method:

Code region – List of databases

```
REXFamilyConverter rex = new REXFamilyConverter();
string directory=@"C:\Documents and Settings\All Users\Application
Data\Autodesk\Structural\Common Data\2011\Data\Prof";
List<string> databases=new List<string>() { "Otuapro", "CorusPro", "Aiscpro"
};
List<REXDBDescription> dbList=rex.GetDBList(dbSection, databases, directory,
ECategoryType.SECTION_DB);
```

It is also possible to point the database for investigation:

Code region

```
REXFamilyConverter rex = new REXFamilyConverter();
string directory= rex.GetDatabaseDirectory(ECategoryType.SECTION_DB);
string dbName = "Otuapro";
List<REXDBDescription> dbList= rex.GetDBList(dbSection, dbName, directory,
ECategoryType.SECTION_DB);
```

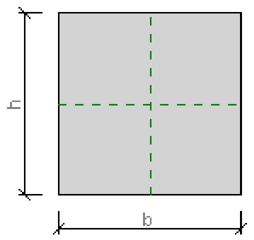
13.8.6. Get a family symbol based on a parametric description

Example: Get a Revit FamilySymbol based on a parametric description of the section (for the known type).

There have to be made two main steps to achieve the goal:

First:	Build the `REXFamilyType_ParamSection` object.
Second:	Get the FamilySymol based on the `REXFamilyType` obtained in the first step (using `RVTFamilyConverter`).

The example shows how to get the rectangle, concrete section in Revit:



Code region

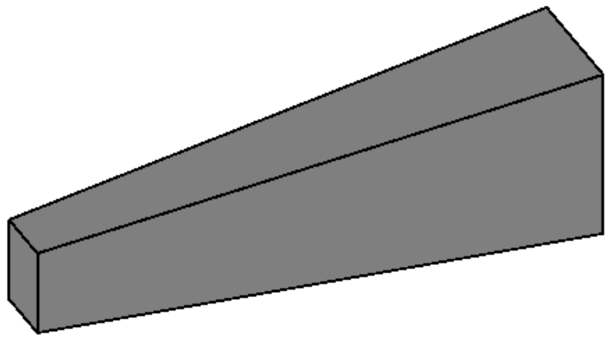
```
//Initialize the converter.
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);

//Creating parametric description.
REXFamilyType_ParamSection paramSection = new REXFamilyType_ParamSection();
paramSection.ElementType = EElementType.BEAM;
paramSection.Material = EMaterial.CONCRETE;

//Settings parameters.
```

```
paramSection.Parameters.SectionType = ESectionType.RECT;
paramSection.Parameters.Tapered = false;

//Settings dimensions.
paramSection.Parameters.Dimensions.h = 0.3;
paramSection.Parameters.Dimensions.b = 0.2;
paramSection.Parameters.CalculataMainAxisAndCharacteristics();
paramSection.TypeName = "RECT1";

//Getting Revit element.
Element el = rvt.GetElement(paramSection);
FamilySymbol fs = el as FamilySymbol;
```

The `CalculataMainAxisAndCharacteristics` function calculates all section characteristics based on applied dimensions (moment of inertia, cross-sectional area etc.).

It is very important to define the `TypeName` of the section (in case of parametric sections *REX.ContentGenerator* doesn't build its own name).

If a user wants to define a tapered section (with different dimensions on ends) it is enough to set a `Tapered` flag and define parameters in `DimensionsEnd`.



Code region

```

//Initialize the converter.
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);

//Creating parametric description.
REXFamilyType_ParamSection paramSection = new REXFamilyType_ParamSection();
paramSection.ElementType = EElementType.BEAM;
paramSection.Material = EMaterial.CONCRETE;

//Settings parameters.
paramSection.Parameters.SectionType = ESectionType.RECT;
paramSection.Parameters.Tapered = false;

//Settings dimensions.
paramSection.Parameters.Dimensions.h = 0.3;
paramSection.Parameters.Dimensions.b = 0.2;

//Setting dimensions end.
paramSection.Parameters.DimensionsEnd.h = 0.6;
paramSection.Parameters.DimensionsEnd.b = 0.4;

paramSection.Parameters.CalculataMainAxisAndCharacteristics();
paramSection.TypeName = "TRECT1";
```

```
//Getting Revit element.
Element el = rvt.GetElement(paramSection);
FamilySymbol fs = el as FamilySymbol;
```

13.8.7. Get a parametric section based on a family instance

Example: Get a parametric section based on a family instance

It is enough to use the `GetFamily` function to get the parametric description of the section:

Code region

```
//Initialize the converter.
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);

//Investigation of selected FamilyInstances
foreach(Element                              el                             in
commandData.Application.ActiveUIDocument.Selection.Elements)
{
    FamilyInstance famInstance = el as FamilyInstance;

    if (famInstance != null
       && (famInstance.StructuralType == StructuralType.Beam
       || famInstance.StructuralType == StructuralType.Brace
       || famInstance.StructuralType == StructuralType.Column))
    {
        //Geting REXFamilyType object from the current element.
        REXFamilyType         fam          =         rvt.GetFamily(famInstance,
ECategoryType.SECTION_PARAM);

        if (fam != null)
        {
            REXFamilyType_ParamSection     paramSection      =      fam      as
REXFamilyType_ParamSection;
        }
    }
}
rvt.Close();
```

In the above example there is iteration threw selected elements. Only beams, bracings and columns are taken to consideration. It is important to remember that in case of not certified elements the FamilyInstance should be put into the `GetFamily` function due to the geometry analysis which is made on elements solids.

The additional question which can appear is how to distinguish certified database section. In case of parametric section investigation it will be treated as not certified element. It is enough to use `GetCertifiedCategoryOfElement` to learn the certified category of the element (in case of uncertified the UNKNOWN result will be returned):

Code region

```
//Initializing the converter.
```

```
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);

FilteredElementCollector            filterCollector            =            new
FilteredElementCollector(commandData.Application.ActiveUIDocument.Document);
foreach (Element el in filterCollector.WhereElementIsElementType())
{
    ECategoryType category = rvt.GetCertifiedCategoryOfElement(el);

    REXFamilyType familyType = null;

    if (category != ECategoryType.UNKNOWN)//certified
    {
        familyType = rvt.GetFamily(el, category);
    }
    else//not certified
    {
        //depends on requirements
    }
}
```

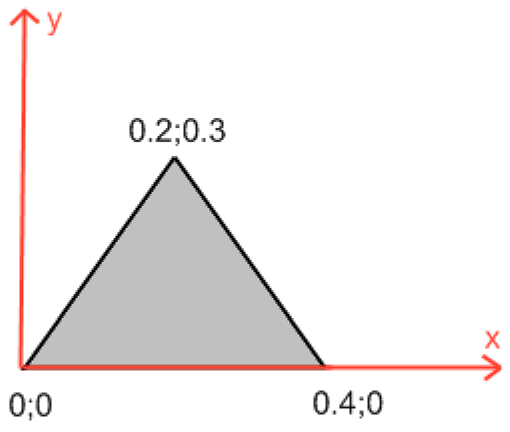
13.8.8. Get Family Symbol based on the parametric description

Example: Get the Revit FamilySymbol based on the parametric description of the section (for unknown type – contour based).

There have to be made two main steps to achieve the goal:

First:	Build the `REXFamilyType_ParamSection` object.
Second:	Get the FamilySymol based on the `REXFamilyType` obtained in the first step (using `RVTFamilyConverter`).

The goal is to create section defined by its contour:

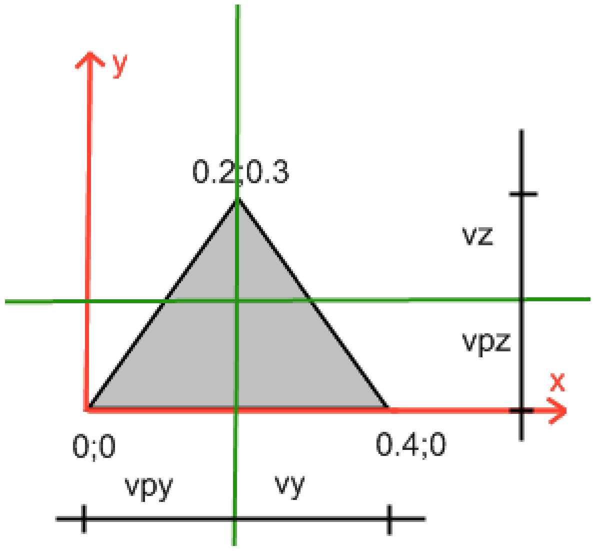


Code region

```
//Initialize the converter.
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);
```

```
//Creating parametric description.
REXFamilyType_ParamSection paramSection = new REXFamilyType_ParamSection();
paramSection.ElementType = EElementType.BEAM;
paramSection.Material = EMaterial.CONCRETE;

//Parameters.
paramSection.Parameters.SectionType = ESectionType.UNKNOWN;
//Dimensions.
paramSection.Parameters.Contour                    =                    new
REX.ContentGenerator.Geometry.Contour_Compound2D();
paramSection.Parameters.Contour.StartContour(0, 0);
paramSection.Parameters.Contour.LineTo(0.4, 0);
paramSection.Parameters.Contour.LineTo(0.2, 0.3);

paramSection.FamilyName = "Family1";
paramSection.TypeName = "example1";

//Getting Revit element.
Element el = rvt.GetElement(paramSection);
FamilySymbol fs = el as FamilySymbol;

rvt.Close();
```

If a user wants section to be defined in main axis he will have to calculate main characteristics and modify contour dimensions:

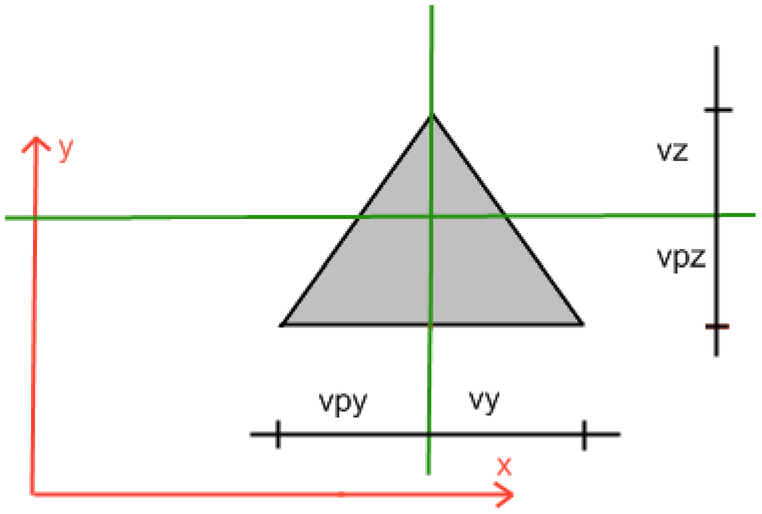


Code region

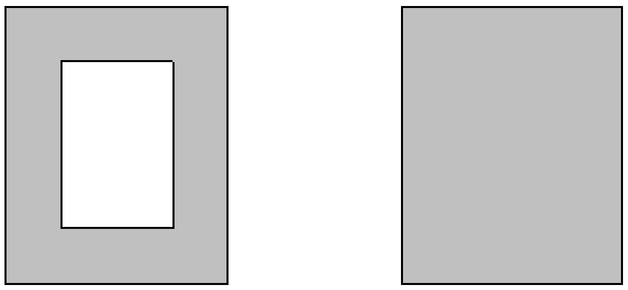
```
//Initialize the converter.
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);

//Creating parametric description.
REXFamilyType_ParamSection paramSection = new REXFamilyType_ParamSection();
paramSection.ElementType = EElementType.BEAM;
paramSection.Material = EMaterial.CONCRETE;

//Parameters.
paramSection.Parameters.SectionType = ESectionType.UNKNOWN;
```

```
//Dimensions.
paramSection.Parameters.Contour                    =                    new
REX.ContentGenerator.Geometry.Contour_Compound2D();
paramSection.Parameters.Contour.StartContour(0, 0);
paramSection.Parameters.Contour.LineTo(0.4, 0);
paramSection.Parameters.Contour.LineTo(0.2, 0.3);
paramSection.Parameters.CalculataMainAxisAndCharacteristics();

paramSection.Parameters.Contour.Translate(-
paramSection.Parameters.Dimensions.vpy,                                      -
paramSection.Parameters.Dimensions.vpz);

paramSection.FamilyName = "Family1";
paramSection.TypeName = "example1";

//Getting Revit element.
Element el = rvt.GetElement(paramSection);
FamilySymbol fs = el as FamilySymbol;

rvt.Close();
```

It is important to remember that `vpy, vpz` is the extreme distance from the local axis of the point on the negative side of the axis. It doesn't depend on coordinates:



It is also possible to define a compound section with holes:



Code region

```
//Initialize the converter.
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);

//Creating parametric description.
REXFamilyType_ParamSection paramSection = new REXFamilyType_ParamSection();
paramSection.ElementType = EElementType.BEAM;
paramSection.Material = EMaterial.CONCRETE;

//Parameters.
paramSection.Parameters.SectionType = ESectionType.UNKNOWN;
//Dimensions.
//Left contour
paramSection.Parameters.Contour.StartContour(0, 0);
paramSection.Parameters.Contour.LineTo(0.4, 0);
paramSection.Parameters.Contour.LineTo(0.4, 0.5);
paramSection.Parameters.Contour.LineTo(0, 0.5);
paramSection.Parameters.Contour.LineTo(0, 0);
//Hole in left contour
paramSection.Parameters.Contour.StartContour(0.1, 0.1,true,true);
paramSection.Parameters.Contour.LineTo(0.3, 0.1);
paramSection.Parameters.Contour.LineTo (0.3, 0.4);
paramSection.Parameters.Contour.LineTo(0.1, 0.4);
paramSection.Parameters.Contour.LineTo(0.1, 0.1);
//Right contour
paramSection.Parameters.Contour.StartContour(1, 0);
paramSection.Parameters.Contour.LineTo(1.4, 0);
paramSection.Parameters.Contour.LineTo(1.4, 0.5);
paramSection.Parameters.Contour.LineTo(1, 0.5);
paramSection.Parameters.Contour.LineTo(1, 0);

paramSection.Parameters.CalculataMainAxisAndCharacteristics();

paramSection.Parameters.Contour.Translate(-
paramSection.Parameters.Dimensions.vpy,                                       -
paramSection.Parameters.Dimensions.vpz);

paramSection.FamilyName = "Family1";
paramSection.TypeName = "example1";

//Getting Revit element.
Element el = rvt.GetElement(paramSection);
FamilySymbol fs = el as FamilySymbol;

rvt.Close();
```

The most important thing is to remember about closing each contour with start point (This indicates the end of the contour definition).

13.8.9. Get the Revit Material based database.

Example: Get the Revit Material based on the “C12/15” material from the “Eurocode” database.

There have to be made two main steps to achieve the goal:

First: Get the `REXFamilyType` object based on the database record (using `REXFamilyConverter`).

Second: Get the Material based on the `REXFamilyType` obtained in the first step (using `RVTFamilyConverter`).

There are several ways to make the first step:

1) Get the `REXFamilyType` object based on material `NAME`

The column NAME is treated as a key for material databases. It is possible to get the required material by defining it in the database description object:

Code region

```
//Initializing converters.
REXFamilyConverter rex = new REXFamilyConverter();
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);

//Creating the database description for the database query.
REXMaterialDBDescription dbdesc = new REXMaterialDBDescription("Eurocode");
dbdesc.Directory = @"C:\Documents and Settings\All Users\Application
Data\Autodesk\Structural\Common Data\2011\Data\Mate";
dbdesc.Description.NAME = "C12/15";

//Getting the REXFamilyType from the database.
REXFamilyType material = rex.GetFamily(dbdesc, EElementType.UNKNOWN,
EMaterial.UNKNOWN, ECategoryType.MATERIAL);
```

or using the `GetFamilyMaterial` function:

Code region

```
//Initializing converters.
REXFamilyConverter rex = new REXFamilyConverter();
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);

//Getting the REXFamilyType from the database.
string dbname = "Eurocode";
string directory = @"C:\Documents and Settings\All Users\Application
Data\Autodesk\Structural\Common Data\2011\Data\Mate";
string name = "C12/15";

REXFamilyType material = rex.GetFamilyMaterial(name,dbname,directory);
```

2) Get the `REXFamilyType` object by filling all properties

In this case a user has to fill each field of the `REXFamilyType_Material` object by himself. This way isn't recommended.

The second step is obtained by using `RVTFamilyConverter`:

Code region

```
//Initializing the RVT converter.
RVTFamilyConverter rvt = new RVTFamilyConverter(m_CommandData, true);
//Getting material from Revit.
Element el = rvt.GetElement(material);
Material revitMaterial = el as Material;
```

The whole algorithm:

Code region

```
//Initializing converters.
REXFamilyConverter rex = new REXFamilyConverter();
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);

//Getting the REXFamilyType from the database.
string dbname = "Eurocode";
string directory = @"C:\Documents and Settings\All Users\Application
Data\Autodesk\Structural\Common Data\2011\Data\Mate";
string name = "C12/15";

REXFamilyType material = rex.GetFamilyMaterial(name,dbname,directory);

//Getting material from Revit.
Element el = rvt.GetElement(material);
Material revitMaterial = el as Material;
```

13.8.10. Database records fitting Revit material

Example: Get the list of database records which fits to the specified Revit material

There have to be made two main steps to achieve the goal:

First: Get the `REXFamilyType` object based on the Revit element (using `RVTFamilyConverter`).

Second: Get the list of database records based on the `REXFamilyType` obtained in the first step.

In the first step it is enough to use the `GetFamily` method from the `RVTFamilyConverter` object.

Code region

```
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);
REXFamilyType familyType = rvt.GetFamily(material, ECategoryType.SECTION_DB);
```

In the second step the `GetDBList` method should be used:

Code region

```
REXFamilyConverter rex = new REXFamilyConverter();
List<REXDBDescription>   dbList   =   rex.GetDBList(familyType,   directory,
ECategoryType.SECTION_DB);
```

In the example below the list of materials defined in the project is investigated. For each material the adequate list of database records is found:

Code region

```
//Initialize converters.
RVTFamilyConverter rvt = new RVTFamilyConverter(commandData, true);
REXFamilyConverter rex = new REXFamilyConverter();

//Database directory.
string directory = rex.GetDatabaseDirectory(ECategoryType.MATERIAL);

//Investigation of materials in Revit project.
Autodesk.Revit.DB.FilteredElementCollector       collector        =        new
FilteredElementCollector(commandData.Application.ActiveUIDocument.Document);
IList<Element> materials = collector.OfClass(typeof(Material)).ToElements();

foreach (Element el in materials)
{
    REXFamilyType familyType = rvt.GetFamily(el);

    //database
    if (familyType != null)
    {
        List<REXDBDescription> dbList = rex.GetDBList(familyType, directory,
ECategoryType.MATERIAL);
        continue;
    }
}
```

In the example above all directory is searched and the result contains list of records from different databases. Additionally if element is found the algorithm doesn't stop and continues an investigation. If a user want to stop algorithm if any record is been detected it is enough to set appropriate flag in the `REXFamilyConverter` object:

Code region

```
REXFamilyConverter rex = new REXFamilyConverter();
rex.Settings.MultiSearch = false;
```

To limit list of investigated databases use the overloaded `GetDBList` method:

Code region

```
REXFamilyConverter rex = new REXFamilyConverter();

string   directory   =   @"C:\Documents   and   Settings\All   Users\Application
```

```
Data\Autodesk\Structural\Common Data\2011\Data\Mate";

List<string> databases = new List<string>() { "Eurocode", " Rmat001};

List<REXDBDescription>   dbList   =   rex.GetDBList(familyType,   databases,
directory, ECategoryType.MATERIAL);
```

It is also possible to point the database for investigation:

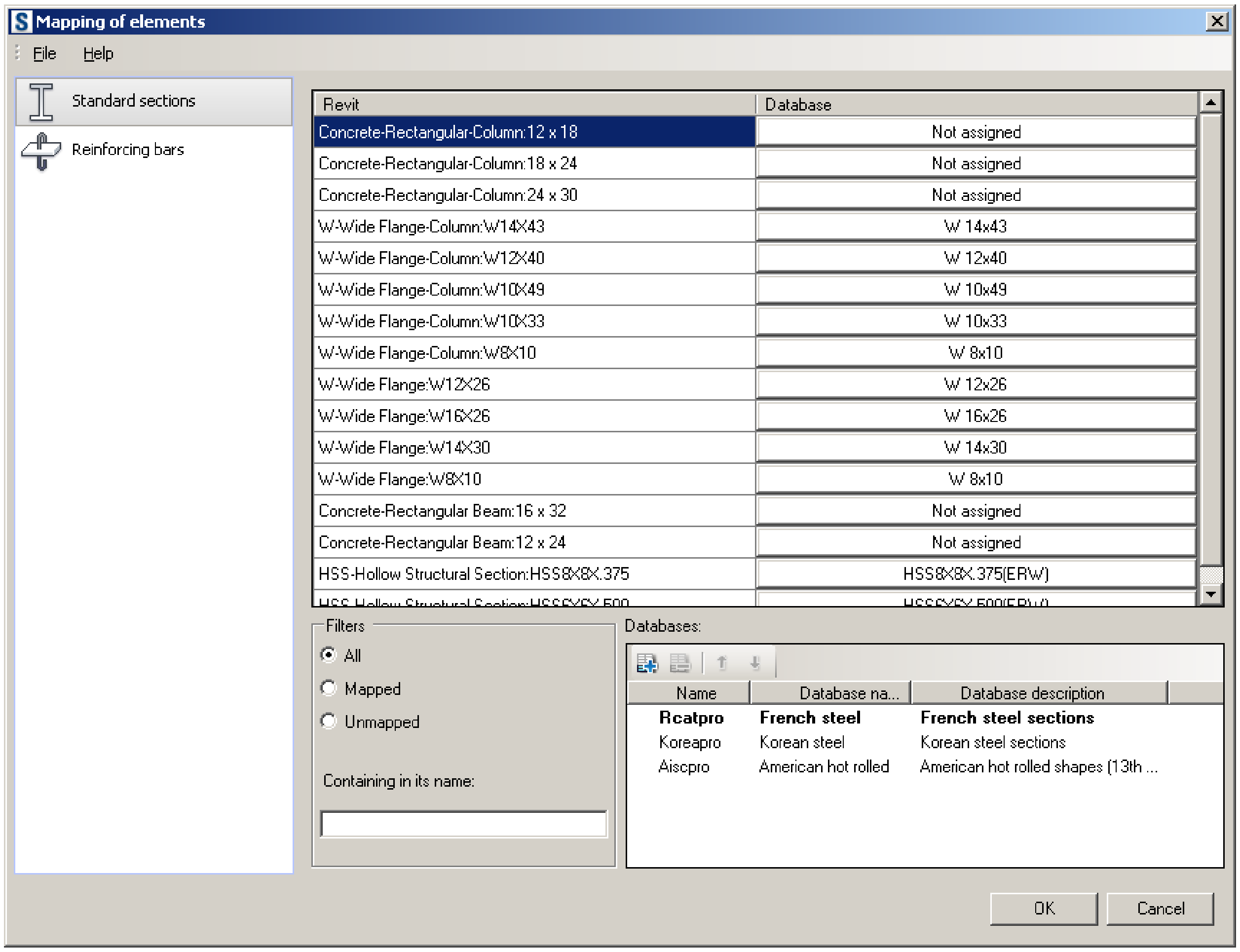
Code region

```
REXFamilyConverter rex = new REXFamilyConverter();

string directory = rex.GetDatabaseDirectory(ECategoryType.MATERIAL);


string dbName = "Eurocode";

List<REXDBDescription> dbList = rex.GetDBList(familyType, dbName, directory,
ECategoryType.MATERIAL);
```

13.8.11. Mapping in a Revit context

Example: Generate the whole “Rcatpro” database in Revit.

The main goal of content mapping is to provide a mechanism to define relations between uncertified elements in Revit and certified which come from databases (or other source).


Mapping of elements

File Help

Standard sections

Reinforcing bars

Revit	Database
Concrete-Rectangular-Column:12 x 18	Not assigned
Concrete-Rectangular-Column:18 x 24	Not assigned
Concrete-Rectangular-Column:24 x 30	Not assigned
W-Wide Flange-Column:W14X43	W 14x43
W-Wide Flange-Column:W12X40	W 12x40
W-Wide Flange-Column:W10X49	W 10x49
W-Wide Flange-Column:W10X33	W 10x33
W-Wide Flange-Column:W8X10	W 8x10
W-Wide Flange:W12X26	W 12x26
W-Wide Flange:W16X26	W 16x26
W-Wide Flange:W14X30	W 14x30
W-Wide Flange:W8X10	W 8x10
Concrete-Rectangular Beam:16 x 32	Not assigned
Concrete-Rectangular Beam:12 x 24	Not assigned
HSS-Hollow Structural Section:HSS8X8X.375	HSS8X8X.375(ERW)

Filters
[✓ All] Mapped Unmapped

Containing in its name:

Databases:

Name	Database na...	Database description
Rcatpro	**French steel**	**French steel sections**
Koreapro	Korean steel	Korean steel sections
Aiscpro	American hot rolled	American hot rolled shapes (13th ...

OK Cancel

Before launching the dialog there has to be made customization of the dialog (in the `DialogSettings` object).
In the example below elements of section category is mapped.
In the first step common settings have to be set:

Code region -

```
rvt.ContentMapper.DialogSettings.CheckIfAllMappedBeforeLeave = true;
rvt.ContentMapper.DialogSettings.NotAllMappedText = "Not all elements are
mapped";
```

In the example above there are set following features:

- The dialog should alert in case not all elements were mapped (`CheckIfAllMappedBeforeLeave`)
- The dialog should have the caption with the specified title

In the second step settings for the section category are set:

Code region

```
//Sections.
rvt.ContentMapper.DialogSettings.SectionSettings.Visible = true;
//-adding elements
```

```
foreach (FamilySymbol obj in
collector.OfClass(typeof(FamilySymbol)).ToElements())
{
    Sections.Add(obj);
    rvt.ContentMapper.DialogSettings.SectionSettings.Elements.Add(obj);
}
//-databases start list
rvt.ContentMapper.DialogSettings.SectionSettings.Databases.Add("Rcatpro");

//Materials.
rvt.ContentMapper.DialogSettings.MaterialSettings.Visible = false;
```

First of all user has to define whether the specified category should be visible (`Visible`). Than the collection of mapped elements has to be filled (`Elements`). There can be certified families among input elements. It is possible to skip them by setting the `MapCertifiedFamilies` to false. If *REX.ContentGenerator* can't find appropriate element for the specified item some default element can be taken (`SetDefaultsForUnknown`). There are also some settings connected to databases:

- List of start databases (Databases)
- If it is possible to add databases from the Dialog (`AllowToAddDatabases`)
- If databases which are loaded with the specfsied map file (opened from the dialog) should be automatically added to the current database list (`AddMappedDatabases`).

The last step is to launch dialog and get results:

Code region

```
//Launching
Autodesk.REX.Common.REXContext cont = ThisExtension.Context;
bool ok = rvt.ContentMapper.LaunchMapDialog(ref cont);

if (ok)
{
    REXFamilyType sect = rvt.ContentMapper.GetMapElement(Sections[0],
        ECategoryType.SECTION_DB);
    List<string>             selectedDatabases             =             new
List<string>(rvt.ContentMapper.DialogSettings.SectionSettings.DatabasesResult
);
}
```

Results:

- The mapped section of the first element on the list.
- The list of selected databases.

Note that the mapped element will be also returned by the `GetFamily` method if the current element is not certified (`RVTFamilyConverter` checks certification first, second mapping, in the end additional actions are taken for not certified elements: e.g. geometry investigation).
The whole algorithm:

Code region

```
//Revit converter.
RVTFamilyConverter rvt = new
RVTFamilyConverter(ThisExtension.Revit.CommandData(), true);
```

```
Autodesk.Revit.DB.FilteredElementCollector collector = new
FilteredElementCollector(ThisExtension.Revit.ActiveDocument);

List<Element> Sections = new List<Element>();

//General settings.
rvt.ContentMapper.DialogSettings.CheckIfAllMappedBeforeLeave = true;
rvt.ContentMapper.DialogSettings.NotAllMappedText = "Not all elements are
mapped";

//Sections.
rvt.ContentMapper.DialogSettings.SectionSettings.Visible = true;
//-adding elements
foreach (FamilySymbol obj in
collector.OfClass(typeof(FamilySymbol)).ToElements())
{
    Sections.Add(obj);
    rvt.ContentMapper.DialogSettings.SectionSettings.Elements.Add(obj);
}
//-databases start list
rvt.ContentMapper.DialogSettings.SectionSettings.Databases.Add("Rcatpro");

//Materials.
rvt.ContentMapper.DialogSettings.MaterialSettings.Visible = false;

//Launching
Autodesk.REX.Common.REXContext cont = ThisExtension.Context;
bool ok = rvt.ContentMapper.LaunchMapDialog(ref cont);

if (ok)
{
    REXFamilyType sect = rvt.ContentMapper.GetMapElement(Sections[0],
        ECategoryType.SECTION_DB);

    List<string> selectedDatabases = new
List<string>(rvt.ContentMapper.DialogSettings.SectionSettings.DatabasesResult
);
}
```

Except databases it is possible to define labels which can represent any element of the specified type. This element will appear in the selection list and a user will be able to map the Revit element to this label:

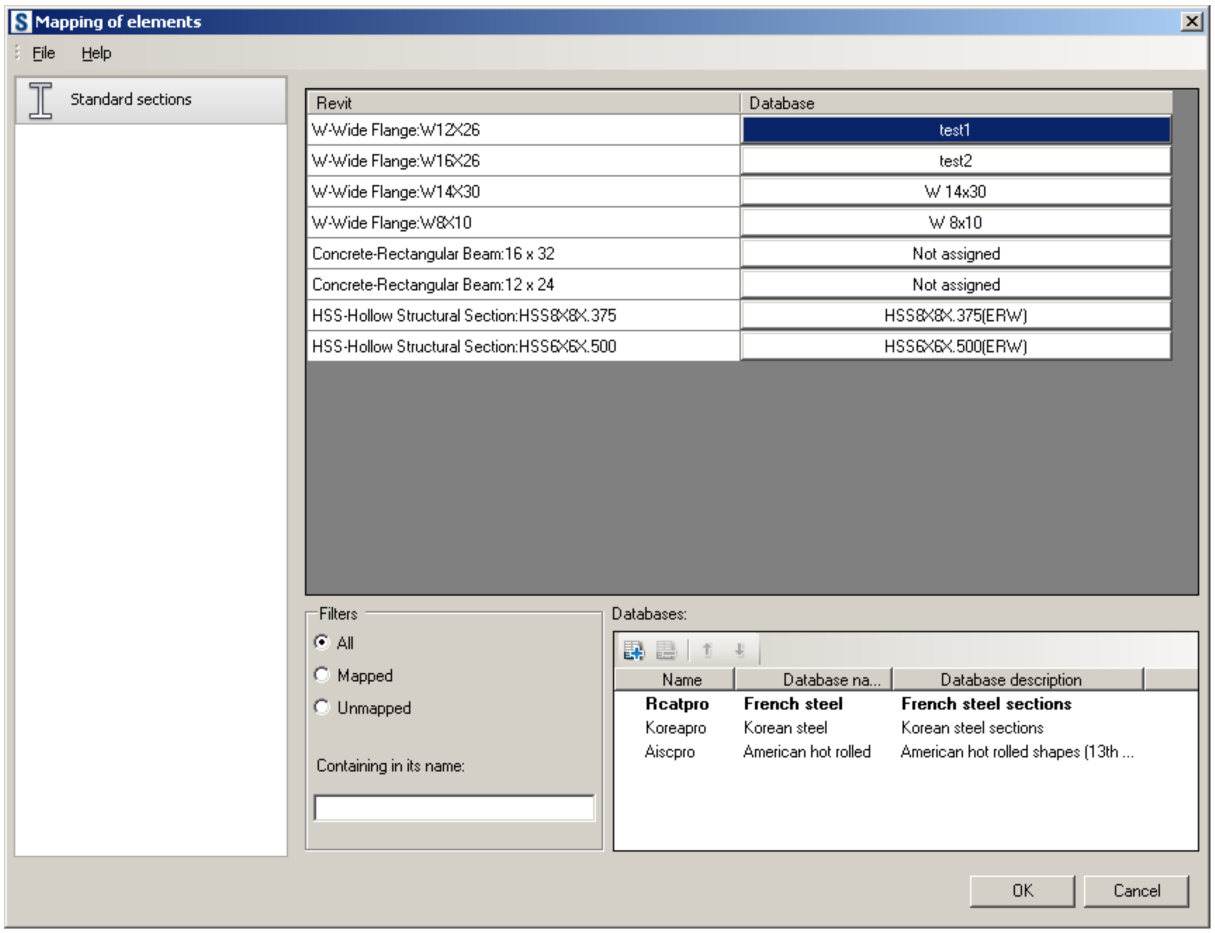
Code region -

```
//Revit converter
RVTFamilyConverter rvt = new
RVTFamilyConverter(ThisExtension.Revit.CommandData(), true);
rvt.Settings.Labels = true;

//-example label without description
REXFamilyType_Label label1 = new
REXFamilyType_Label(ECategoryType.SECTION_DB, "test1");

//-example label with section defined manually
REXFamilyType_Label label2 = new
REXFamilyType_Label(ECategoryType.SECTION_DB, "test2");
```

```
REXFamilyType_ParamSection paramSection = new REXFamilyType_ParamSection();
paramSection.Material = EMaterial.CONCRETE;
paramSection.ElementType = EElementType.BEAM;
paramSection.Parameters.SectionType = ESectionType.RECT;
paramSection.Parameters.Dimensions.b = 1;
paramSection.Parameters.Dimensions.h = 1;
label2.Parameters.LabelFamilyDescription = paramSection;

//-definition of category
rvt.ContentMapper.DialogSettings.SectionSettings.LabelCategoryName =
"MyCategory";

//-adding labels to list
rvt.ContentMapper.DialogSettings.SectionSettings.Labels.Add(label1);
rvt.ContentMapper.DialogSettings.SectionSettings.Labels.Add(label2);

//Map dialog.
Autodesk.Revit.DB.FilteredElementCollector collector = new
FilteredElementCollector(ThisExtension.Revit.ActiveDocument);
List<Element> Sections = new List<Element>();
//Sections.
rvt.ContentMapper.DialogSettings.SectionSettings.Visible = true;
//-adding elements
foreach (FamilySymbol obj in
collector.OfClass(typeof(FamilySymbol)).ToElements())
{
    if(((BuiltInCategory)obj.Category.Id.IntegerValue) ==
BuiltInCategory.OST_StructuralFraming)
        rvt.ContentMapper.DialogSettings.SectionSettings.Elements.Add(obj);
}
if (rvt.ContentMapper.DialogSettings.SectionSettings.Elements.Count > 1)
{
    rvt.ContentMapper.SetMapElement(true,
(Element)rvt.ContentMapper.DialogSettings.SectionSettings.Elements[0],
label1, ECategoryType.SECTION_DB);
    rvt.ContentMapper.SetMapElement(true,
(Element)rvt.ContentMapper.DialogSettings.SectionSettings.Elements[1],
label2, ECategoryType.SECTION_DB);
}
//-databases start list
rvt.ContentMapper.DialogSettings.SectionSettings.Databases.Add("Rcatpro");

//Materials.
rvt.ContentMapper.DialogSettings.MaterialSettings.Visible = false;

//Launching
Autodesk.REX.Common.REXContext cont = ThisExtension.Context;
bool ok = rvt.ContentMapper.LaunchMapDialog(ref cont);

if (ok)
{
    REXFamilyType sect =
rvt.ContentMapper.GetMapElement(Sections[0],ECategoryType.SECTION_DB);
}
```

In the example above two labels are created. For the “test2” label there is assigned a`FamilyLabelDescription` which allows showing its properties and a preview in the Content dialog. If a label doesn't have this description (as “test 1”) it will be shown on the selection list but after selection no

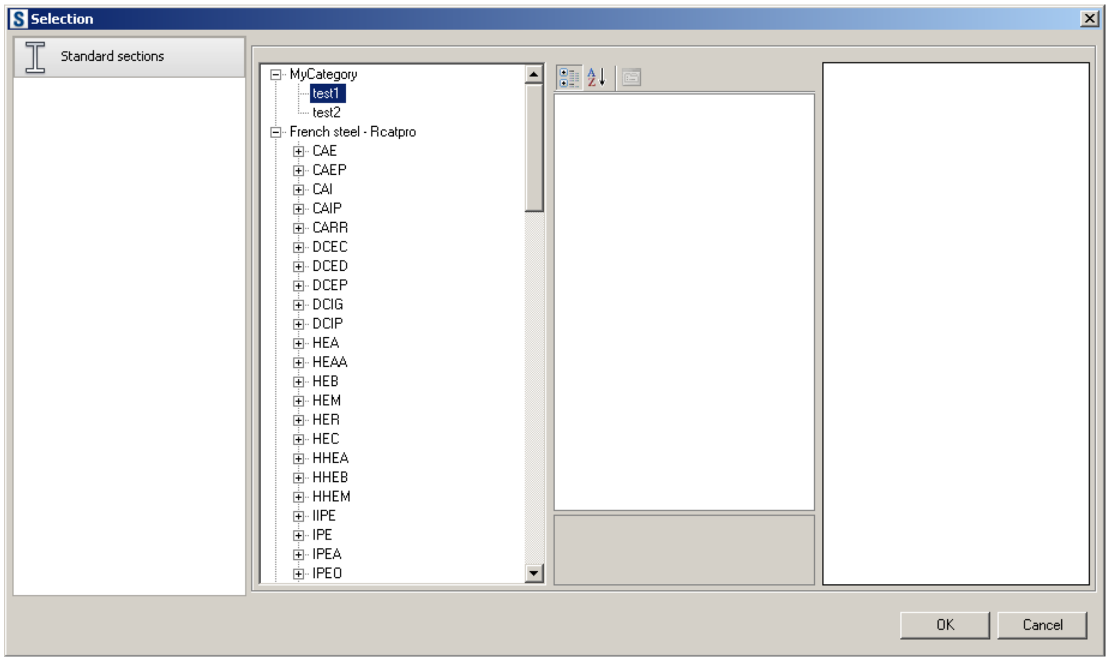
properties and preview will appear. Labels shows as an additional database which name has to be defined in the `LabelCategoryName`. Before launching the dialog two first elements in the project are assigned to labels (by `SetMapElement` function) to have it on the start as defaults. It is important to remember that the Mapping dialog is a REXExtension module so it needs context defined (it should be taken from current extension).
The main mapping dialog



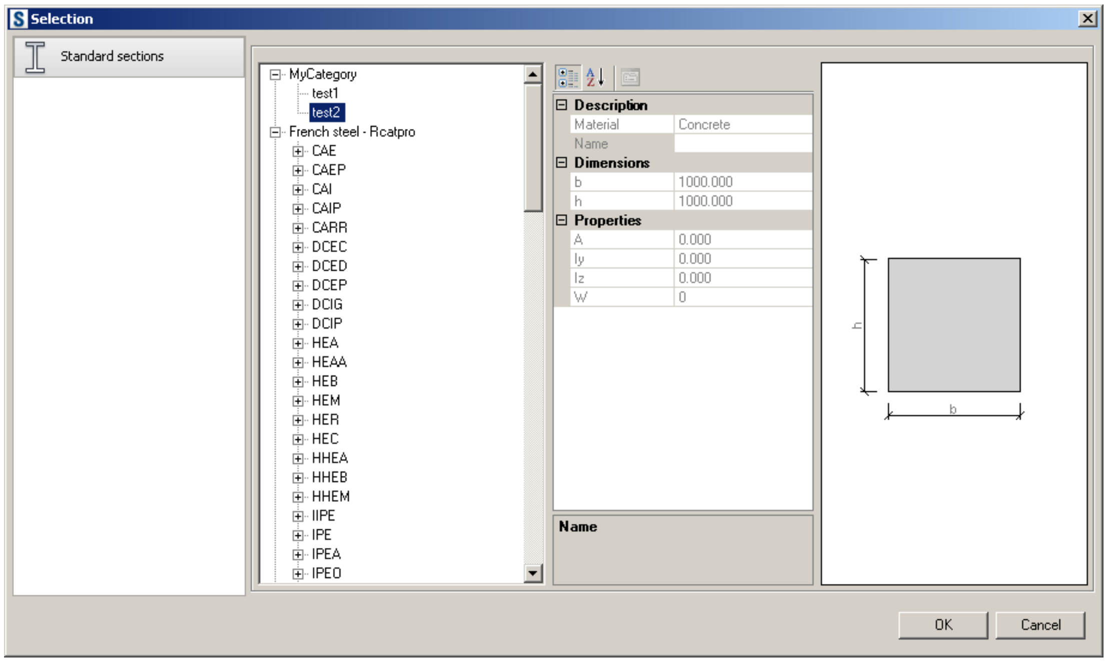
Revit	Database
W-Wide Flange:W12X26	test1
W-Wide Flange:W16X26	test2
W-Wide Flange:W14X30	W 14x30
W-Wide Flange:W8X10	W 8x10
Concrete-Rectangular Beam:16 x 32	Not assigned
Concrete-Rectangular Beam:12 x 24	Not assigned
HSS-Hollow Structural Section:HSS8X8X.375	HSS8X8X.375(ERW)
HSS-Hollow Structural Section:HSS6X6X.500	HSS6X6X.500(ERW)

Name	Database na...	Database description
Rcatpro	**French steel**	**French steel sections**
Koreapro	Korean steel	Korean steel sections
Aiscpro	American hot rolled	American hot rolled shapes (13th ...

Database browser with selected "test1" label



Database browser with selected “test2” label



Description	
Material	Concrete
Name	
Dimensions	
b	1000.000
h	1000.000
Properties	
A	0.000
Iy	0.000
Iz	0.000
W	0

Name

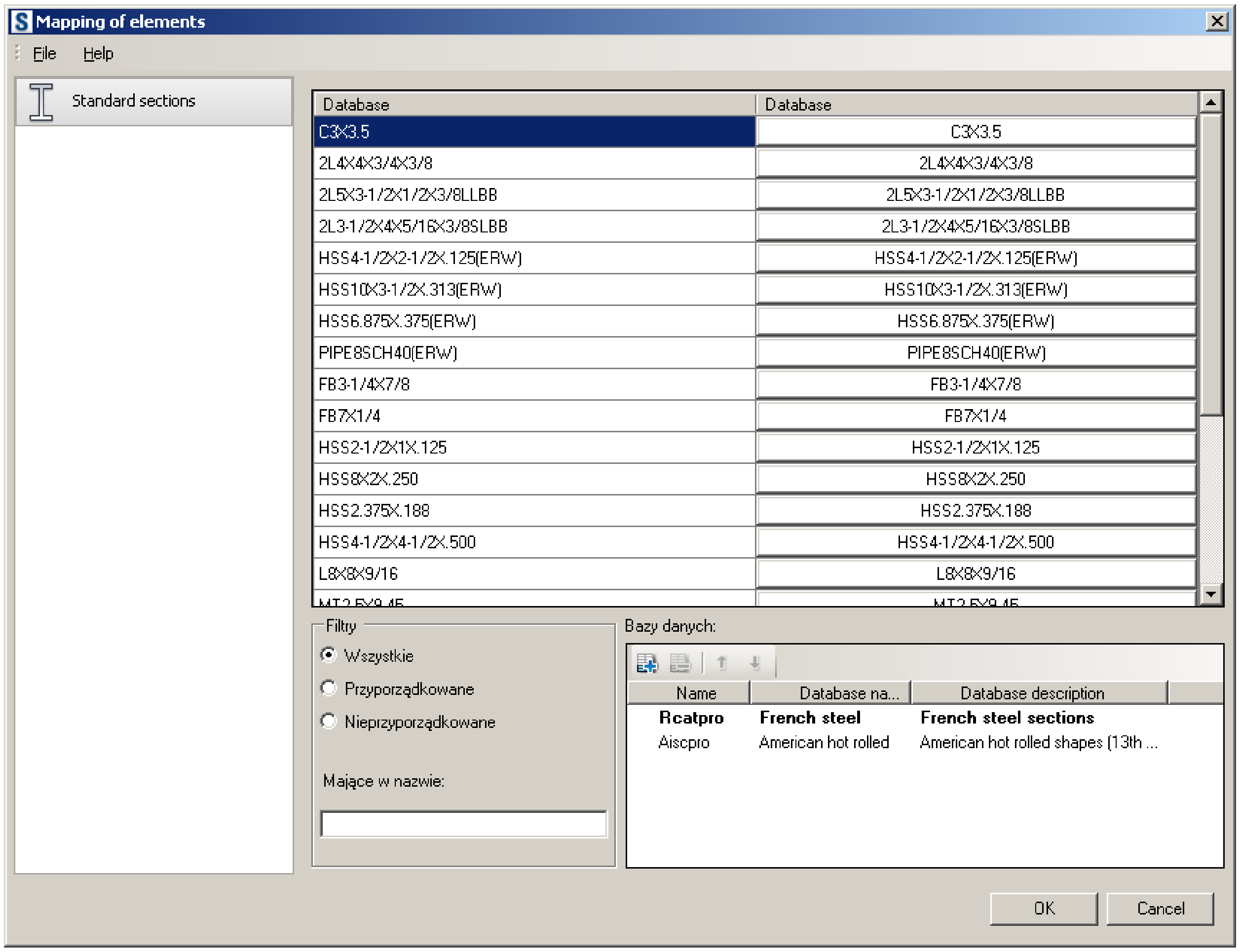
13.8.12. Mapping in Extensions context

Example: Mapping in Extensions context

The mapping system in a REX context works in the same way as in a Revit context. The example shows how it can be used in case of mapping between two databases Rcatpro and Aiscpro with additional labels added:

Code region -

```
//Revit converter
REXFamilyConverter rex = new REXFamilyConverter();

rex.ContentMapper.DialogSettings.Menu = true;
rex.ContentMapper.DialogSettings.CheckIfAllMappedBeforeLeave = true;
rex.ContentMapper.DialogSettings.NotAllMappedText = "Not all elements
mapped";
rex.ContentMapper.DialogSettings.SectionSettings.LabelCategoryName = "MyCat";
rex.Settings.Labels = true;


//Section settings
rex.ContentMapper.DialogSettings.SectionSettings.Visible = true;

List<REXFamilyType> Sections =
rex.GetFamilies(rex.GetDatabaseDirectory(ECategoryType.SECTION_DB),
"Aiscpro", ECategoryType.SECTION_DB);

//adding elements
for (int i = 0; i < Sections.Count;i++ )
{
rex.ContentMapper.DialogSettings.SectionSettings.Elements.Add(Sections[i]);
}

//databases start list
rex.ContentMapper.DialogSettings.SectionSettings.Databases.Add("Rcatpro");

REXFamilyType_Label dbSectForMap = new
REXFamilyType_Label(ECategoryType.SECTION_DB, "forMap");
rex.ContentMapper.DialogSettings.SectionSettings.Elements.Add(dbSectForMap);

REXFamilyType_Label dbSectForChoose = new
REXFamilyType_Label(ECategoryType.SECTION_DB, "forChoose");
rex.ContentMapper.AddLabel(dbSectForChoose);

//Launching
Autodesk.REX.Framework.REXContext cont = ThisExtension.Context;
bool ok = rex.ContentMapper.LaunchMapDialog(ref cont);

if (ok)
{
    REXFamilyType fam = rex.ContentMapper.GetMapElement(dbSectForMap,
ECategoryType.SECTION_DB);
}
```



Database	Database
C3X3.5	C3X3.5
2L4X4X3/4X3/8	2L4X4X3/4X3/8
2L5X3-1/2X1/2X3/8LLBB	2L5X3-1/2X1/2X3/8LLBB
2L3-1/2X4X5/16X3/8SLBB	2L3-1/2X4X5/16X3/8SLBB
HSS4-1/2X2-1/2X.125(ERW)	HSS4-1/2X2-1/2X.125(ERW)
HSS10X3-1/2X.313(ERW)	HSS10X3-1/2X.313(ERW)
HSS6.875X.375(ERW)	HSS6.875X.375(ERW)
PIPE8SCH40(ERW)	PIPE8SCH40(ERW)
FB3-1/4X7/8	FB3-1/4X7/8
FB7X1/4	FB7X1/4
HSS2-1/2X1X.125	HSS2-1/2X1X.125
HSS8X2X.250	HSS8X2X.250
HSS2.375X.188	HSS2.375X.188
HSS4-1/2X4-1/2X.500	HSS4-1/2X4-1/2X.500
L8X8X9/16	L8X8X9/16

Filtry
(●) Wszystkie
( ) Przyporządkowane
( ) Nieprzyporządkowane

Mające w nazwie:

Bazy danych:

Name	Database na...	Database description
Rcatpro	**French steel**	**French steel sections**
Aiscpro	American hot rolled	American hot rolled shapes (13th ...

OK Cancel

All procedure is similar to one described in the previous example. One thing which is worth emphasizing is that ***REXFamilyType_Labels*** can be used as mapped element as well as destination element. Thanks to this programmer is able to use it for its own purposes (unknown elements for REX.ContentGenerator).

13.8.13. Mapping control in Extensions and Revit contexts

Example: Mapping control in Extensions and Revit contexts

The mapping dialog can be embedded inside user's dialog as a control. In both contexts (REX, RVT) it works in the same way. It is enough to define all settings (see previous examples) and call the CreateMapControl method. To apply changes made in the control the user calls the ApplyMapControl, to refresh it (when some settings have been changed) the user calls the RefreshMapControl, to close it the user calls the CloseMapControl.

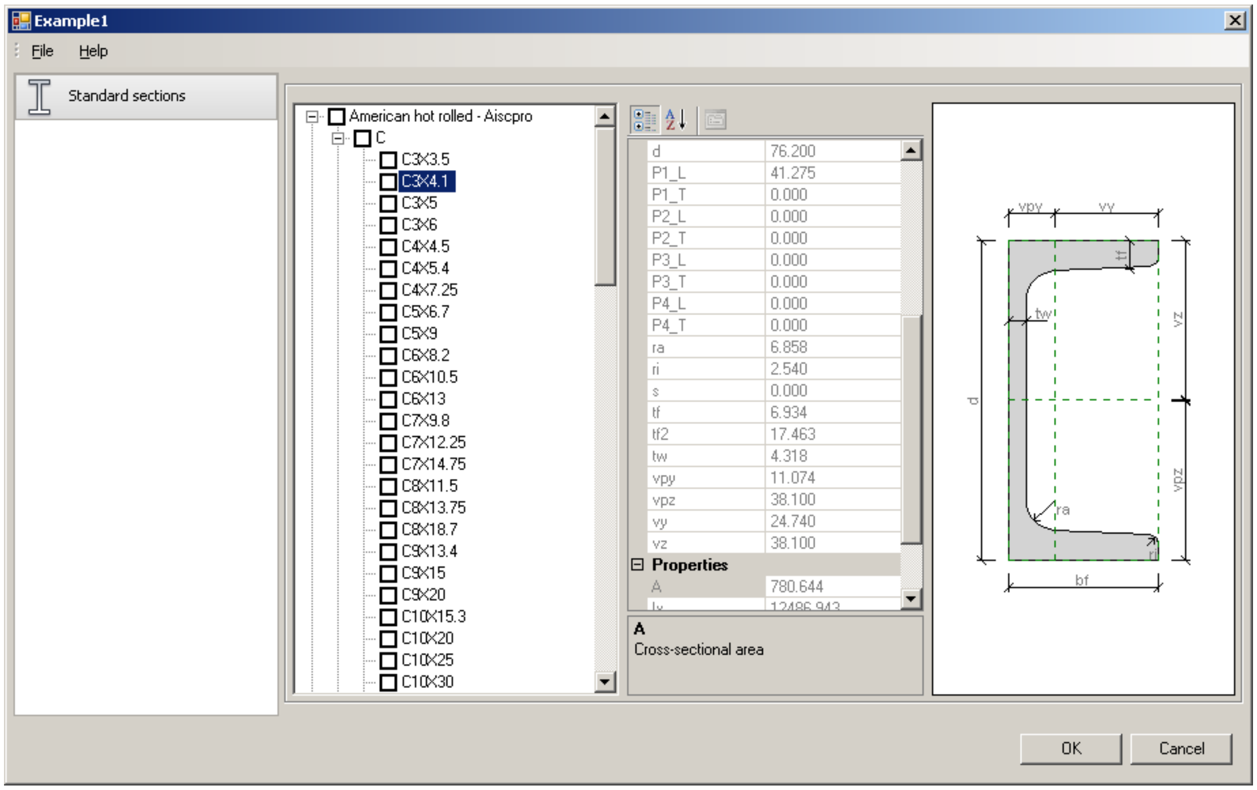
Code region -

```
//Revit converter
RVTFamilyConverter rvt = new
RVTFamilyConverter(ThisExtension.Revit.CommandData(), true);
rvt.ContentMapper.DialogSettings.Menu = false;
rvt.ContentMapper.DialogSettings.List = false;
rvt.ContentMapper.DialogSettings.ShowWhenNoElementsToMap = true;
rvt.ContentMapper.DialogSettings.CheckIfAllMappedBeforeLeave = false;
```

```
//Map dialog.
Autodesk.Revit.DB.FilteredElementCollector collector = new
FilteredElementCollector(ThisExtension.Revit.ActiveDocument);

//Sections.
rvt.ContentMapper.DialogSettings.SectionSettings.Visible = true;
//-adding elements
foreach (FamilySymbol obj in
collector.OfClass(typeof(FamilySymbol)).ToElements())
{
    if(((BuiltInCategory)obj.Category.Id.IntegerValue) ==
BuiltInCategory.OST_StructuralFraming)
        rvt.ContentMapper.DialogSettings.SectionSettings.Elements.Add(obj);
}

//-databases start list
rvt.ContentMapper.DialogSettings.SectionSettings.Databases.Add("Rcatpro");


//Launching
Autodesk.REX.Common.REXContext context = ThisExtension.Context;
object control = rvt.ContentMapper.CreateMapControl (ref context);

//Applying changes
rvt.ContentMapper.ApplyMapControl();

//Refresh the map control
rvt.ContentMapper.RefreshMapControl();

//Closing the map control
rvt.ContentMapper.CloseMapControl();
```

13.8.14. Database explorer

Example: Database explorer



American hot rolled - Aiscpro
- C
  - C3X3.5
  - C3X4.1
  - C3X5
  - C3X6
  - C4X4.5
  - C4X5.4
  - C4X7.25
  - C5X6.7
  - C5X9
  - C6X8.2
  - C6X10.5
  - C6X13
  - C7X9.8
  - C7X12.25
  - C7X14.75
  - C8X11.5
  - C8X13.75
  - C8X18.7
  - C9X13.4
  - C9X15
  - C9X20
  - C10X15.3
  - C10X20
  - C10X25
  - C10X30

d	76.200
P1_L	41.275
P1_T	0.000
P2_L	0.000
P2_T	0.000
P3_L	0.000
P3_T	0.000
P4_L	0.000
P4_T	0.000
ra	6.858
ri	2.540
s	0.000
tf	6.934
tf2	17.463
tw	4.318
vpy	11.074
vpz	38.100
vy	24.740
vz	38.100
Properties	
A	780.644
Iy	12486.943

A
Cross-sectional area

As it was mentioned before the `REXContentDialogManager` class should be used to manage the *DContentGenerator* module. The example below shows how to use it as a database explorer working due to REX RegionalSettings in the multi selection mode.

Code region -

```
//Manager initialize.
REXContentDialogManagermng=new EXContentDialogManager(ThisExtension.Context);

//Settings initialize
//-general
REXContentDialogSettings settings = new REXContentDialogSettings();
settings.DialogMode = EContentDialogMode.MultiSelection;
settings.RegionalSettings = true;
settings.ShowList = true;
settings.Progress = false;
settings.Caption = "Example1";
//-for sections
settings.SectionDBSettings.SelectedElements.Clear();
settings.SectionDBSettings.Visible = true;

//Launching the dialog.
mng.ShowContentDialog(settings, ThisExtension.GetWindowForParent());

//Getting selection.
List<REXFamilyType>families=new
List<REXFamilyType>(settings.SectionDBSettings.SelectedElements);

//Disposing dialog.
```

```
mng.DisposeContentDialog();
```

Additionally a user has to define:

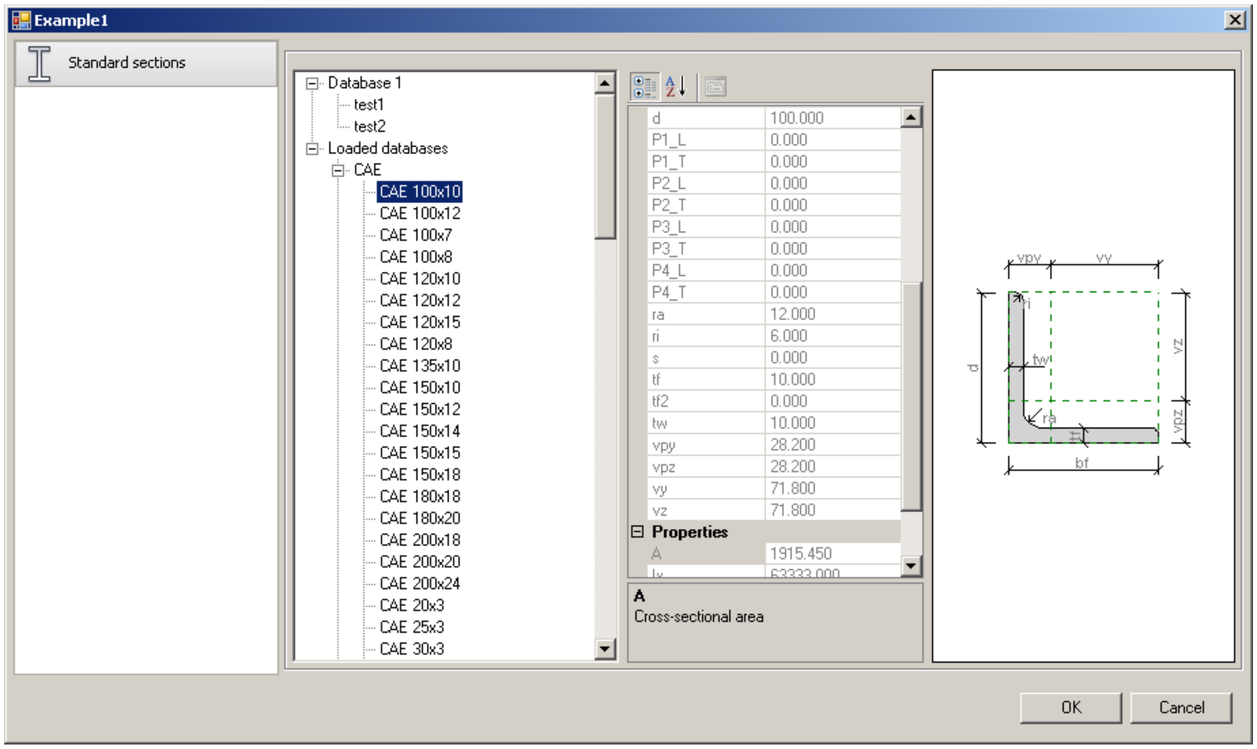
- If an Extension ListView should be exposed.
- If a Progress should be run.
- The title of the dialog.
- Settings for each category (visibility, optionally the list of elements if not run with RegionalSettings).

If a user doesn't want to use RegionalSettings and would like to have his own families the RegionalSettings flag should be set to false and Databases list should be filled:

Code region -

```
//Manager initialize.
REX.ContentGenerator.Dialog.REXContentDialogManager mng= new
REX.ContentGenerator.Dialog.REXContentDialogManager(ThisExtension.Context);

//Settings initialize
//-general
REX.ContentGenerator.Dialog.REXContentDialogSettings settings=new
REX.ContentGenerator.Dialog.REXContentDialogSettings();
settings.DialogMode=REX.ContentGenerator.Dialog.EContentDialogMode.Selection;
settings.RegionalSettings = false;
settings.ShowList = true;
settings.Progress = false;
settings.Caption = "Example1";
//-for sections
settings.SectionDBSettings.SelectedElements.Clear();
settings.SectionDBSettings.Visible = true;

//Labels
//-example label without description
REXFamilyType_Labellabel1 = new REXFamilyType_Label(ECategoryType.SECTION_DB,
"test1");

//-example label with section defined manually
REXFamilyType_Labellabel2 = new REXFamilyType_Label(ECategoryType.SECTION_DB,
"test2");
REXFamilyType_ParamSection paramSection = new REXFamilyType_ParamSection();
paramSection.Material = EMaterial.CONCRETE;
paramSection.ElementType = EElementType.BEAM;
paramSection.Parameters.SectionType = ESectionType.RECT;
paramSection.Parameters.Dimensions.b = 1;
paramSection.Parameters.Dimensions.h = 1;
label2.Parameters.LabelFamilyDescription = paramSection;

settings.SectionDBSettings.Databases.Add("Database1",new
List<REXFamilyType>() { label1, label2 });

//Sections from specific database
REXFamilyConverter rex = new REXFamilyConverter();
string path = @"C:\Documents and Settings\All Users\Application
Data\Autodesk\Structural\Common Data\2011\Data\Prof\Rcatpro.xml";
```

```
List<REXFamilyType>sections= rex.GetFamilies(path, ECategoryType.SECTION_DB);
settings.SectionDBSettings.Databases.Add("Loaded databases", sections);

//Launching the dialog.
mng.ShowContentDialog(settings, ThisExtension.GetWindowForParent());

//Getting selection.
REXFamilyType family = settings.SectionDBSettings.SelectedElement;

//Disposing dialog.
mng.DisposeContentDialog();
```



d	100.000
P1_L	0.000
P1_T	0.000
P2_L	0.000
P2_T	0.000
P3_L	0.000
P3_T	0.000
P4_L	0.000
P4_T	0.000
ra	12.000
ri	6.000
s	0.000
tf	10.000
tf2	0.000
tw	10.000
vpy	28.200
vpz	28.200
vy	71.800
vz	71.800
Properties	
A	1915.450
Iy	63333.000

A
Cross-sectional area

In the example above dialog is run in a single selection mode with own defined databases (it is important to keep compatibility with categories e.g. DBSections elements to DBSections settings). Among databases there are databases loaded from specific file and own database (defined by labels). The result selection is taken from `SelectedElement` property in specific category (in this case Section). If `SelectedElement` is set on start it will be selected on the dialog (if exist).

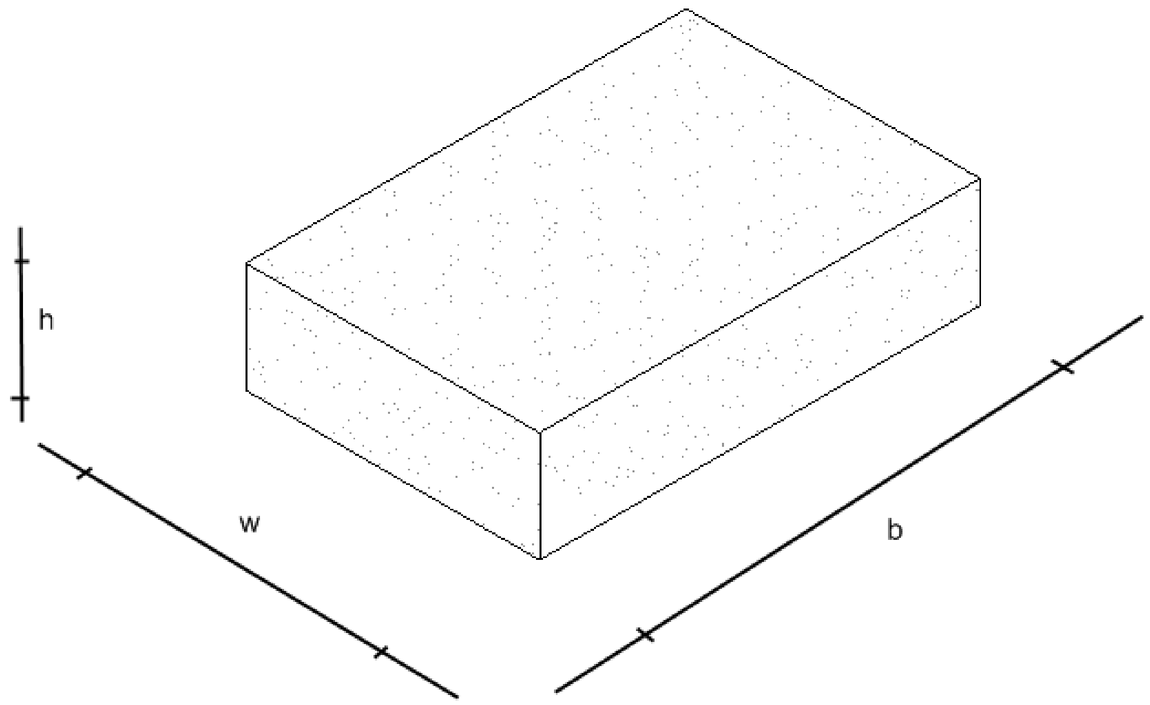
To optimize loading the dialog it is good to initialize it before (not as action on button click). There is `SetApplicationInstance` for this purpose. The dialog is created in the memory and it isn't necessary to load it every time you launch it (and load databases which is time consuming). If `SetApplicationInstance` method isn't called it will take some time when it is launched for the first time. It is good to call it somewhere where other initialize actions are taken to don't have delay e.g. on the first button press event.

The *DRevitContentDialog* is released by calling the `DisposeContentDialog` method.

13.8.15. Custom footing category

Example: Custom footing category

The goal of the example is to create a custom category of footings elements in Revit. There will be only one type supported of the rectangle shape. It will be parameterized as below:



Following this document and description of custom factories there has to be made

- Definition of common, consistent name.
- Implementation of a REXFamilyType_Custom class.
- Preparation of family templates.
- Preparation of configuration files (xml).
- Preparation of databases (xml).
- Registration of new factories inside REX and RVT converters.

Definition of common, consistent name

For example purpose the “CustomFooting” name was chosen. This name will be used for proper identification.

Implementation of a `REXFamilyType_Custom` *class*

The main class which represents footing type is `CustomFooting` which is derived from `REXFamilyType_Custom`:

Code region –

```
class CustomFooting : REXFamilyType_Custom
```

```
{
    public static string CustomCategoryName = "CustomFooting";

    public CustomFooting():base()
    {
        Description = new CustomFootingDescription();
    }

    public CustomFooting(CustomFooting fam)
        : base(fam)
    {
        Description                                 =                                  new
CustomFootingDescription((CustomFootingDescription)fam.Description);
    }

    public override string CustomCategory
    {
        get
        {
            return CustomCategoryName;
        }
    }

    public override object Clone()
    {
        return new CustomFooting(this);
    }
}
```

It is worth remembering to take care about proper cloning and appropriate `CustomCategory` returning.

Description of the footing is represented by `CustomFootingDescription` class. As it was mentioned it can be organized in a flat or more complex way. In the example this two approaches are implemented. Some properties are defined directly inside the description; others are taken from `Dimensions` object of the `CustomFootingDimension` class:

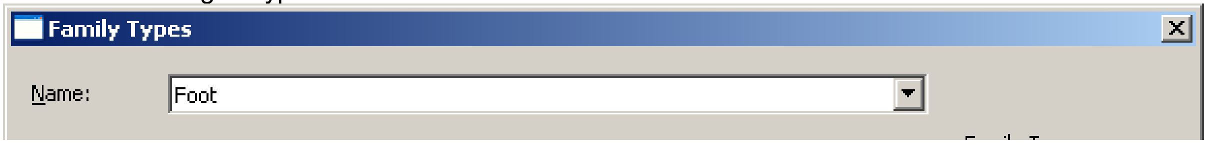
Code region –

```
    class CustomFootingDescription:REXCustomDescription
    {
        public CustomFootingDimension Dimensions{get;set;}
        public string name { get; set; }
        public string FootingType { get; set; }

        public CustomFootingDescription()
            : base()
        {
            Dimensions = new CustomFootingDimension();
        }

        public CustomFootingDescription(CustomFootingDescription desc)
            : base(desc)
        {
            Dimensions = new CustomFootingDimension(desc.Dimensions);
        }
        public override object Clone()
        {
            return new CustomFootingDescription(this);
        }

        public override string GetTypeName()
        {
            return name;
        }
    }

    class CustomFootingDimension : REXDescription
    {
        public double h { get; set; }
        public double b { get; set; }
        public double w { get; set; }

         public CustomFootingDimension()
            : base()
        {
        }

         public CustomFootingDimension(CustomFootingDimension desc)
            : base(desc)
        {
        }

         public override object Clone()
         {
             return new CustomFootingDimension(this);
         }
    }
```

Preparation of family templates

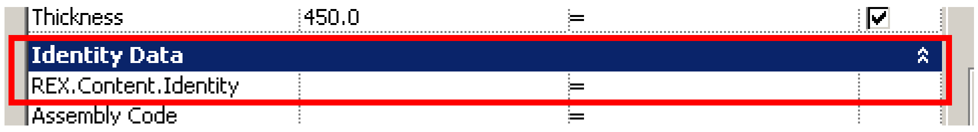
For the example purpose the footing family which is provided with Revit content was used (M_Footing-Rectangle.rfa). The way of dimensioning was preserved (parameters: Width, Length, Thickness), the only things that have been done were:

- Removing all types and add new one of “Foot” name:

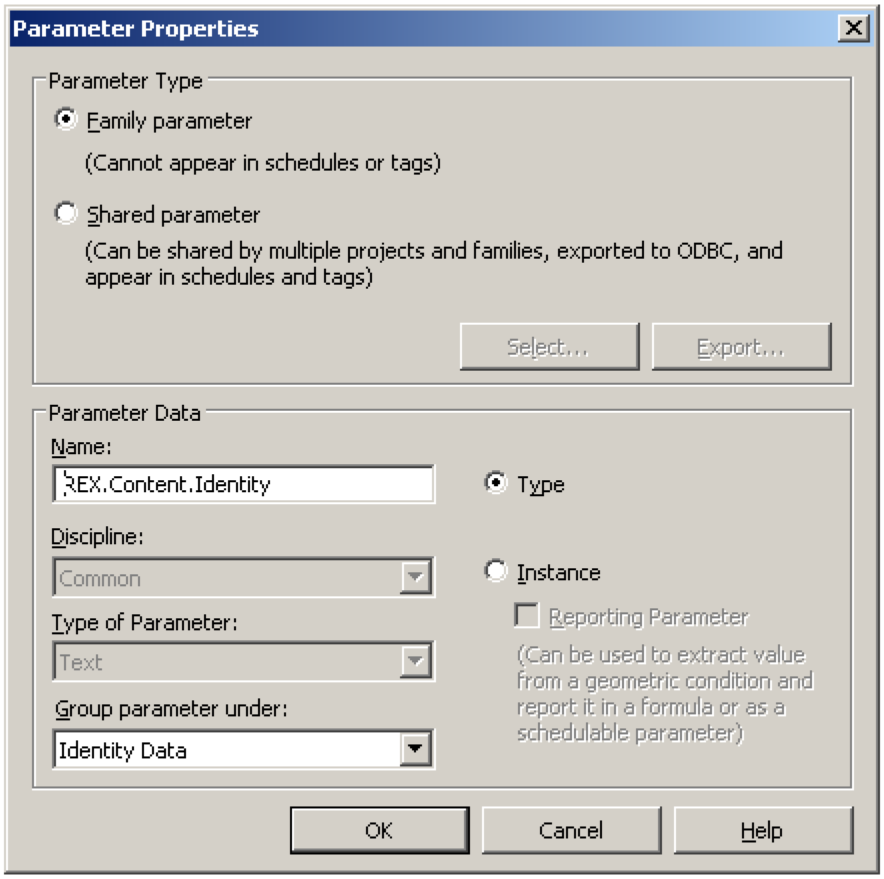


Name: Foot

- Adding “REX.Content.Identity” parameter:



```
Thickness             450.0   =   ☑
REX.Content.Identity          =
Assembly Code                 =
```


Parameter Properties
Parameter Type
Family parameter
(Cannot appear in schedules or tags)
Shared parameter
(Can be shared by multiple projects and families, exported to ODBC, and appear in schedules and tags)
Select...
Export...
Parameter Data
Name:
REX.Content.Identity
Type
Discipline:
Common
Instance
Type of Parameter:
Text
Reporting Parameter
(Can be used to extract value from a geometric condition and report it in a formula or as a schedulable parameter)
Group parameter under:
Identity Data
OK
Cancel
Help

- Saving files as footing.rfa

Preparation of configuration files (xml).

There were prepared two parameters files:

- Parameters.xml for parameters definition:

Code region –

```
<?xml version="1.0" encoding="utf-8"?>
<Categories>
  <Category name="CustomFooting">
    <Field name="Dimensions.b" unit="UC_SectionDim" power="1"
display="Length" database="b"/>
    <Field name="Dimensions.w" unit="UC_SectionDim" power="1" display="Width"
database="w"/>
    <Field name="Dimensions.h" unit="UC_SectionDim" power="1"
display="Thickness"  database="h"/>
    <Field name="name" unit="UC_Text" visible="true" identity="true"
display="name" />
    <Field name="FootingType" unit="UC_Text" template="true" identity="true"
database="type" />
  </Category>
</Categories>
```

It is very important to remember that in case of embedded hierarchy all names have to be defined with a full path. In the example 3 properties are defined in this way (b,w,h). In case of “h” parameter the attribute `display` was set to “Thickness” (according rfa file). For databases the “h” name was input.

On the level of Parameters file decision is taken about parameters which are responsible for identity (`identity` attribute) and Revit template (`template`). In the example `FootingType` property decides about templates conformity as well as identity (with `name` parameter).

- Templates.xml file defines set of rfa templates with appropriate connection with elements

Code region –

```
<?xml version="1.0" encoding="utf-8"?>
<Categories>
  <Category name="CustomFooting">
    <Family FootingType="rect" file ="footing.rfa" type="Foot" description
="My footing"/>
  </Category>
</Categories>
```

In the example there is only one footing type of the rectangle shape. In the xml above:

- There is only one template prepared for “CustomFooting” category.
- The `FootingType` property decides about template identity (it is known from Parameters.xml file) and it is of “rect” type.
- The family file is named “footing.rfa” (already prepared).
- The type defined inside the footing.rfa file is named “Foot”.
- The name of the family in Revit will be set to “My footing”.

Preparation of databases (xml)

For the example purpose simple database is created:

Code region –

```
<?xml version="1.0" encoding="utf-8"?>
<Database>
  <Units>
    <Unit type="UC_SectionDim" symbol ="mm"/>
  </Units>
  <Families>
    <Type name="Type1" b="5000" h ="3000" w="2000" type="rect" />
    <Type name="Type2" b="1000" h ="1000" w="1000" type="rect" />
  </Families>
</Database>
```

There are two types defined in the database (Type1, Type2). All parameters definition is taken from the Parameters.xml file (`database` attribute and `unit` is most important in this case). Unit settings are defined for whole category. In the example there is “UC_SectionDim” unit category set to millimeters.

Registration of new factories inside REX and RVT converters

If all items are prepared registration may be performed.

Code region –

```
//Initialize converters
RVTFamilyConverter  rvtConverter  =  new  RVTFamilyConverter(m_CommandData,
false);
REXFamilyConverter rex = new REXFamilyConverter();

//Registration of the custom factory inside RVTFamilyConverter
rvtConverter.RegisterCustomFactory<CustomFooting,
CustomFootingDescription>("CustomFooting",      @"D:\footing\Parameters.xml",
@"D:\footing\Templates.xml",                                   @"D:\footing",
BuiltInCategory.OST_StructuralFoundation);

//Registration of the custom factory inside REXFamilyConverter
rex.RegisterCustomFactory<REX.Custom.CustomFooting,
REX.Custom.CustomFootingDescription>("CustomFooting",
@"D:\footing\Parameters.xml");
```

In case of `RVTFamilyConverter` there are defined:

- Category
- The full path to Paremeters.xml file
- The full path to Templates.xml file
- The directory where family templates are stored

In case of `REXFamilyConverter` there are defined:

- Category
- The full path to Paremeters.xml file

Using custom factories

Generation of FamilySymbol based on parameters defined manually

Code region –

```
REX.Custom.CustomFooting footing = new REX.Custom.CustomFooting();

REX.Custom.CustomFootingDescription  footingDescription  =  footing.Description
as REX.Custom.CustomFootingDescription;

footingDescription.Dimensions.h = 1;
footingDescription.Dimensions.b = 2;
footingDescription.Dimensions.w = 5;
footingDescription.name = "example";
footingDescription.FootingType = "rect";
footingDescription.DBName = "database";

Element familySymbol = rvtConverter.GetElement(footing, "CustomFooting");
```

Generation of FamilySymbols based on elements taken from the database

Code region –

```
List<REXFamilyType>  elements  =  rex.GetFamilies(@"D:\footing\Database.xml",
"CustomFooting");

foreach (REXFamilyType ft in elements)
    rvtConverter.GetElement(ft);
```

Analysis of existing family symbol and find databases where it exists (all databases from specific category)

Code region –

```
//Analysis of existing family
REXFamilyType family = rvtConverter.GetFamily(familySymbol);
REXFamilyType_Custom customFamily = family as REXFamilyType_Custom;

//Getting databases where specific family exists
List<REXDBDescription> databases = rex.GetDBList(customFamily, @"D:\footing",
"CustomFooting");
```