Revit 2011 API

Developer's Guide

Version 1.0

January, 2010

Table of Contents

Contents

Table of Code Samples

1 Welcome to the Revit Platform API

All Revit-based products are Parametric Building Information Modeling (BIM) tools. These tools are similar to Computer-Aided Design (CAD) programs but are used to build 3D models as well as 2D drawings. In Revit, you place real-world elements like columns and walls into the model. Once the model is built, you can create model views such as sections and callouts. Views are generated from the 3D physical model; consequently, changes made in one view automatically propagate through all views. This virtually eliminates the need to update multiple drawings and details when you make changes to the model.

1.1 Introduction to the Revit Platform API

The Revit .NET API allows you to program with any .NET compliant language including Visual Basic.NET, C#, and C++/CLI.

Revit Architecture 2011, Revit Structure 2011, and Revit MEP 2011 all contain the Revit Platform API so that you can integrate your applications into Revit. The three APIs are very similar and are jointly referred to as the Revit Platform API. Before using the API, learn to use Revit and its features so that you can better understand the relevant areas related to your programming. Learning Revit can help you:

Maintain consistency with the Revit UI and commands.

Design your add-in application seamlessly.

Master API classes and class members efficiently and effectively.

If you are not familiar with Revit or BIM, learn more in the Revit product center at http://www.autodesk.com/revit.

1.2 What Can You Do with the Revit Platform API?

You can use the Revit Platform API to:

- Gain access to model graphical data.
- Gain access to model parameter data.
- Create, edit, and delete model elements like floors, walls, columns, and more.
- Create add-ins to automate repetitive tasks.
- Integrate applications into Revit-based vertical products. Examples include linking an external relational database to Revit or sending model data to an analysis application.
- Perform analysis of all sorts using BIM.
- Automatically create project documentation.

1.3 Requirements

To go through the user manual, you need the following:

1. A working understanding of Revit Architecture 2011, Revit Structure 2011, or Revit MEP 2011.
2. Familiarity with a Common Language Specification compliant language like C# or VB.NET.
3. Microsoft Visual Studio 2008, or Microsoft Visual Studio 2008 Express Edition. Alternatively, you can use the built-in Visual Studio Tools for Applications (VSTA) development environment in Revit. You must install VSTA from the Revit DVD separately from the Revit

product installation. For more information on writing applications using VSTA, see the section "Creating Macros with Revit VSTA" in the Revit 2011 User's Guide.

4. Microsoft .NET Framework 3.5.
5. The Revit Software Developer's Kit (SDK) which you can download from the Autodesk Developer Network (ADN) or the Revit installation CD/DVD (<DVD_Drive>:\Utilities\Common\Software Development Kit).

1.4 Install and Learn the Revit-Based Product

The examples in this document use Microsoft Visual Studio 2008 for the Integrated Development Environment (IDE).

If you are a Revit novice, go through the tutorials which are accessible from the Revit Help menu. If possible, take a training class from your local Autodesk reseller to help you quickly get up to speed.

Regardless of your experience level, you can join one of the many discussion groups dedicated to Revit and the Revit Platform API. The following resource links are a good starting point.

www.autodesk.com/revitbuilding

www.autodesk.com/revitstructure

www.autodesk.com/revitsystems

www.autodesk.com/bim

www.revitcity.com

www.augi.com

http://thebuildingcoder.typepad.com/

http://discussion.autodesk.com

Select = Autodesk Revit Architecture

Then Select = Autodesk Revit API

http://forums.augi.com/forumdisplay.php?f=93

http://discussion.autodesk.com/index2.jspa?categoryID=27

http://discussion.autodesk.com/index2.jspa?categoryID=104

http://discussion.autodesk.com/index2.jspa?categoryID=114

1.5 Installation

The Revit Platform API is installed with Revit Architecture, Revit Structure, and Revit MEP. Any .NET based application will reference the RevitAPI.dll and the RevitAPIUI.dll located in the Revit Program directory. The RevitAPI.dll contains methods used to access Revit's application, documents, elements and parameters at the database level. The RevitAPIUI.dll contains the interfaces related to manipulation and customization of the Revit user interface.

1.6 Supported Programming Languages

The Revit Platform API is fully accessible by any language compatible with the Microsoft .NET Framework 3.5, such as Visual Basic .NET or Visual C#.

1.7 User Manual

This document is part of the Revit SDK. It provides an introduction to implementing Revit add-in applications using the Revit Platform API.

Before creating a Revit Platform API add-in application read through the manual and try the sample code. If you already have some experience with the Revit Platform API, you may just want to review the Notes and Troubleshooting sections.

1.7.1 Introduction to the Revit Platform API

The first two chapters present an introduction to the Revit Platform API and provide an overview of the User Manual.

Welcome to the Revit Platform API - Presents an introduction to the Revit Platform API and necessary prerequisite knowledge before you create your first add-in.

Getting Started - Step-by-step instructions for creating your first Hello World add-in application using Visual Studio 2008 and four other walkthroughs covering primary add-in functions.

1.7.2 Basic Topics

These chapters cover the Revit Platform API basic mechanisms and functionality.

Add-in Integration - Discusses how an add-in is integrated into the Revit UI and invoked by user commands or specific Revit events such as program startup.

Application and Document - Application and Document classes respectively represent the Revit application and project file in the Revit Platform API. This chapter explains basic concepts and links to pertinent chapters and sections.

Elements Essentials - The bulk of the data in a Revit project is in a collection of Elements. This chapter discusses the essential Element mechanism, classification, and features.

Filtering – Filtering is used to get a set of elements from the document.

Selection – Working with the set of selected elements in a document

Parameters - Most Element information is stored as Parameters. This chapter discusses Parameter functionality.

Collection - Utility collection types such as Array, Map, Set collections, and related Iterators.

1.7.3 Element Topics

Elements are introduced based on element classification. Make sure that you read the Elements Essentials and Parameter chapters before reading about the individual elements.

Editing Elements – Learn how to move, rotate, delete, mirror, group, and array elements.

Wall, Floors, Roofs and Openings – Discusses Elements, their corresponding ElementTypes representing built-in place construction, and different types of Openings in the API.

Family Instances – Learn about the relationship between family and family instance, family and family instance features, and how to load or create them.

Family Creation – Learn about creation and modification of Revit Family documents.

Conceptual Design – Discusses how to create complex geometry and forms in a Revit Conceptual Mass document.

Datum and Information Elements – Learn how to set up grids, add levels, use design options, and more.

Annotation Elements – Discusses document annotation including adding dimensions, detail curves, tags, and annotation symbols.

Sketching - Sketch functions include 2D and 3D sketch classes such as SketchPlane, ModelCurve, GenericForm, and more.

Views – Learn about the different ways to view models and components and how to manipulate the view in the API.

Material - Material data is an Element that identifies the physical materials used in the project as well as texture, color, and more.

1.7.4 Advanced Topics

Geometry – Discusses graphics-related types in the API used to describe the graphical representation of the model including the three classes that describe and store the geometry information.

Place and Locations – Defines the project location including city, country, latitude, and longitude.

Shared Parameters – Shared parameters are external text files containing parameter specifications. This chapter introduces how to access to shared parameters through the Revit Platform API.

Transaction – Introduces the two uses for Transaction and the limits that you must consider when using Transaction.

Events – Discusses how to take advantage of Revit Events.

Dynamic Model Update – Learn how to use updaters to modify the model in reaction to changes in the document.

Failure Posting and Handling – Learn how to post failures and interact with Revit's failure handling mechanism.

Analysis Visualization – How to display analysis results in a Revit project.

1.7.5 Product Specific

Revit products include Revit Architecture, Revit Structure, and Revit MEP. Some APIs only work in specific products.

Revit Architecture - Discusses the APIs specific to Revit Architecture.

Revit Structure - Discusses the APIs specific to Revit Structure.

Revit MEP – Discusses the APIs specific to Revit MEP.

1.7.6 Other

Glossary – Definitions of terms used in this document.

Appendix – Additional information such as Frequently Asked Questions, Using Visual Basic.Net for programming, and more.

1.8 Documentation Conventions

This document contains class names in namespace format, such as Autodesk.Revit.DB.Element. In C++/CLI Autodesk.Revit.Element is Autodesk::Revit::DB::Element. Since only C# is used for sample code in this manual, the default namespace is Autodesk.Revit.DB. If you want to see code in Visual Basic, you will find several VB.NET applications in the SDK Samples directory.

1.8.1 Indexed Properties

Some Revit Platform API class properties are "indexed", or described as overloaded in the API help file (RevitAPI.chm). For example, the Document.Element property has two overloads. In the text of this document, these are referred to as properties, although you access them as if they were

methods in C# code by pre-pending the property name with "get_" or "set_". For example, to use the Document.Element(String) property overload, you use Document.get_Element(String).

1.9 Object Model Diagram

An object model diagram is included in the Revit Platform SDK, in a file named Revit API Diagram.dwf, which you can open using Autodesk Design Review.

1.10 What's new in this release

Please see the "What's New" section in Revit 2011 API.chm for information about changes and new features.

2 Getting Started

The Revit Platform API is fully accessible by any language compatible with the Microsoft .NET Framework 3.5, such as Visual C# or Visual Basic .NET (VB.NET). Both Visual C# and VB.NET are commonly used to develop Revit Platform API applications. However, the focus of this manual is developing applications using Visual C#.

2.1 Walkthroughs

If you are new to the Revit Platform API, the following topics are good starting points to help you understand the product. Walkthroughs provide step-by-step instructions for common scenarios, helping you learn about the product or a particular feature. The following walkthroughs will help you get started using the Revit Platform API:

Walkthrough: Hello world - Illustrates how to create an add-in using the Revit Platform API.

Walkthrough: Add Hello World Ribbon Panel - Illustrates how to add a custom ribbon panel.

Walkthrough: Retrieve Selected Elements - Illustrates how to retrieve selected elements.

Walkthrough: Retrieve Filtered Elements – Illustrates how to retrieve elements based on filter criteria.

2.2 Walkthrough: Hello World

Use the Revit Platform API and C# to create a Hello World program using the directions provided. For information about how to create an add-in application using VB.NET, refer to the section Hello World for VB.NET in the Appendix.

The Hello World walkthrough covers the following topics:

- Create a new project.
- Add references.
- Change the class name.
- Write the code
- Debug the add-in.

All operations and code in this section were created using Visual Studio 2008.

2.2.1 Create a New Project

The first step in writing a C# program with Visual Studio is to choose a project type and create a new Class Library.

1. From the File menu, select New> Project….
2. In the Project Types frame, click Visual C#.
3. In the Templates frame, click Class Library (see Figure 1: Add New Project below). This walkthrough assumes that the project location is: D:\Sample.
4. In the Name field, type HelloWorld as the project name.
5. Click OK.

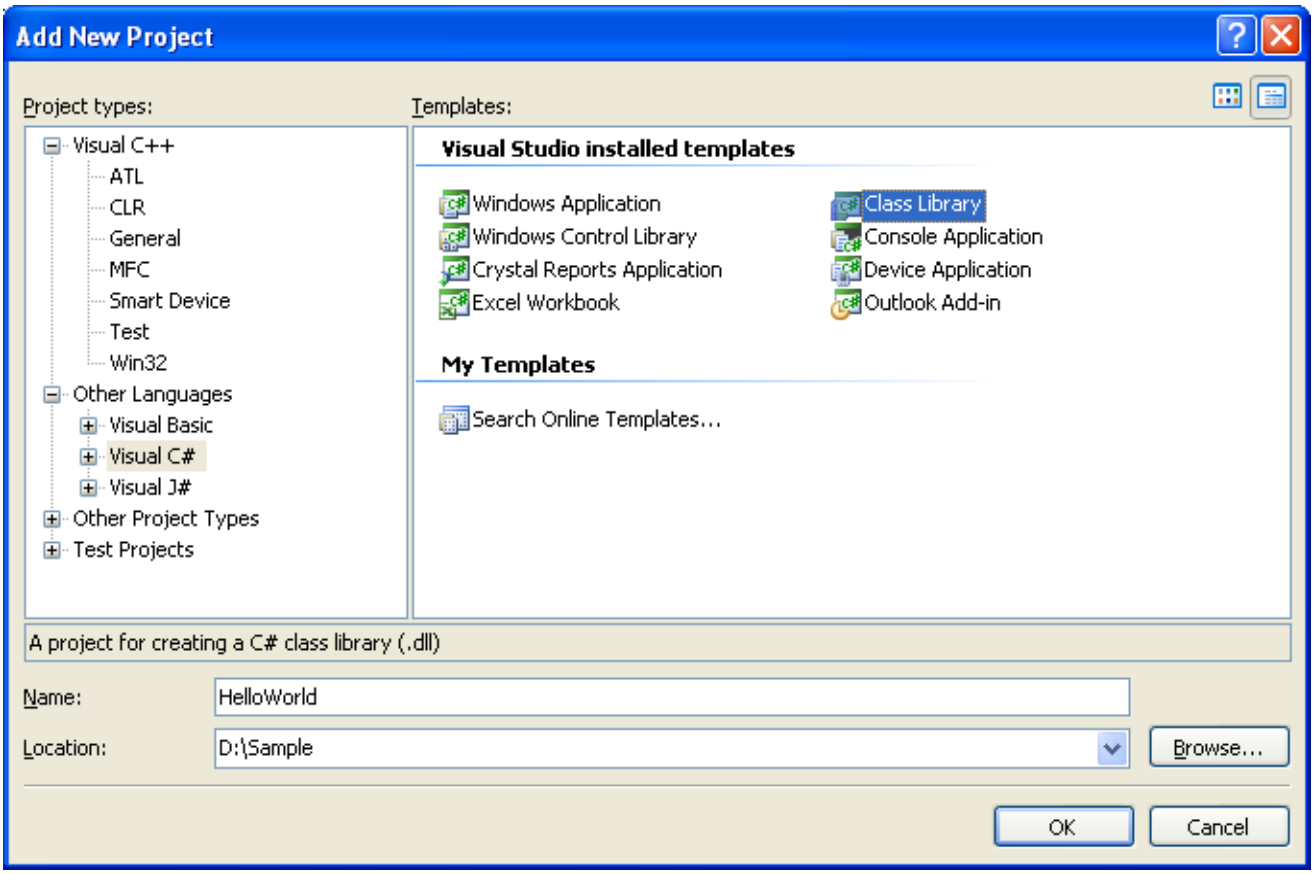


Figure 1: Add New Project

2.2.2 Add References

1. To add the RevitAPI reference:

- From the View menu select Solution Explorer if the Solution Explorer window is not open.
- In the Solution Explorer, right-click References to display a context menu.
- From the context menu, click Add Reference. The Add Reference dialog box appears.
- In the Add Reference dialog box, click the Browse tab. Locate the folder where Revit is installed and click the RevitAPI.dll. For example, the installed folder location is usually C:\Program Files\Autodesk\Revit Architecture 2011\Program\RevitAPI.dll.
- Click OK to select the .dll and close the dialog box. RevitAPI appears in the Solution Explorer reference tree.
- **Note**: You should always set the Copy Local property of RevitAPI.dll to false for new projects. This saves disk space, and prevents the Visual Studio debugger from getting confused about which copy of the DLL to use. Right-click the RevitAPI.dll, select Properties, and change the Copy Local setting from true (the default) to false.

2. Repeat the steps above for the RevitAPIUI.dll.

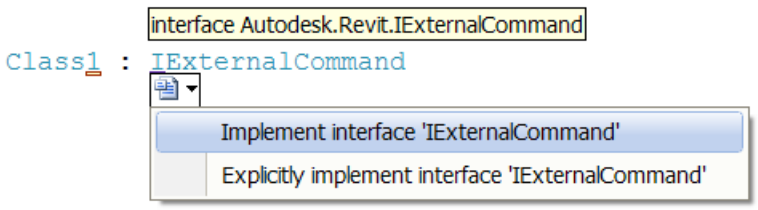
2.2.3 Add Code

Add the following code to create the add-in:

Code Region 2-1: Getting Started

```
using System;

using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
namespace HelloWorld
{

[Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.Automatic)]

[Autodesk.Revit.Attributes.Regeneration(Autodesk.Revit.Attributes.RegenerationOption.Manual)]
    public class Class1 : IExternalCommand
    {
        public Autodesk.Revit.UI.Result Execute(ExternalCommandData revit,
                    ref string message, ElementSet elements)
        {
            TaskDialog.Show("Revit", "Hello World");
            return Autodesk.Revit.UI.Result.Succeeded;
        }
    }
}
```

Tip: The Visual Studio Intellisense feature can create a skeleton implementation of an interface for you, adding stubs for all the required methods. After you add ":IExternaCommand" after Class1 in the example above, you can select "Implement IExternalCommand" from the Intellisense menu to get the code:



Every Revit add-in application must have an entry point class that implements the IExternalCommand interface, and you must implement the Execute() method. The Execute() method is the entry point for the add-in application similar to the Main() method in other programs. The add-in entry point class definition is contained in an assembly. For more details, refer to the Add-in Integration chapter.

2.2.4 Build the Program

After completing the code, you must build the file. From the Build menu, click Build Solution. Output from the build appears in the Output window indicating that the project compiled without errors.

2.2.5 Create a .addin manifest file

The HelloWorld.dll file appears in the project output directory. If you want to invoke the application in Revit, create a manifest file to register it into Revit.

1. To create a manifest file, create a new text file in Notepad.
2. Add the following text:

Code Region 2-2: Creating a .addin manifest file for an external command

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
  <AddIn Type="Command">
    <Assembly>D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll</Assembly>
    <AddInId>239BD853-36E4-461f-9171-C5ACEDA4E721</AddInId>
    <FullClassName>HelloWorld.Class1</FullClassName>
    <Text>HelloWorld</Text>
  </AddIn>
</RevitAddIns>
```

3. Save the file as HelloWorld.addin and put it in the following location:

- For Windows XP - C:\Documents and Settings\All Users\Application Data\Autodesk\Revit\Addins\2011\
- For Vista/Windows 7 - C:\ProgramData\Autodesk\Revit\Addins\2011\

Refer to the Add-in Integration chapter for more details using manifest files.

2.2.6 Debug the Add-in

Running a program in Debug mode uses breakpoints to pause the program so that you can examine the state of variables and objects. If there is an error, you can check the variables as the program runs to deduce why the value is not what you might expect.

1. In the Solution Explorer window, right-click the HelloWorld project to display a context menu.
2. From the context menu, click Properties. The Properties window appears.
3. Click the Debug tab.
4. Under the Start Action section, click Start external program and browse to the Revit.exe file. By default, the file is located at the following path, C:\Program Files\Autodesk\Revit Structure 2011\Program\Revit.exe.

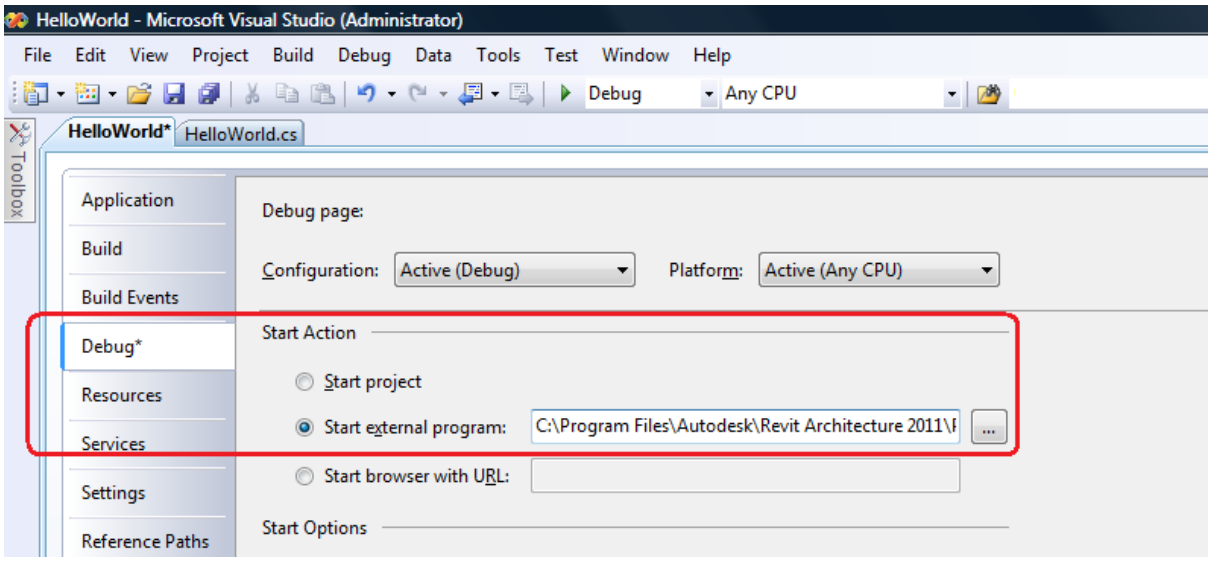


Figure 2: Set debug environment

5. From the Debug menu, select Toggle Breakpoint (or press F9) to set a breakpoint on the following line.

```
TaskDialog.Show("Revit", "Hello World");
```

6. Press F5 to start the debug procedure.

Test debugging:

- On the Add-Ins tab, HelloWorld appears in the External Tools menu-button.

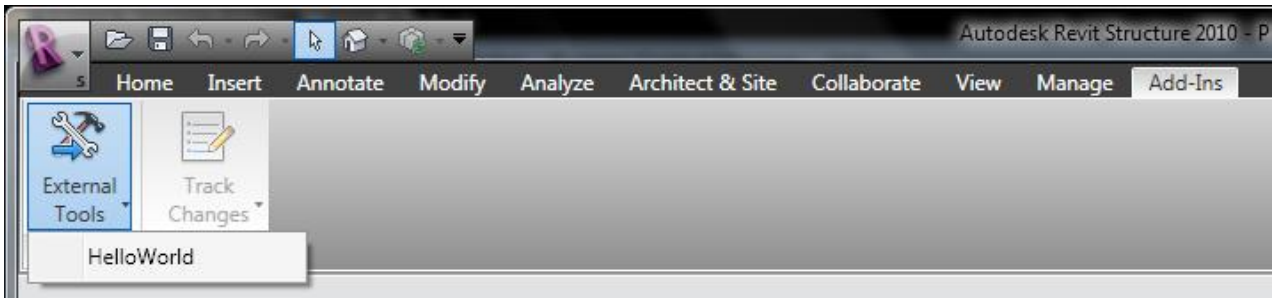


Figure 3: HelloWorld External Tools command

- Click HelloWorld to execute the program, activating the breakpoint.
- Press F5 to continue executing the program. The following system message appears.

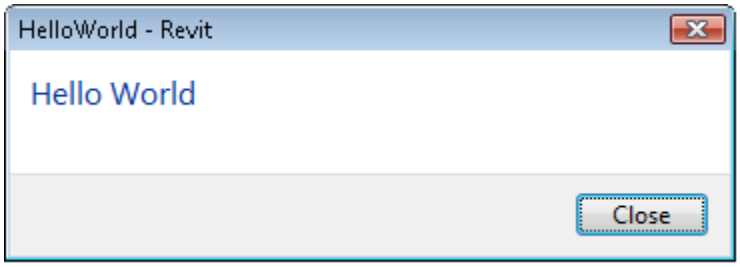


Figure 4: TaskDialog message

2.2.7 **Troubleshooting**

Q: My add-in application will not compile.

A: If an error appears when you compile the sample code, the problem may be with the version of the RevitAPI used to compile the add-in. Delete the old RevitAPI reference and load a new one. For more details, refer to Add Reference.

Q: Why is there no Add-Ins tab or why isn't my add-in application displayed under External Tools?

A: In many cases, if an add-in application fails to load, Revit will display an error dialog on startup with information about the failure. For example, if the add-in DLL cannot be found in the location specified in the Revit.ini file, a message similar to the following appears.

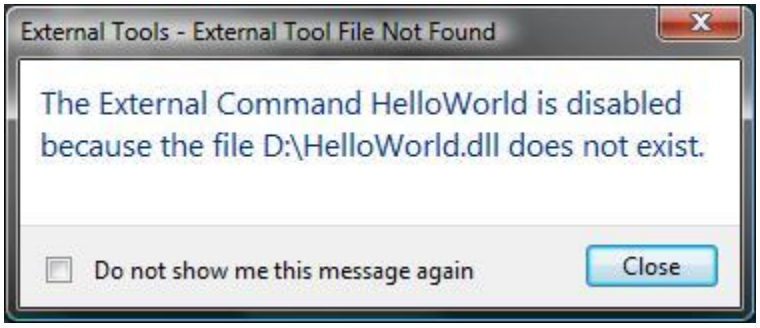


Figure 5: External Tools Error Message

Error messages will also be displayed if the class name specified in ECClassName is not found or does not inherit from IExternalCommand.

However, in some cases, an add-in application may fail to load without any message. Possible causes include:

- The add-in application is compiled with a different RevitAPI version
- Revit cannot find the Revit.ini file in the Revit setup folder
- [ExternalCommands] block is missing from Revit.ini
- [ExternalCommands] block is empty
- ECCount is not equal to the real number of commands

Q: Why does my add-in application not work?

A: Even though your add-in application is available under External Tools, it may not work. This is most often caused by an exception in the code.

For example:

Code Region 2-3: Exceptions in Execute()

```
Command: IExternalCommand
{
   A a = new A();//line x
   public IExternalCommand.Result Execute ()
   {
     //…
   }
}
Class A
{
  //…
}
```

The following two exceptions clearly identify the problem:

- An error in line x
- An exception is thrown in the Execute() method.

Revit will display an error dialog with information about the uncaught exception when the command fails.

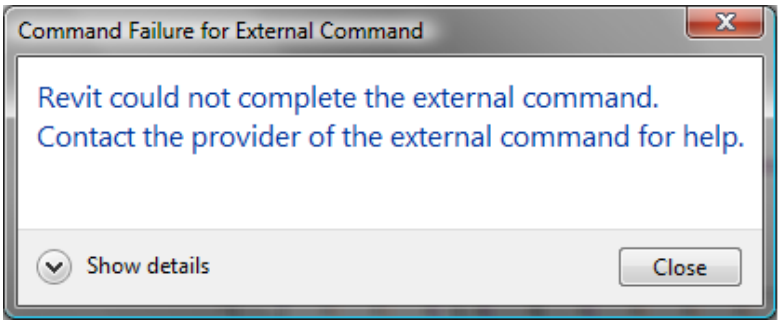


Figure 6: Unhandled exception in External Command

This is intended as an aid to debugging your command; commands deployed to users should use try..catch..finally in the example entry method to prevent the exception from being caught by Revit. Here's an example:

Code Region 2-4: Using try catch in execute:

```
public IExternalCommand.Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements)
{
    ExternalCommandData cdata = commandData;
    Autodesk.Revit.ApplicationServices.Application app = cdata.Application;

    try
    {
        // Do some stuff
    }

    catch (Exception ex)
    {
        message = ex.Message;
        return IExternalCommand.Result.Failed;
    }

    return IExternalCommand.Result.Succeeded;    }
```

2.3 Walkthrough: Add Hello World Ribbon Panel

In the Walkthrough: Hello World section you learn how to create an add-in application and invoke it in Revit. You also learn to create a .addin manifest file to register the add-in application as an external tool. Another way to invoke the add-in application in Revit is through a custom ribbon panel.

2.3.1 Create a New Project

Complete the following steps to create a new project:

1. Create a C# project in Visual Studio using the Class Library template.
2. Type AddPanel as the project name.
3. Add references to the RevitAPI.dll and RevitAPIUI.dll using the directions in the previous walkthrough, Walkthrough: Hello World.
4. Add the PresentationCore reference:

- In the Solution Explorer, right-click References to display a context menu.
- From the context menu, click Add Reference. The Add Reference dialog box appears.
- In the Add Reference dialog box, click the .NET Tab.
- From the Component Name list, select PresentationCore.
- Click OK to close the dialog box. PresentationCore appears in the Solution Explorer reference tree.

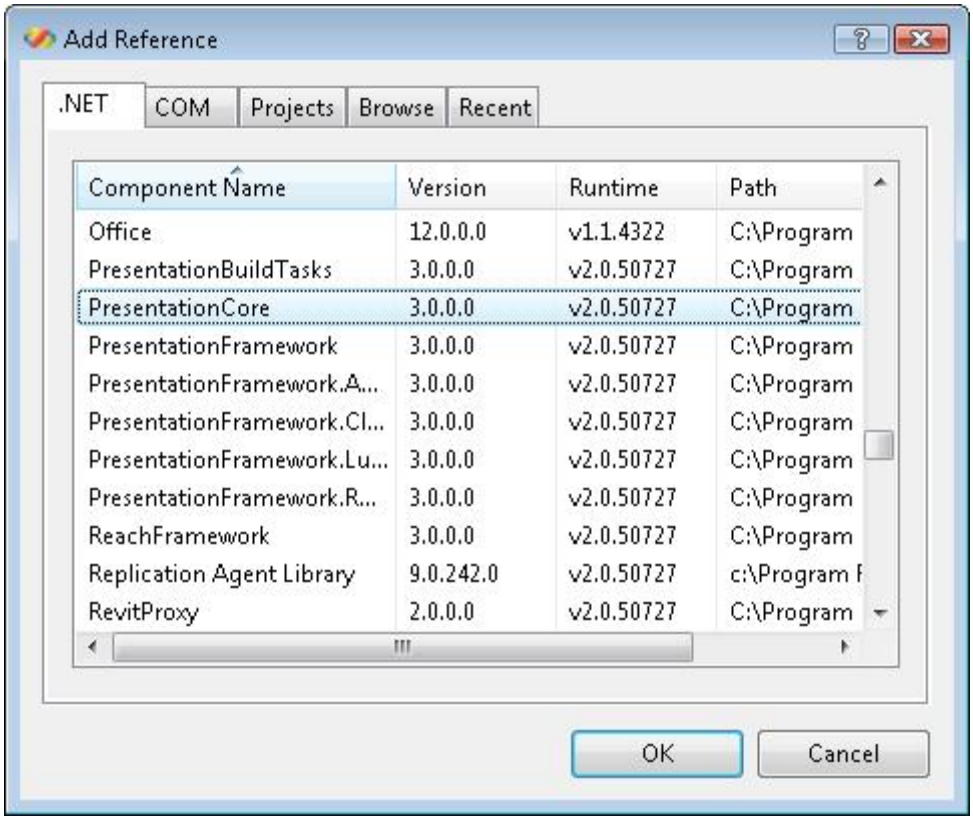


Figure 7: Add Reference

5. Add the WindowsBase reference following similar steps as above.

2.3.2 Change the Class Name

To change the class name, complete the following steps:

1. In the class view window, right-click Class1 to display a context menu.
2. From the context menu, select Rename and change the class' name to CsAddPanel.
3. In the Solution Explorer, right-click the Class1.cs file to display a context menu.
4. From the context menu, select Rename and change the file's name to CsAddPanel.cs.
5. Double click CsAddPanel.cs to open it for editing.

2.3.3 Add Code

The Add Panel project is different from Hello World because it is automatically invoked when Revit runs. Use the IExternalApplication interface for this project. The IExternalApplication interface contains two abstract methods, OnStartup() and OnShutdown(). For more information about IExternalApplication, refer to the Add-in Integration chapter.

Add the following code for the ribbon panel:

Code Region 2-5: Adding a ribbon panel

```
public class CsAddpanel : Autodesk.Revit.UI.IExternalApplication
{
    public Autodesk.Revit.UI.Result OnStartup(UIControlledApplication application)
    {
        // add new ribbon panel
        RibbonPanel ribbonPanel = application.CreateRibbonPanel("NewRibbonPanel");
```

```
        //Create a push button in the ribbon panel "NewRibbonPanel"
        //the add-in application "HelloWorld" will be triggered when button is pushed

        PushButton pushButton = ribbonPanel.AddItem(new PushButtonData("HelloWorld",
            "HelloWorld", @"D:\HelloWorld.dll", "HelloWorld.CsHelloWorld")) as PushButton;

        // Set the large image shown on button
        Uri uriImage = new Uri(@"D:\Sample\HelloWorld\bin\Debug\39-Globe_32x32.png");
        BitmapImage largeImage = new BitmapImage(uriImage);
        pushButton.LargeImage = largeImage;

        return Result.Succeeded;
    }

    public Result OnShutdown(UIControlledApplication application)
    {
        return Result.Succeeded;
    }
}
```

2.3.4 Build the Application

After completing the code, build the application. From the Build menu, click Build Solution. Output from the build appears in the Output window indicating that the project compiled without errors. AddPanel.dll is located in the project output directory.

2.3.5 Create the .addin manifest file

To invoke the application in Revit, create a manifest file to register it into Revit.

1. Create a new text file using Notepad.
2. Add the following text to the file:

Code Region 2-6: Creating a .addin file for an external application

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
<AddIn Type="Application">
    <Name>SampleApplication</Name>
    <Assembly>D:\Sample\AddPanel\AddPanel\bin\Debug\AddPanel.dll</Assembly>
    <AddInId>604B1052-F742-4951-8576-C261D1993107</AddInId>
    <FullClassName>AddPanel.CsAddPanel</FullClassName>
  </AddIn>
</RevitAddIns>
```

3. Save the file as HelloWorldRibbon.addin and put it in the following location:

- For Windows XP - C:\Documents and Settings\All Users\Application Data\Autodesk\Revit\Addins\2011\
- For Vista/Windows 7 - C:\ProgramData\Autodesk\Revit\Addins\2011\

Note: The AddPanel.dll file is in the default file folder in a new folder called Debug (D:\Sample\HelloWorld\bin\Debug\AddPanel.dll). Use the file path to evaluate Assembly.

Refer to the Add-in Integration chapter for more information about .addin manifest files.

2.3.6 Debugging

To begin debugging, build the project, and run Revit. A new ribbon panel appears on the Add-Ins tab named NewRibbonPanel and Hello World appears as the only button on the panel, with a large globe image.

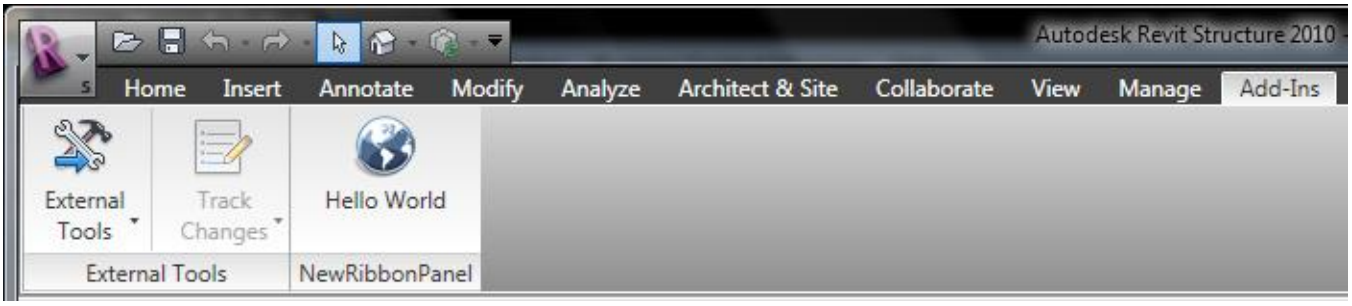


Figure 8: Add a new ribbon panel to Revit

Click Hello World to run the application and display the following dialog box.

Figure 9: Hello World dialog box

2.4 Walkthrough: Retrieve Selected Elements

This section introduces you to an add-in application that gets selected elements from Revit.

In add-in applications, you can perform a specific operation on a specific element. For example, you can get or change an element's parameter value. Complete the following steps to get a parameter value:

1. Create a new project and add the references as summarized in the previous walkthroughs.
2. Use the UIApplication.ActiveUIDocument.Selection.Elements property to retrieve the selected object.

The selected object is a Revit SelElementSet. Use the IEnumerator interface or foreach loop to search the ElementSet.

The following code is an example of how to retrieve selected elements.

Code Region 2-7: Retrieving selected elements

```
[Autodesk.Revit.Attributes.Transaction(TransactionMode.ReadOnly)]
[Autodesk.Revit.Attributes.Regeneration(RegenerationOption.Automatic)]
public class Document_Selection : IExternalCommand
{
    public Autodesk.Revit.UI.Result Execute(ExternalCommandData commandData,
        ref string message, ElementSet elements)
```

```
    {
        try
        {
            // Select some elements in Revit before invoking this command

            // Get the handle of current document.
            UIDocument uidoc = commandData.Application.ActiveUIDocument;
            // Get the element selection of current document.
            Selection selection = uidoc.Selection;
            ElementSet collection = selection.Elements;

            if (0 == collection.Size)
            {
                // If no elements selected.
                TaskDialog.Show("Revit","You haven't selected any elements.");
            }
            else
            {
                String info = "Ids of selected elements in the document are: ";
                foreach (Element elem in collection)
                {
                   info += "\n\t" + elem.Id.IntegerValue;
                }

                TaskDialog.Show("Revit",info);
            }
        }
        catch (Exception e)
        {
            message = e.Message;
            return Autodesk.Revit.UI.Result.Failed;
        }

        return Autodesk.Revit.UI.Result.Succeeded;
    }
}
```

After you get the selected elements, you can get the properties or parameters for the elements. For more information, see the Parameter chapter.

2.5 Walkthrough: Retrieve Filtered Elements

You can use a filter to select only elements that meet certain criteria. For more information on creating and using element filters, see Iterating the Elements Collection.

This example retrieves all the doors in the document and displays a dialog listing their ids.

Code Region 2-8: Retrieve filtered elements

```
// Create a Filter to get all the doors in the document
ElementClassFilter familyInstanceFilter = new ElementClassFilter(typeof(FamilyInstance));
```

```
ElementCategoryFilter doorsCategoryfilter =
    new ElementCategoryFilter(BuiltInCategory.OST_Doors);
LogicalAndFilter doorInstancesFilter =
    new LogicalAndFilter(familyInstanceFilter, doorsCategoryfilter);
FilteredElementCollector collector = new FilteredElementCollector(document);
ICollection<ElementId> doors = collector.WherePasses(doorInstancesFilter).ToElementIds();

String prompt = "The ids of the doors in the current document are:";
foreach(ElementId id in doors)
{
    prompt += "\n\t" + id.IntegerValue;
}

// Give the user some information
TaskDialog.Show("Revit",prompt);
```

3 Add-In Integration

Developers add functionality by creating and implementing External Commands and External Applications. Revit identifies the new commands and applications using the Revit.ini and .addin manifest files. The Revit.ini entries that were used in Revit 2010 and earlier will continue to work with legacy functionality in 2011, but we recommend developers transition to .addin files as .ini support will be removed in a future release. Additionally, some new functionality, such as the use of Updaters, will only work when using .addin manifest files.

- External Commands appear under the External Tools menu-button on the Add-Ins tab.
- External Applications are invoked when Revit starts up and unloaded when Revit shuts down

This chapter focuses on the following:

- Learning how to add functionality using External Commands and External Applications.
- How to access Revit events.
- How to customize the Revit UI.

3.1 Overview

The Revit Platform API is based on Revit application functionality. The Revit Platform API is composed of two class Libraries that only work when Revit is running.

The RevitAPI.dll contains methods used to access Revit's application, documents, elements, and parameters at the database level.

The RevitAPIUI.dll contains all API interfaces related to manipulation and customization of the Revit user interface, including:

- IExternalCommand and External Command related interfaces
- IExternalApplication and related interfaces
- Selection
- RibbonPanel, RibbonItem and subclasses
- TaskDialogs

As the following picture shows, Revit Architecture, Revit Structure, and Revit MEP are specific to Architecture, Structure, and MEP respectively.

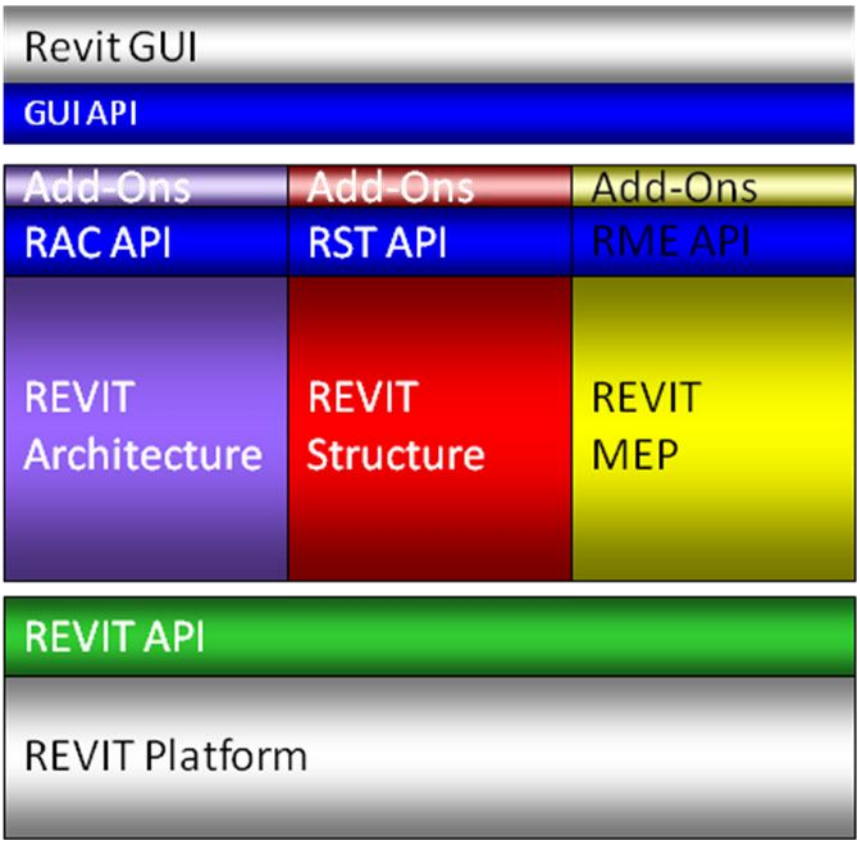


Figure 10: Revit, RevitAPI and Add-ins

To create a RevitAPI based add-in, you must provide specific entrypoint types in your add-in DLL. These entrypoint classes implement interfaces, either IExternalCommand or IExternalApplication. In this way, the add-in is run automatically on certain events or manually from the External Tools menu-button.

IExternalCommand, IExternalApplication, and other available Revit events for add-in integration are introduced in this chapter.

3.2 External Commands

Developers can add functionality by implementing External Commands which appear in the External Tools menu-button.

3.2.1 **Loading and Running External Commands**

When no other commands or edit modes are active in Revit, registered external commands are enabled. When a command is selected, a command object is created and its Execute() method is called. Once this method returns back to Revit, the command object is destroyed. As a result, data cannot persist in the object between command executions. However, there are other ways to save data between command executions; for example you can use the Revit shared parameters mechanism to store data in the Revit project.

You can add External Commands to the External Tools Panel under the External Tools menu-button, or as a custom ribbon panel on the Add-Ins tab. See the Walkthrough: Hello World and Walkthrough: Add Hello World Ribbon Panel for examples of these two approaches.

External tools and ribbon panels are initialized upon start up. The initialization steps are as follows:

- Revit reads manifest files and identifies:

- External Applications that can be invoked.
- External Tools that can be added to the Revit External Tools menu-button.

- External Application session adds panels and content to the Add-ins tab.

3.2.2 IExternalCommand

You create an external command by creating an object that implements the IExternalCommand interface. The IExternalCommand interface has one abstract method, Execute, which is the main method for external commands.

The Execute() method has three parameters:

- commandData (ExternalCommandData)
- message (String)
- elements (ElementSet)

3.2.2.1 commandData (ExternalCommandData)

The ExternalCommandData object contains references to Application and View which are required by the external command. All Revit data is retrieved directly or indirectly from this parameter in the external command.

For example, the following statement illustrates how to retrieve Autodesk.Revit.Document from the commandData parameter:

Code Region 3-1: Retrieving the Active Document

```
Document doc = commandData.Application.ActiveUIDocument.Document;
```

The following table illustrates the ExternalCommandData public properties

Property	Description
Application (Autodesk.Revit.UI.UIApplication)	Retrieves an object that represents the current UIApplication for external command.
Data (Autodesk.Revit.Collections.StringStringMap)	A data map that can be used to read and write data to the Revit journal file.
View (Autodesk.Revit.DB.View)	Retrieves an object that represents the View external commands work on.

Table 1: ExternalCommandData public properties

3.2.2.2 message (String):

Error messages are returned by an external command using the output parameter message. The string-type parameter is set in the external command process. When Autodesk.Revit.UI.Result.Failed or Autodesk.Revit.UI.Result.Cancelled is returned, and the message parameter is set, an error dialog appears.

The following code sample illustrates how to use the message parameter.

Code Region 3-2: Setting an error message string

```
class IExternalCommand_message : IExternalCommand
{
    public Autodesk.Revit.UI.Result Execute(
        Autodesk.Revit.ExternalCommandData commandData, ref string message,
        Autodesk.Revit.ElementSet elements)
```

```
    {
        message = "Could not locate walls for analysis.";
        return Autodesk.Revit.IExternalCommand.Result.Failed;
    }
}
```

Implementing the previous external command causes the following dialog box to appear:

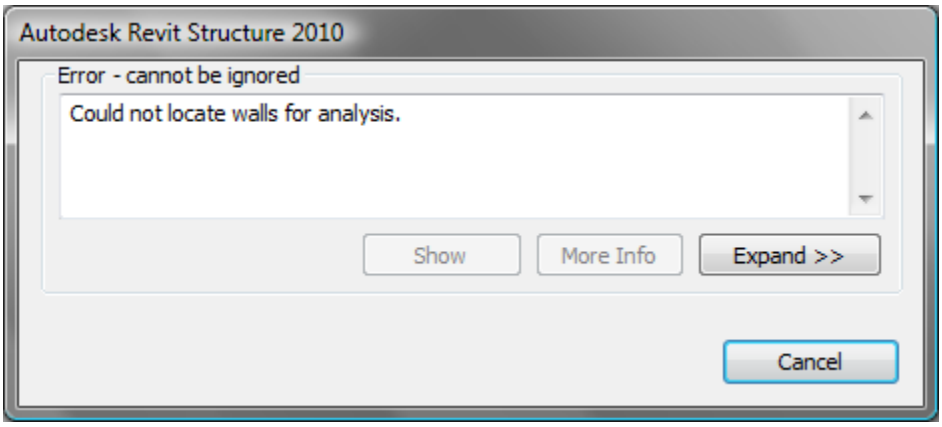


Figure 11: Error message dialog box

3.2.2.3 elements (ElementSet):

Whenever Autodesk.Revit.UI.Result.Failed or Autodesk.Revit.UI.Result.Canceled is returned and the parameter message is not empty, an error or warning dialog box appears. Additionally, if any elements are added to the elements parameter, these elements will be highlighted on screen. It is a good practice to set the message parameter whenever the command fails, whether or not elements are also returned.

The following code highlights pre-selected walls:

Code Region 3-3: Highlighting walls

```
class IExternalcommand_elements : IExternalCommand
{
    public Result Execute(
        Autodesk.Revit.UI.ExternalCommandData commandData, ref string message,
        Autodesk.Revit.DB.ElementSet elements)
    {
        message = "Please note the highlighted Walls.";
        FilteredElementCollector collector =
            new FilteredElementCollector(commandData.Application.ActiveUIDocument.Document);
        ICollection<Element> collection = collector.OfClass(typeof(Wall)).ToElements();
        foreach (Element e in collection)
        {
            elements.Insert(e);
        }

        return Result.Failed;
    }
}
```

The following picture displays the result of the previous code.

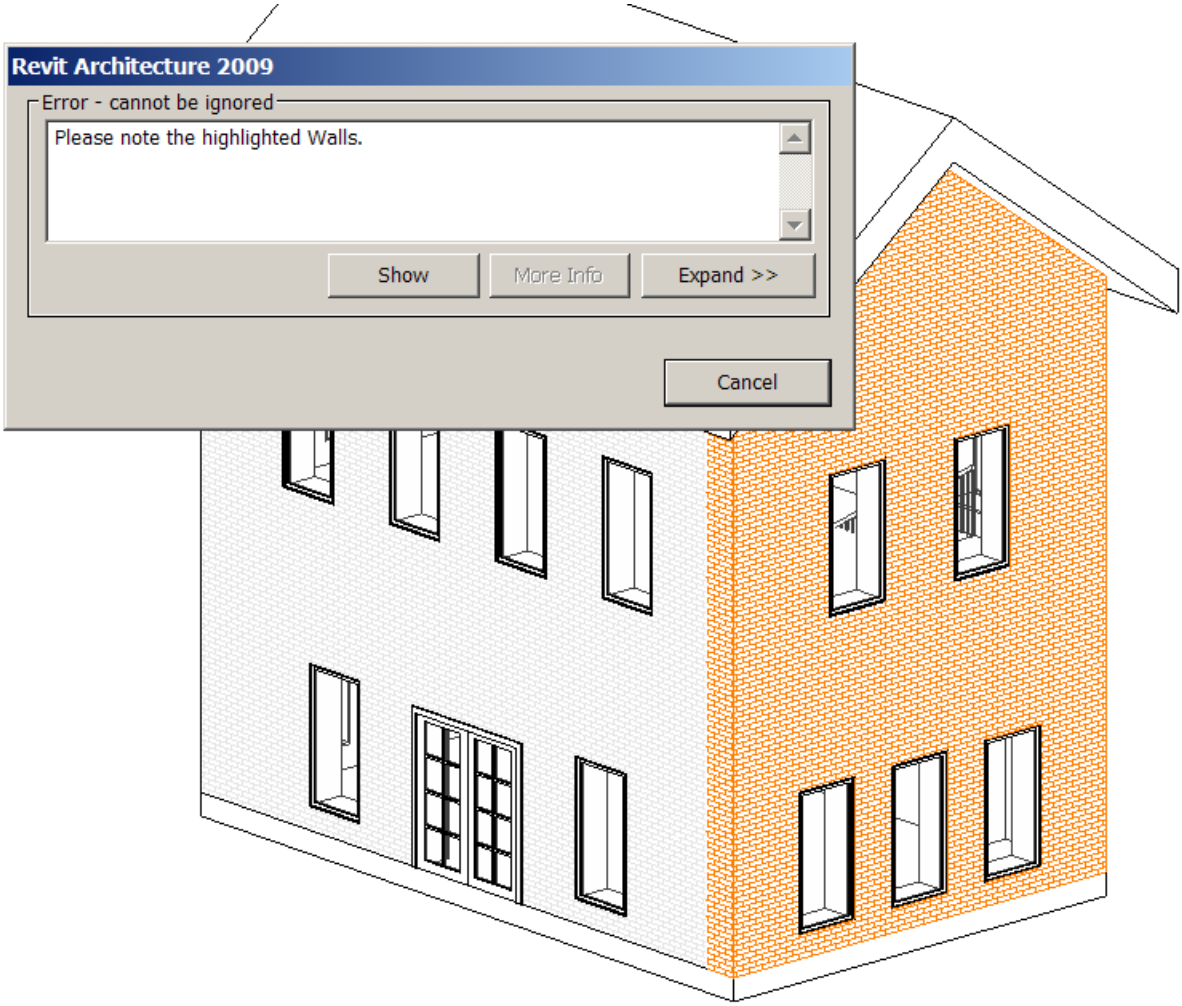


Figure 12: Error message dialog box and highlighted elements

3.2.2.4 **Return**

The Return result indicates that the execution failed, succeeded, or is canceled by the user. If it does not succeed, Revit reverses changes made by the external command.

Member Name	Description
Autodesk.Revit.UI.Result.Succeeded	The external command completed successfully. Revit keeps all changes made by the external command.
Autodesk.Revit.UI.Result.Failed	The external command failed to complete the task. Revit reverses operations performed by the external command. If the message parameter of Execute is set, Revit displays a dialog with the text "Error – cannot be ignored" .
Autodesk.Revit.UI.Result.Cancelled	The user cancelled the external command. Revit reverses changes made by the external command. If the message parameter of Execute is set, Revit displays a dialog with the text "Warning – can be ignored".

Table 2: IExternalCommand.Result

The following example displays a greeting message and allows the user to select the return value. Use the Execute() method as the entrance to the Revit application.

Code Region 3-4: Prompting the user

```
public Autodesk.Revit.UI.Result Execute(ExternalCommandData commandData,
    ref string message, ElementSet elements)
{
    try
    {
        Document doc = commandData.Application.ActiveUIDocument.Document;
        UIDocument uidoc = commandData.Application.ActiveUIDocument;
        // Delete selected elements
        ICollection<Autodesk.Revit.DB.ElementId> ids =
                    doc.Delete(uidoc.Selection.Elements);

        TaskDialog taskDialog = new TaskDialog("Revit");
        taskDialog.MainContent =
           ("Click Yes to return Succeeded. Selected members will be deleted.\n" +
            "Click No to return Failed.  Selected members will not be deleted.\n" +
            "Click Cancel to return Cancelled.  Selected members will not be deleted.");
        TaskDialogCommonButtons buttons = TaskDialogCommonButtons.Yes |
            TaskDialogCommonButtons.No | TaskDialogCommonButtons.Cancel;
        taskDialog.CommonButtons = buttons;
        TaskDialogResult taskDialogResult = taskDialog.Show();

        if (taskDialogResult == TaskDialogResult.Yes)
        {
            return Autodesk.Revit.UI.Result.Succeeded;
        }
        else if (taskDialogResult == TaskDialogResult.No)
        {
            elements = uidoc.Selection.Elements;
            message = "Failed to delete selection.";
            return Autodesk.Revit.UI.Result.Failed;
        }
        else
        {
            return Autodesk.Revit.UI.Result.Cancelled;
        }
    }
    catch
    {
        message = "Unexpected Exception thrown.";
        return Autodesk.Revit.UI.Result.Failed;
    }
}
```

3.2.2.5 **IExternalCommandAvailability**

This interface allows you control over whether or not an external command button may be pressed. The IsCommandAvailable interface method passes the application and a set of categories matching

the categories of selected items in Revit to your implementation. The typical use would be to check the selected categories to see if they meet the criteria for your command to be run.

In this example the accessibility check allows a button to be clicked when there is no active selection, or when at least one wall is selected:

Code Region 3-5: Setting Command Availability

```
public class SampleAccessibilityCheck : IExternalCommandAvailability
{
    public bool IsCommandAvailable(AutodeskAutodesk.Revit.UI.UIApplication applicationData,
         CategorySet selectedCategories)
    {
        // Allow button click if there is no active selection
        if (selectedCategories.IsEmpty)
            return true;
        // Allow button click if there is at least one wall selected
        foreach (Category c in selectedCategories)
        {
            if (c.Id.IntegerValue == (int)BuiltInCategory.OST_Walls)
            return true;
        }
        return false;
    }
}
```

3.3 External Application

Developers can add functionality through External Applications as well as External Commands. Ribbon panels are customized using the External Application. Ribbon panel buttons are bound to an External command.

3.3.1 IExternalApplication

To add an External Application to Revit, you create an object that implements the IExternalApplication interface.

The IExternalApplication interface has two abstract methods, OnStartup() and OnShutdown(), which you override in your external application. Revit calls OnStartup() when it starts, and OnShutdown() when it closes.

This is the OnStartup() and OnShutdown() abstract definition:

Code Region 3-6: OnShutdown() and OnStartup()

```
public interface IExternalApplication
{
   public Autodesk.Revit.UI.Result OnStartup(UIControlledApplication application);
   public Autodesk.Revit.UI.Result OnShutdown(UIControlledApplication application);
}
```

The UIControlledApplication parameter provides access to certain Revit events and allows customization of ribbon panels and controls. For example, the public event DialogBoxShowing of UIControlledApplication can be used to capture the event of a dialog being displayed. The following code snippet registers the handling function that is called right before a dialog is shown.

Code Region 3-7: DialogBoxShowing Event

```
application.DialogBoxShowing += new
         EventHandler<Autodesk.Revit.Events.DialogBoxShowingEventArgs>(AppDialogShowing);
```

The following code sample illustrates how to use the UIControlledApplication type to register an event handler and process the event when it occurs.

Code Region 3-8: Using ControlledApplication

```
public class Application_DialogBoxShowing : IExternalApplication
{
    // Implement the OnStartup method to register events when Revit starts.
    public Result OnStartup(UIControlledApplication application)
    {
        // Register related events
        application.DialogBoxShowing +=
      new EventHandler<Autodesk.Revit.UI.Events.DialogBoxShowingEventArgs>(AppDialogShowing);
        return Result.Succeeded;
    }

    // Implement this method to unregister the subscribed events when Revit exits.
    public Result OnShutdown(UIControlledApplication application)
    {
        // unregister events
        application.DialogBoxShowing -=
      new EventHandler<Autodesk.Revit.UI.Events.DialogBoxShowingEventArgs>(AppDialogShowing);
        return Result.Succeeded;
    }

    // The DialogBoxShowing event handler, which allow you to
    // do some work before the dialog shows
    void AppDialogShowing(object sender, DialogBoxShowingEventArgs args)
    {
        // Get the help id of the showing dialog
        int dialogId = args.HelpId;

        // Format the prompt information string
        String promptInfo = "A Revit dialog will be opened.\n";
        promptInfo += "The help id of this dialog is " + dialogId.ToString() + "\n";
        promptInfo += "If you don't want the dialog to open, please press cancel button";

        // Show the prompt message, and allow the user to close the dialog directly.
        TaskDialog taskDialog = new TaskDialog("Revit");
        taskDialog.MainContent = promptInfo;
        TaskDialogCommonButtons buttons = TaskDialogCommonButtons.Ok |
                                          TaskDialogCommonButtons.Cancel;
        taskDialog.CommonButtons = buttons;
        TaskDialogResult result = taskDialog.Show();
        if (TaskDialogResult.Cancel == result)
```

```
        {
            // Do not show the Revit dialog
            args.OverrideResult(1);
        }
        else
        {
            // Continue to show the Revit dialog
            args.OverrideResult(0);
        }
    }
}
```

-

3.4 Add-in Registration

External commands and external applications need to be registered in order to appear inside Revit. They can be registered in one of two ways: by adding them to the Revit.ini file or by adding them to a .addin manifest file. However, the Revit.ini method will be deprecated in a future release of Revit, making the manifest file the preferred method. For more information on registering via the Revit.ini file, see Registering Add-Ins Using Revit.ini in the appendix.

The order that external commands and applications are listed in Revit is determined by the order in which they are read in when Revit starts up. External commands and applications in the Revit.ini file are processed by Revit after commands and applications found in manifest files. The external commands and applications within the Revit.ini file are processed in the order listed in their respective sections.

3.4.1 Manifest Files

Starting with Revit 2011, the Revit API offers the ability to register API applications via a .addin manifest file. Manifest files are read automatically by Revit when they are placed in one of two locations on a user's system:

In a non-user-specific location in "application data":

- For Windows XP - C:\Documents and Settings\All Users\Application Data\Autodesk\Revit\Addins\2011\
- For Vista/Windows 7 - C:\ProgramData\Autodesk\Revit\Addins\2011\

In a user-specific location in "application data":

- For Windows XP - C:\Documents and Settings\<user>\Application Data\Autodesk\Revit\Addins\2011\
- For Vista/Windows 7 - C:\Users\<user>\AppData\Roaming\Autodesk\Revit\Addins\2011\

All files named .addin in these locations will be read and processed by Revit during startup. All of the files in both the user-specific location and the all users location are considered together and loaded in alphabetical order. If an all users manifest file shares the same name with a user-specific manifest file, the all users manifest file is ignored. Within each manifest file, the external commands and external applications are loaded in the order in which they are listed.

A basic file adding one ExternalCommand looks like this:

Code Region 3-9: Manifest .addin ExternalCommand

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
```

```
  <AddIn Type="Command">
    <Assembly>c:\MyProgram\MyProgram.dll</Assembly>
    <AddInId>76eb700a-2c85-4888-a78d-31429ecae9ed</AddInId>
    <FullClassName>Revit.Samples.SampleCommand</FullClassName>
    <Text>Sample command</Text>
    <VisibilityMode>NotVisibleInFamily</VisibilityMode>
    <VisibilityMode>NotVisibleInMEP</VisibilityMode>
    <AvailabilityClassName>Revit.Samples.SampleAccessibilityCheck</AvailabilityClassName>
    <LongDescription>
      <p>This is the long description for my command.</p>
      <p>This is another descriptive paragraph, with notes about how to use the command
properly.</p>
    </LongDescription>
    <TooltipImage>c:\MyProgram\Autodesk.jpg</TooltipImage>
    <LargeImage>c:\MyProgram\MyProgramIcon.png</LargeImage>
  </AddIn>
</RevitAddIns>
```

A basic file adding one ExternalApplication looks like this:

Code Region 3-10: Manifest .addin ExternalApplication

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
<AddIn Type="Application">
    <Name>SampleApplication</Name>
    <Assembly>c:\MyProgram\MyProgram.dll</Assembly>
    <AddInId>604B1052-F742-4951-8576-C261D1993107</AddInId>
    <FullClassName>Revit.Samples.SampleApplication</FullClassName>
  </AddIn>
</RevitAddIns>
```

Multiple AddIn elements may be provided in a single manifest file.

The following table describes the available XML tags:

Tag	Description
Assembly	The full path to the add-in assembly file. Required for all ExternalCommands and ExternalApplications.
FullClassName	The full name of the class in the assembly file which implements IExternalCommand or IExternalApplication. Required for all ExternalCommands and ExternalApplications.
AddInId	A GUID which represents the id of this particular application. AddInIds must be unique for a given session of Revit. Autodesk recommends you generate a unique GUID for each registered application or command. Required for all ExternalCommands and ExternalApplications.
Name	The name of application. Required; for ExternalApplications only.
Text	The name of the button. Optional; use this tag for ExternalCommands

	only. The default is "External Tool".
Description	Short description of the command, will be used as the button tooltip. Optional; use this tag for ExternalCommands only. The default is a tooltip with just the command text.
VisibilityMode	Provides the ability to specify if the command is visible in project documents, family documents, or no document at all. Also provides the ability to specify the discipline(s) where the command should be visible. Multiple values may be set for this option. Optional; use this tag for ExternalCommands only. The default is to display the command in all modes and disciplines, including when there is no active document. Previously written external commands which need to run against the active document should either be modified to ensure that the code deals with invocation of the command when there is no active document, or apply the NotVisibleWhenNoActiveDocument mode.
AvailabilityClassName	The full name of the class in the assembly file which implemented IExternalCommandAvailability. This class allows the command button to be selectively grayed out depending on context. Optional; use this tag for ExternalCommands only. The default is a command that is available whenever it is visible.
LargeImage	The icon to use for the button in the External Tools pulldown menu. Optional; use this tag for ExternalCommands only. The default is to show a button without an icon.
SmallImage	The icon to use if the button is promoted to the Quick Access Toolbar. Optional; use this tag for ExternalCommands only. The default is to show a Quick Access Toolbar button without an icon, which can be confusing to users.
LongDescription	Long description of the command, will be used as part of the button extended tooltip, shown when the mouse hovers over the command for a longer amount of time. Optional; use this tag for ExternalCommands only. If this property and TooltipImage are not supplied, the button will not have an extended tooltip.
TooltipImage	An image file to show as a part of the button extended tooltip, shown when the mouse hovers over the command for a longer amount of time. Optional; use this tag for ExternalCommands only. If this property and TooltipImage are not supplied, the button will not have an extended tooltip.
LanguageType	Localization setting for Text, Description, LargeImage, LongDescription, and TooltipImage of external tools buttons. Revit will load the resource values from the specified language resource dll. The value can be one of the eleven languages supported by Revit. If no LanguageType is specified, the language resource which the current session of Revit is using will be automatically loaded. For more details see the section on Localization.

3.4.2 .NET Add-in Utility for manifest files

The .NET utility DLL RevitAddInUtility.dll offers a dedicated API capable of reading, writing and modifying Revit Add-In manifest files. It is intended for use from product installers and scripts. Consult the API documentation in the RevitAddInUtility.chm help file in the SDK installation folder.

Code Region 3-11: Creating and editing a manifest file

```
//create a new addin manifest
RevitAddInManifest Manifest = new RevitAddInManifest();

//create an external command
RevitAddInCommand command1 = new RevitAddInCommand("full path\\assemblyName.dll",
                Guid.NewGuid(), "namespace.className");
command1.Description = "description";
command1.Text = "display text";

// this command only visible in Revit MEP, Structure, and only visible
// in Project document or when no document at all
command1.VisibilityMode = VisibilityMode.NotVisibleInArchitecture |
                          VisibilityMode.NotVisibleInFamily;

//create an external application
RevitAddInApplication application1 = new RevitAddInApplication("appName",
     "full path\\assemblyName.dll", Guid.NewGuid(), "namespace.className");

//add both command(s) and application(s) into manifest
Manifest.AddInCommands.Add(command1);
Manifest.AddInApplications.Add(application1);

//save manifest to a file
RevitProduct revitProduct1 = RevitProductUtility.GetAllInstalledRevitProducts()[0];
Manifest.SaveAs(revitProduct1.AllUsersAddInFolder + "\\RevitAddInUtilitySample.addin");
```

Code Region 3-12: Reading an existing manifest file

```
RevitProduct revitProduct1 = RevitProductUtility.GetAllInstalledRevitProducts()[0];

RevitAddInManifest revitAddInManifest =
                Autodesk.RevitAddIns.AddInManifestUtility.GetRevitAddInManifest(
                    revitProduct1.AllUsersAddInFolder + "\\RevitAddInUtilitySample.addin");
```

3.5 Localization

You can let Revit localize the user-visible resources of an external command button (including Text, large icon image, long and short descriptions and tooltip image). You will need to create a .NET Satellite DLL which contains the strings, images, and icons for the button. Then change the values of the tags in the .addin file to correspond to the names of resources in the Satellite dll, but prepended with the @character. So the tag:

Code Region 3-13: Non-localized Text Entry

```
<Text>Extension Manager</Text>
```

Becomes:

Code Region 3-14: Localized Text Entry

```
<Text>@ExtensionText</Text>
```

where ExtensionText is the name of the resource found in the Satellite DLL.

The Satellite DLLs are expected to be in a directory with the name of the language of the language-culture, such as en or en-US. The directory should be located in the directory that contains the add-in assembly. See http://msdn.microsoft.com/en-us/library/e9zazcx5.aspx to create managed Satellite DLLs.

You can force Revit to use a particular language resource DLL, regardless of the language of the Revit session, by specifying the language and culture explicitly with a LanguageType tag.

Code Region 3-15: Using LanguageType Tag

```
<LanguageType>English_USA</LanguageType>
```

For example, the entry above would force Revit to always load the values from the en-US Satellite DLL and to ignore the current Revit language and culture settings when considering the localizable members of the external command manifest file.

Revit supports the 11 languages defined in the Autodesk.Revit.ApplicationServices.LanguageType enumerated type: English_USA, German, Spanish, French, Italian, Dutch, Chinese_Simplified, Chinese_Traditional, Japanese, Korean, and Russian.

3.6 Attributes

The Revit API provides several attributes for configuring ExternalCommand and ExternalApplication behavior.

3.6.1 RegenerationAttribute

The custom attribute Autodesk.Revit.Attributes.RegenerationAttribute must be applied to your implementation class of the IExternalCommand interface and IExternalApplication interface to control the regeneration behavior for the external command and external application. There is no default for this option.

This mode controls whether or not Revit automatically regenerates after every model modification. There are two options:

- **RegenerationOption.Automatic** - Revit will regenerate after every model change. Regeneration and update can be suspended using SuspendUpdating for some operations, but in general the performance of multiple modifications within the same file will be slower

than RegenerationOption.Manual. This mode is provided for behavioral equivalence with Revit 2010 and earlier; it is obsolete and will be removed in a future release.

- **RegenerationOption.Manual** - Revit will not regenerate after every model change. Instead, you use the regeneration APIs to regenerate the document as desired. SuspendUpdating blocks should not be used. Performance of multiple modifications of the Revit document should be faster than RegenerationOption.Automatic. Because this mode suspends all updates to the document, your application should not read data from the document after it has been modified until the document has been regenerated. Manual will become the only regeneration option in a future release.

For example, to set an external command to use manual regeneration mode:

Code Region 3-16: Using Manual Regeneration Mode

```
[Regeneration(RegenerationOption.Manual)]
public class Command : IExternalCommand
{
     public Autodesk.Revit.IExternalCommand.Result Execute(
         Autodesk.Revit.ExternalCommandData commandData,
         ref string message, Autodesk.Revit.DB.ElementSet elements)
     {
         // Command implementation, which modifies the document and calls regeneration APIs
         // when needed.
     }
}
```

Note: When using manual regeneration mode, use the Document.Regenerate() and Document.AutoJoinElements() methods to update elements. See the section on Element Geometry and AnalyticalModel at the end of the Transactions chapter.

To set an external application to use automatic mode:

Code Region 3-17: Using Automatic Regeneration Mode

```
[Regeneration(RegenerationOption.Automatic)]
public class Application : IExternalApplication
{
    public Autodesk.Revit.UI.Result OnStartup(ControlledApplication application)
    {
        // OnStartup implementation
    }
    public Autodesk.Revit.UI.Result OnShutdown(ControlledApplication application)
    {
        // OnShutdown implementation
    }
}
```

Note: The regeneration mode used for code executed during events and updater callbacks will be the same mode applied to the ExternalApplication or ExternalCommand during which the event or updater was registered.

3.6.2 TransactionAttribute

The custom attribute Autodesk.Revit.Attributes.TransactionMode must be applied to your implementation class of the IExternalCommand interface to control transaction behavior for external command. There is no default for this option. This mode controls how the API framework expects transactions to be used when the command is invoked. The supported values are:

- **TransactionMode.Automatic** - Revit will create a transaction in the active document before the external command is executed and the transaction will be committed or rolled back after the command is completed (based upon the return value of the ExternalCommand callback). The command cannot create and start its own Transactions, but it can create SubTransactions. The command must report its success or failure status with the Result return value.
- **TransactionMode.Manual** - Revit will not create a transaction (but it will create an outer transaction group to roll back all changes if the external command returns a failure). Instead, you may use combinations of Transactions, SubTransactions, and TransactionGroups as you please. You will have to follow all rules regarding use of transactions and related classes. You will have to give your transactions names which will then appear in the Undo menu. Revit will check that all transactions (also groups and sub-transactions) are properly closed upon return from an external command. If not, Revit will discard all changes made to the model.
- **TransactionMode.ReadOnly** - No transaction (nor group) will be created, and no transaction may be created for the lifetime of the command. The External Command may only use methods that read from the model. Exceptions will be thrown if the command either tries to start a transaction (or group) or attempts to write to the model.

In all three modes, the TransactionMode applies only to the active document. You may open other documents during the course of the command, and you may have complete control over the creation and use of Transactions, SubTransactions, and TransactionGroups on those other documents (even in ReadOnly mode).

For example, to set an external command to use automatic transaction mode:

Code Region 3-18: TransactionAttribute

```
[Transaction(TransactionMode.Automatic)]
public class Command : IExternalCommand
{
    public Autodesk.Revit.IExternalCommand.Result Execute(
        Autodesk.Revit.ExternalCommandData commandData,
        ref string message, Autodesk.Revit.DB.ElementSet elements)
    {
        // Command implementation, which modifies the active document directly
        // and no need to start/commit transaction.
    }
}
```

See the chapter on Transactions for more information.

3.6.3 JournalingAttribute

The custom attribute Autodesk.Revit.Attributes.JournalingAttribute can optionally be applied to your implementation class of the IExternalCommand interface to control the journaling behavior during the external command execution. There are two options for journaling:

- **JournalMode.NoCommandData** - Contents of the ExternalCommandData.Data map are not written to the Revit journal. This option allows Revit API calls to write to the journal as needed. This option allows commands which invoke the Revit UI for selection or responses to task dialogs to replay correctly.
- **JournalMode.UsingCommandData** - Uses the "StringStringMap" supplied in the command data. This will hide all Revit journal entries between the external command invocation and the StringStringMap entry. Commands which invoke the Revit UI for selection or responses to task dialogs may not replay correctly. This is the default if the JournalingAttribute is not specified.

Code Region 3-19: JournalingAttribute

```
[Journaling(JournalingMode.UsingCommandData)]
public class Command : IExternalCommand
{
    public Autodesk.Revit.IExternalCommand.Result Execute(
        Autodesk.Revit.ExternalCommandData commandData,
        ref string message, Autodesk.Revit.DB.ElementSet elements)
    {
        return Autodesk.Revit.IExternalCommand.Result.Succeeded;
    }
}
```

3.7 Revit Exceptions

When API methods encounter a non-fatal error, they throw an exception. Exceptions should be caught by Revit add-ins. The Revit API help file specifies exceptions that are typically encountered for specific methods. All Revit API methods throw a subclass of Autodesk.Revit.Exceptions.ApplicationException. These exceptions closely mirror standard .NET exceptions such as:

- ArgumentException
- InvalidOperationException
- FileNotFoundException

However, some of these subclasses are unique to Revit:

- AutoJoinFailedException
- RegenerationFailedException
- ModificationOutsideTransactionException

In addition, there is a special exception type called InternalException, which represents a failure path which was not anticipated. Exceptions of this type carry extra diagnostic information which can be passed back to Autodesk for diagnosis.

3.8 Ribbon Panels and Controls

Revit provides API solutions to integrate custom ribbon panels and controls. These APIs are used with IExternalApplication. Custom ribbon panels will appear on the Add-Ins tab in Revit by default, but they can also be added to the Analyze tab.

Panels can include buttons, both large and small, which can be either simple push buttons, pulldown buttons containing multiple commands, or split buttons which are pulldown buttons with a default push button attached. In addition to buttons, panels can include radio groups, combo boxes and text boxes. Panels can also include vertical separators to help separate commands into logical groups. Finally, panels can include a slide out control accessed by clicking on the bottom of the panel.

Please see the Ribbon Guidelines in the API User Interface Guidelines appendix for information on developing a user interface that is compliant with the standards used by Autodesk.

3.8.1 Create a New Ribbon Panel and Controls

The following image shows a ribbon panel using various ribbon panel controls. The following sections describe these controls in more detail and provide code samples for creating each portion of the ribbon.

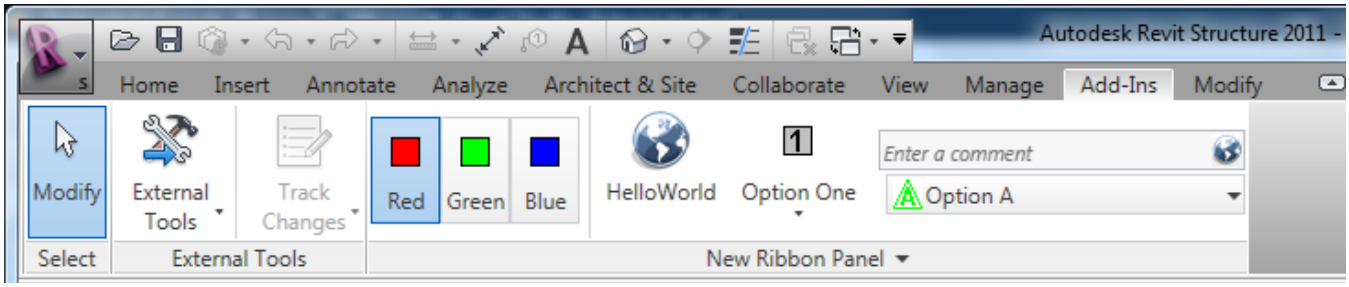


Figure 13: New ribbon panel and controls

The following code outlines the steps taken to create the ribbon panel pictured above. Each of the functions called in this sample is provided in subsequent samples later in this section. Those samples assume that there is an assembly located at D:\ Sample\HelloWorld\bin\Debug\Hello.dll which contains the External Command Types:

- Hello.HelloButton
- Hello.HelloOne
- Hello.HelloTwo
- Hello.HelloThree
- Hello.HelloA
- Hello.HelloB
- Hello.HelloC
- Hello.HelloRed
- Hello.HelloBlue
- Hello.HelloGreen

Code Region 3-20: Ribbon panel and controls

```
public Result OnStartup(Autodesk.Revit.UI.UIControlledApplication app)
{
    RibbonPanel panel = app.CreateRibbonPanel("New Ribbon Panel");

    AddRadioGroup(panel);
    panel.AddSeparator();
    AddPushButton(panel);
    AddSplitButton(panel);
    AddStackedButtons(panel);
    AddSlideOut(panel);

    return Result.Succeeded;
}
```

3.8.2 **Ribbon Panel**

Custom ribbon panels can be added to the Add-Ins tab (the default) or the Analyze tab. There are various types of ribbon controls that can be added to ribbon panels which are discussed in more detail in the next section. All ribbon controls have some common properties and functionality.

3.8.2.1 **Ribbon Control Classes**

Each ribbon control has two classes associated with it – one derived from RibbonItemData that is used to create the control (i.e. SplitButtonData) and add it to a ribbon panel and one derived from RibbonItem (i.e. SplitButton) which represents the item after it is added to a panel. The properties available from RibbonItemData (and the derived classes) are also available from RibbonItem (and the corresponding derived classes). These properties can be set prior to adding the control to the panel or can be set using the RibbonItem class after it has been added to the panel.

3.8.2.2 **Tooltips**

Most controls can have a tooltip set (using the ToolTip property) which is displayed when the user moves the mouse over the control. When the user hovers the mouse over a control for an extended period of time, an extended tooltip will be displayed using the LongDescription and the ToolTipImage properties. If neither LongDescription nor ToolTipImage are set, the extended tooltip is not displayed. If no ToolTip is provided, then the text of the control (RibbonItem.ItemText) is displayed when the mouse moves over the control.

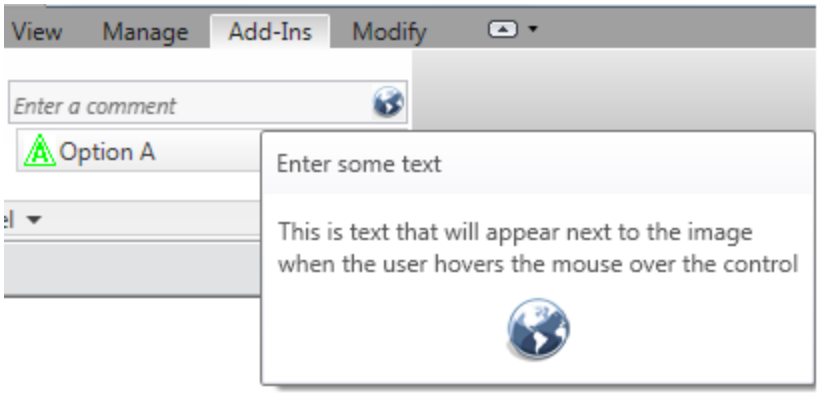


Figure 14: Extended Tooltip

3.8.2.3 **Associating images with controls**

All of these controls can have an image associated with them using the LargeImage property. The best size for images associated with large controls, such as non-stacked ribbon and drop-down buttons, is 32x32 pixels, but larger images will be adjusted to fit the button. Stacked buttons and small controls such as text boxes and combo boxes should have a 16x16 pixel image set. Large buttons should also have a 16x16 pixel image set for the Image property. This image is used if the command is moved to the Quick Access Toolbar. If the Image property is not set, no image will be displayed if the command is moved to the Quick Access Toolbar. Note that if an image larger than 16x16 pixels is used, it will NOT be adjusted to fit the toolbar.

The ToolTipImage will be displayed below the LongDescription in the extended tooltip, if provided. There is no recommended size for this image.

3.8.2.4 **Ribbon control availability**

Ribbon controls can be enabled or disabled with the RibbonItem.Enabled property or made visible or invisible with the RibbonItem.Visible property.

3.8.3 Ribbon Controls

In addition to the following controls, vertical separators can be added to ribbon panels to group related sets of controls.

3.8.3.1 Push Buttons

There are three types of buttons you can add to a panel: simple push buttons, drop-down buttons, and split buttons. The HelloWorld button in Figure 13 is a push button. When the button is pressed, the corresponding command is triggered.

In addition to the Enabled property, PushButton has the AvailabilityClassName property which can be used to set the name of an IExternalCommandAvailability interface that controls when the command is available.

Code Region 3-21: Adding a push button

```
private void AddPushButton(RibbonPanel panel)
{
    PushButton pushButton = panel.AddItem(new PushButtonData("HelloWorld",
        "HelloWorld", @"D:\HelloWorld.dll", "HelloWorld.CsHelloWorld")) as PushButton;

    pushButton.ToolTip = "Say Hello World";
    // Set the large image shown on button
    pushButton.LargeImage =
       new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\39-Globe_32x32.png"));
}
```

3.8.3.2 Drop-down buttons

Drop-down buttons expand to display two or more commands in a drop-down menu. In the Revit API, drop-down buttons are referred to as PulldownButtons. Horizontal separators can be added between items in the drop-down menu.

Each command in a drop-down menu can also have an associated LargeImage as shown in the example above.

3.8.3.3 Split buttons

Split buttons are drop-down buttons with a default push button attached. The top half of the button works like a push button while the bottom half functions as a drop-down button. The Option One button in Figure 13 is a split button.

Initially, the push button will be the top item in the drop-down list. However, by using the IsSynchronizedWithCurrentItem property, the default command (which is displayed as the push button top half of the split button) can be synchronized with the last used command. By default it will be synched. Selecting Option Two in the split button from Figure 13 above yields:

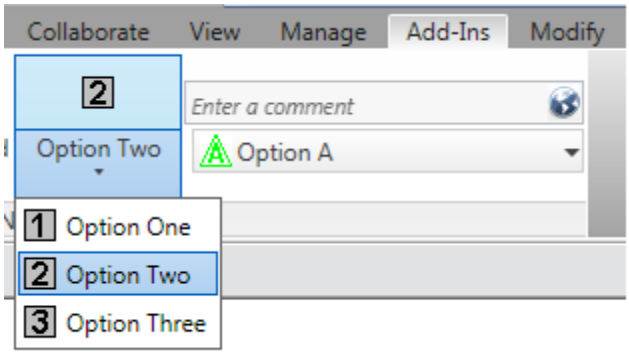


Figure 15: Split button synchronized with current item

Note that the ToolTip, ToolTipImage and LongDescription properties for SplitButton are ignored. The tooltip for the current push button is shown instead.

Code Region 3-22: Adding a split button

```
private void AddSplitButton(RibbonPanel panel)
{
    string assembly = @"D:\Sample\HelloWorld\bin\Debug\Hello.dll";

    // create push buttons for split button drop down
    PushButtonData bOne = new PushButtonData("ButtonNameA", "Option One",
        assembly, "Hello.HelloOne");
    bOne.LargeImage =
        new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\One.bmp"));

    PushButtonData bTwo = new PushButtonData("ButtonNameB", "Option Two",
        assembly, "Hello.HelloTwo");
    bTwo.LargeImage =
        new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Two.bmp"));

    PushButtonData bThree = new PushButtonData("ButtonNameC", "Option Three",
        assembly, "Hello.HelloThree");
    bThree.LargeImage =
        new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Three.bmp"));

    SplitButtonData sb1 = new SplitButtonData("splitButton1", "Split");
    SplitButton sb = panel.AddItem(sb1) as SplitButton;
    sb.AddPushButton(bOne);
    sb.AddPushButton(bTwo);
    sb.AddPushButton(bThree);
}
```

3.8.3.4 **Radio buttons**

A radio button group is a set of mutually exclusive toggle buttons; only one can be selected at a time. After adding a RadioButtonGroup to a panel, use the AddItem() or AddItems() methods to add toggle buttons to the group. Toggle buttons are derived from PushButton. The RadioButtonGroup.Current property can be used to access the currently selected button.

Note that tooltips do not apply to radio button groups. Instead, the tooltip for each toggle button is displayed as the mouse moves over the individual buttons.

Code Region 3-23: Adding radio button group

```
private void AddRadioGroup(RibbonPanel panel)
{
    // add radio button group
    RadioButtonGroupData radioData = new RadioButtonGroupData("radioGroup");
    RadioButtonGroup radioButtonGroup = panel.AddItem(radioData) as RadioButtonGroup;

    // create toggle buttons and add to radio button group
    ToggleButtonData tb1 = new ToggleButtonData("toggleButton1", "Red");
    tb1.ToolTip = "Red Option";
    tb1.LargeImage = new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Red.bmp"));
    ToggleButtonData tb2 = new ToggleButtonData("toggleButton2", "Green");
    tb2.ToolTip = "Green Option";
    tb2.LargeImage = new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Green.bmp"));
    ToggleButtonData tb3 = new ToggleButtonData("toggleButton3", "Blue");
    tb3.ToolTip = "Blue Option";
    tb3.LargeImage = new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Blue.bmp"));
    radioButtonGroup.AddItem(tb1);
    radioButtonGroup.AddItem(tb2);
    radioButtonGroup.AddItem(tb3);
}
```

3.8.3.5 **Text box**

A text box is an input control for users to enter text. The image for a text box can be used as a clickable button by setting the ShowImageAsButton property to true. The default is false. The image is displayed to the left of the text box when ShowImageAsButton is false, and displayed at the right end of the text box when it acts as a button, as in Figure 13.

The text entered in the text box is only accepted if the user hits the Enter key or if they click the associated image when the image is shown as a button. Otherwise, the text will revert to its previous value.

In addition to providing a tooltip for a text box, the PromptText property can be used to indicate to the user what type of information to enter in the text box. Prompt text is displayed when the text box is empty and does not have keyboard focus. This text is displayed in italics. The text box in Figure 13 has the prompt text "Enter a comment".

The width of the text box can be set using the Width property. The default is 200 device-independent units.

The TextBox.EnterPressed event is triggered when the user presses enter, or when they click on the associated image for the text box when ShowImageAsButton is set to true. When implementing an EnterPressed event handler, cast the sender object to TextBox to get the value the user has entered as shown in the following example.

Code Region 3-24: TextBox.EnterPressed event handler

```
void ProcessText(object sender, Autodesk.Revit.UI.Events.TextBoxEnterPressedEventArgs args)
{
    // cast sender as TextBox to retrieve text value
    TextBox textBox = sender as TextBox;
```

```
    string strText = textBox.Value as string;
}
```

The inherited ItemText property has no effect for TextBox. The user-entered text can be obtained from the Value property, which must be converted to a string.

See the section on stacked ribbon items for an example of adding a TextBox to a ribbon panel, including how to register the event above.

3.8.3.6 **Combo box**

A combo box is a pulldown with a set of selectable items. After adding a ComboBox to a panel, use the AddItem() or AddItems() methods to add ComboBoxMembers to the list.

Separators can also be added to separate items in the list or members can be optionally grouped using the ComboBoxMember.GroupName property. All members with the same GroupName will be grouped together with a header that shows the group name. Any items not assigned a GroupName will be placed at the top of the list. Note that when grouping items, separators should not be used as they will be placed at the end of the group rather than in the order they are added.

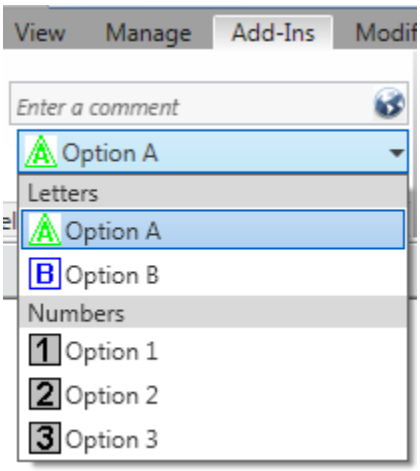


Figure 16: Combo box with grouping

ComboBox has three events:

- CurrentChanged – triggered when the current item of the ComboBox is changed
- DropDownClosed – triggered when the drop-down of the ComboBox is closed
- DropDownClosed – triggered when the drop-down of the ComboBox is opened

See the code region in the following section on stacked ribbon items for a sample of adding a ComboBox to a ribbon panel.

3.8.3.7 **Stacked Panel Items**

To conserve panel space, you can add small panel items in stacks of two or three. Each item in the stack can be a push button, a drop-down button, a combo box or a text box. Radio button groups and split buttons cannot be stacked. Stacked buttons should have an image associated through their Image property, rather than LargeImage. A 16x16 image is ideal for small stacked buttons.

The following example produces the stacked text box and combo box in Figure 13.

Code Region 3-25: Adding a text box and combo box as stacked items

```
private void AddStackedButtons(RibbonPanel panel)
{
    ComboBoxData cbData = new ComboBoxData("comboBox");
```

```
    TextBoxData textData = new TextBoxData("Text Box");
    textData.Image =
        new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\39-Globe_16x16.png"));
    textData.Name = "Text Box";
    textData.ToolTip = "Enter some text here";
    textData.LongDescription = "<p>This is text that will appear next to the image</p>"
        + "<p>when the user hovers the mouse over the control</p>";
    textData.ToolTipImage =
        new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\39-Globe_32x32.png"));

    IList<RibbonItem> stackedItems = panel.AddStackedItems(textData, cbData);
    if (stackedItems.Count > 1)
    {
        TextBox tBox = stackedItems[0] as TextBox;
        if (tBox != null)
        {
            tBox.PromptText = "Enter a comment";
            tBox.ShowImageAsButton = true;
            tBox.ToolTip = "Enter some text";
            // Register event handler ProcessText
            tBox.EnterPressed +=
        new EventHandler<Autodesk.Revit.UI.Events.TextBoxEnterPressedEventArgs>(ProcessText);
        }

        ComboBox cBox = stackedItems[1] as ComboBox;
        if (cBox != null)
        {
            cBox.ItemText = "ComboBox";
            cBox.ToolTip = "Select an Option";
            cBox.LongDescription = "Select a number or letter";

            ComboBoxMemberData cboxMemDataA = new ComboBoxMemberData("A", "Option A");
            cboxMemDataA.Image =
                new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\A.bmp"));
            cboxMemDataA.GroupName = "Letters";
            cBox.AddItem(cboxMemDataA);

            ComboBoxMemberData cboxMemDataB = new ComboBoxMemberData("B", "Option B");
            cboxMemDataB.Image =
                new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\B.bmp"));
            cboxMemDataB.GroupName = "Letters";
            cBox.AddItem(cboxMemDataB);

            ComboBoxMemberData cboxMemData = new ComboBoxMemberData("One", "Option 1");
            cboxMemData.Image =
                new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\One.bmp"));
            cboxMemData.GroupName = "Numbers";
            cBox.AddItem(cboxMemData);
```

```
            ComboBoxMemberData cboxMemData2 = new ComboBoxMemberData("Two", "Option 2");
            cboxMemData2.Image =
                new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Two.bmp"));
            cboxMemData2.GroupName = "Numbers";
            cBox.AddItem(cboxMemData2);
            ComboBoxMemberData cboxMemData3 = new ComboBoxMemberData("Three", "Option 3");
            cboxMemData3.Image =
                new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Three.bmp"));
            cboxMemData3.GroupName = "Numbers";
            cBox.AddItem(cboxMemData3);
        }
    }
}
```

3.8.3.8 **Slide-out panel**

Use the RibbonPanel.AddSlideOut() method to add a slide out to the bottom of the ribbon panel. When a slide-out is added, an arrow is shown on the bottom of the panel, which when clicked will display the slide-out. After calling AddSlideOut(), subsequent calls to add new items to the panel will be added to the slide-out, so the slide-out must be added after all other controls have been added to the ribbon panel.

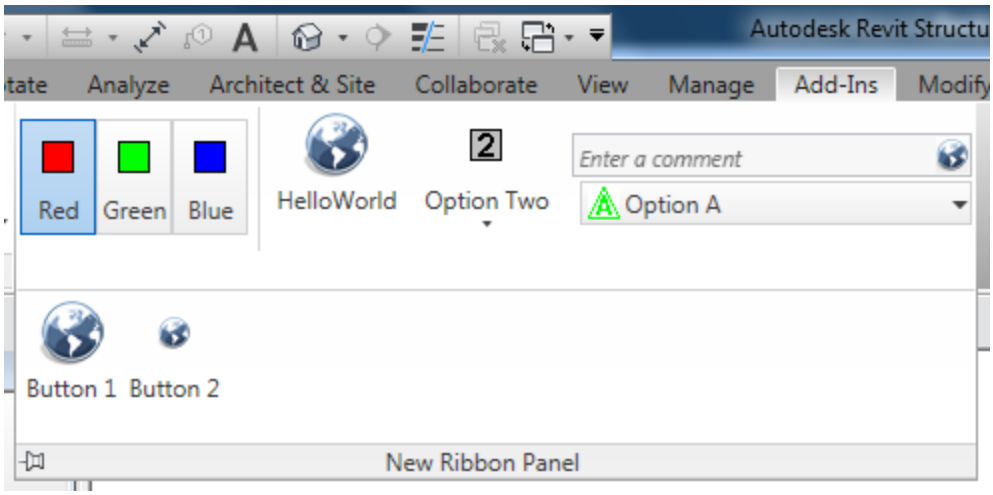


Figure 17: Slide-out

The following example produces the slide-out shown above:

Code Region 3-26: TextBox.EnterPressed event handler

```
private static void AddSlideOut(RibbonPanel panel)
{
    string assembly = @"D:\Sample\HelloWorld\bin\Debug\Hello.dll";

    panel.AddSlideOut();

    // create some controls for the slide out
    PushButtonData b1 = new PushButtonData("ButtonName1", "Button 1",
        assembly, "Hello.HelloButton");
    b1.LargeImage =
        new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\39-Globe_32x32.png"));
```

```
    PushButtonData b2 = new PushButtonData("ButtonName2", "Button 2",
        assembly, "Hello.HelloTwo");
    b2.LargeImage =
        new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\39-Globe_16x16.png"));

    // items added after call to AddSlideOut() are added to slide-out automatically
    panel.AddItem(b1);
    panel.AddItem(b2);
}
```

3.9 Revit-style Task Dialogs

A TaskDialog is a Revit-style alternative to a simple Windows MessageBox. It can be used to display information and receive simple input from the user. It has a common set of controls that are arranged in a standard order to assure consistent look and feel with the rest of Revit.

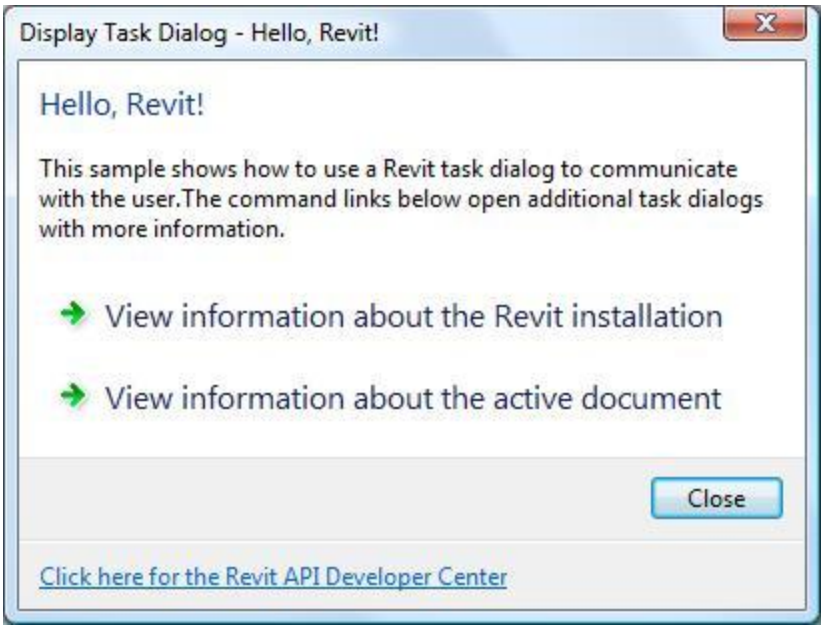


Figure 18: Revit-style Task Dialog

There are two ways to create and show a task dialog to the user. The first option is to construct the TaskDialog, set its properties individually, and use the instance method Show() to show it to the user. The second is to use one of the static Show() methods to construct and show the dialog in one step. When you use the static methods only a subset of the options can be specified.

Please see the Task Dialog section of the API User Interface Guidelines appendix for information on developing a task dialog that is compliant with the standards used by Autodesk.

The following example shows how to create and display the task dialog shown above.

Code Region 3-27: Displaying Revit-style TaskDialog

```
[Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.Automatic)]
[Autodesk.Revit.Attributes.Regeneration(Autodesk.Revit.Attributes.RegenerationOption.Manual)]
class TaskDialogExample : IExternalCommand
{
    public Autodesk.Revit.UI.Result Execute(ExternalCommandData commandData,
        ref string message, Autodesk.Revit.DB.ElementSet elements)
```

```
    {
        // Get the application and document from external command data.
        Application app = commandData.Application.Application;
        Document activeDoc = commandData.Application.ActiveUIDocument.Document;

        // Creates a Revit task dialog to communicate information to the user.
        TaskDialog mainDialog = new TaskDialog("Hello, Revit!");
        mainDialog.MainInstruction = "Hello, Revit!";
        mainDialog.MainContent =
            "This sample shows how to use a Revit task dialog to communicate with the user."
            + "The command links below open additional task dialogs with more information.";

        // Add commmandLink options to task dialog
        mainDialog.AddCommandLink(TaskDialogCommandLinkId.CommandLink1,
                                  "View information about the Revit installation");
        mainDialog.AddCommandLink(TaskDialogCommandLinkId.CommandLink2,
                                  "View information about the active document");

        // Set common buttons and default button. If no CommonButton or CommandLink is added,
        // task dialog will show a Close button by default
        mainDialog.CommonButtons = TaskDialogCommonButtons.Close;
        mainDialog.DefaultButton = TaskDialogResult.Close;

        // Set footer text. Footer text is usually used to link to the help document.
        mainDialog.FooterText =
          "<a href=\"http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=2484975 \">"
            + "Click here for the Revit API Developer Center</a>";

        TaskDialogResult tResult = mainDialog.Show();

        // If the user clicks the first command link, a simple Task Dialog
        // with only a Close button shows information about the Revit installation.
        if (TaskDialogResult.CommandLink1 == tResult)
        {
            TaskDialog dialog_CommandLink1 = new TaskDialog("Revit Build Information");
            dialog_CommandLink1.MainInstruction =
                "Revit Version Name is: " + app.VersionName + "\n"
                + "Revit Version Number is: " + app.VersionNumber + "\n"
                + "Revit Version Build is: " + app.VersionBuild;

            dialog_CommandLink1.Show();
        }

        // If the user clicks the second command link, a simple Task Dialog
        // created by static method shows information about the active document
        else if (TaskDialogResult.CommandLink2 == tResult)
        {
            TaskDialog.Show("Active Document Inforamtion",
```

```
                "Active document: " + activeDoc.Title + "\n"
                + "Active view name: " + activeDoc.ActiveView.Name);
        }

        return Autodesk.Revit.UI.Result.Succeeded;
    }
}
```

4 Application and Document

The top level objects in the Revit Platform API are application and document. These are represented by the classes Application, UIApplication, Document and UIDocument.

- The application object refers to an individual Revit session, providing access to documents, options, and other application-wide data and settings.
  - Autodesk.Revit.UI.UIApplication - provides access to UI-level interfaces for the application, including the ability to add RibbonPanels to the user interface, and the ability to obtain the active document in the user interface.
  - Autodesk.Revit.ApplicationServices.Application - provides access to all other application level properties.
- The document object is a single Revit project file representing a building model. Revit can have multiple projects open and multiple views for one project.
  - Autodesk.Revit.UI.UIDocument - provides access to UI-level interfaces for the document, such as the contents of the selection and the ability to prompt the user to make selections and pick points
  - Autodesk.Revit.DB.Document - provides access to all other document level properties
- If multiple documents are open, the active document is the one whose view is active in the Revit session.

This chapter identifies all Application and Document functions, and then focuses on file management, settings, and units. For more details about the Element class, refer to the Elements Essentials and Editing Elements chapters and refer to the Views chapter for more details about the view elements.

4.1 Application Functions

Application functions provide access to documents, objects, and other application data and settings. All Application and UIApplication functions are identified and defined in the following sections.

4.1.1 Application

The class represents the Autodesk Revit Application, providing access to documents, options and other application wide data and settings.

4.1.1.1 Application Version Information

Application properties include VersionBuild, VersionNumber and VersionName.

4.1.1.2 Application-wide Settings

The SharedParametersFilename and LibraryPaths properties provide access to these application-wide settings.

4.1.1.3 Document Management

The Application class provides methods to create the following types of documents:

- Family document
- Project document
- Project template

The OpenDocumentFile() method can be used to open any of these document types.

All open documents can be retrieved using the Documents property.

For more details, see the Document and Management section in this chapter.

4.1.1.4 Shared Parameter Management

Revit uses one shared parameter file at a time. The Application.OpenSharedParameterFile() method accesses the shared parameter file whose path is set in the SharedParametersFilename property. For more details, see the Shared Parameter chapter.

4.1.1.5 Events

The Application class exposes document and application events such as document open and save. Subscribing to these events notifies the application when the events are enabled and acts accordingly. For more details, see the Access to Revit Event section in the Add-In Integration chapter.

4.1.1.6 Create

The Create property returns an Object Factory used to create application-wide utility and geometric objects in the Revit Platform API. Create is used when you want to create an object in the Revit application memory rather than your application's memory.

4.1.1.7 Failure Posting and Handling

The FailureDefinitionRegistry, which contains all registered FailureDefinitions is available from the static GetFailureDefinitionRegistry() method. The static method RegisterFailuresProcessor() can be used to register a custom IFailuresProcessor. For more information on posting and handling failures, see the Failure Posting and Handling chapter.

4.1.2 UIApplication

This class represents an active session of the Autodesk Revit user interface, providing access to UI customization methods, events, and the active document.

4.1.2.1 Document Management

The UIApplication provides access to the active document using the UIActiveDocument property.

4.1.2.2 Add-in Management

The ActiveAddInId property gets the current active external application or external command id, while the LoadedApplications property returns an array of successfully loaded external applications.

4.1.2.3 Ribbon Panel Utility

Use the UIApplication object to add new ribbon panels and controls to Revit.

For more details, see the Ribbon Panel and Controls section in the Add-In Integration chapter.

4.1.2.4 Extents

The DrawingAreaExtents property returns a rectangle that represents the screen pixel coordinates of drawing area, while the MainWindowExtents property returns the rectangle that represents the screen pixel coordinates of the Revit main window

4.1.2.5 Events

The UIApplication class exposes UI related events such as when a dialog box is displayed. Subscribing to these events notifies the application when the events are enabled and acts accordingly. For more details, see the Access to Revit Event section in the Add-In Integration chapter.

4.2 Document Functions

Document stores the Revit Elements, manages the data, and updates multiple data views. The Document class mainly provides the following functions.

4.2.1 Document

The Document class represents an open Autodesk Revit project.

4.2.1.1 Settings Property

The Settings property returns an object that provides access to general components within Revit projects. For more details, see the Settings section in this chapter.

4.2.1.2 Place and Locations

Each project has only one site location that identifies the physical project location on Earth. There can be several project locations for one project. Each location is an offset, or rotation, of the site location. For more details, see the Place and Locations chapter.

4.2.1.3 Type Collections

Document provides properties such as FloorTypes, WallTypes, and so on. All properties return a collection object containing the corresponding types loaded into the project.

4.2.1.4 View Management

A project document can have multiple views. The ActiveView property returns a View object representing the active view. You can filter the elements in the project to retrieve other views. For more details, see the Views chapter.

4.2.1.5 Element Retrieval

The Document object stores elements in the project. Retrieve specific elements by ElementId or UniqueId using the Element property.

For more details, see the Elements Essentials chapter.

4.2.1.6 File Management

Each Document object represents a Revit project file. Document provides the following functions:

- Retrieve file information such as file path name and project title.
- Provides Close() and Save() methods to close and save the document.

For more details, see the Document and File management section in this chapter.

4.2.1.7 Element Management

Revit maintains all Element objects in a project. To create new elements, use the Create property which returns an Object Factory used to create new project element instances in the Revit Platform API, such as FamilyInstance or Group.

The Document class can also be used to delete elements. Use the Delete() method to delete an element in the project. Deleted elements and any dependent elements are not displayed and are removed from the Document. References to deleted elements are invalid and cause an exception. For more details, see the Editing Element chapter.

Elements can also be modified using common operations such as:

- Move
- Rotate
- Mirror

- Array.

For more details, see the Editing Element chapter.

4.2.1.8 Events

Events are raised on certain actions, such as when you save a project using Save or Save As. To capture the events and respond in the application, you must register the event handlers. For more details, see the Events chapter.

4.2.1.9 Document Status

Several properties provide information on the status of the document:

- IsModifiable – whether the document may be modified
- IsModified – whether the document was changed since it was opened or saved
- IsReadOnly – whether the document is currently read only or can be modified
- IsReadOnlyFile – whether the document was opened in read-only mode
- IsFamilyDocument – whether the document is a family document
- IsWorkshared – whether worksets have been enabled in the document

4.2.1.10 Others

Document also provides other functions:

- ParameterBindings Property - Mapping between parameter definitions and categories. For more details, see the Shared Parameter chapter.
- ReactionsAreUpToDate Property - Reports whether the reactionary loads changed. For more details, see the Loads section in the Revit Structure chapter.

4.2.2 UIDocument

The UIDocument class represents an Autodesk Revit project opened in the Revit user interface.

4.2.2.1 Element Retrieval

Retrieve selected elements using the Selection property in UIDocument. This property returns an object representing the active selection containing the selected project elements. It also provides UI interaction methods to pick objects in the Revit model.

For more details, see the Elements Essentials chapter.

4.2.2.2 Element Display

The ShowElements() method uses zoom to fit to focus in on one more elements.

4.2.2.3 View Management

The UIDocument class can be used to refresh the active view in the active document.

4.3 Document and File Management

Document and file management make it easy to create and find your documents.

4.3.1 Document Retrieval

The Application class maintains all documents. As previously mentioned, you can open more than one document in a session. The active document is retrieved using the UIApplication class property, ActiveUIDocument.

All open documents, including the active document, are retrieved using the Application class Documents property. The property returns a set containing all open documents in the Revit session.

4.3.2 Document File Information

The Document class provides two properties for each corresponding file, PathName, and Title.

- PathName returns the document's fully qualified file path. The PathName returns an empty string if the project has not been saved since it was created.
- Title is the project title, which is usually derived from the project filename. The returned value varies based on your system settings.

4.3.3 Open a Document

The Application class provides a method to open an existing project file:

Method	Event
`Document OpenDocumentFile(string filename )`	`DocumentOpened`

Table 3: Open Document in API

When you specify a string with a fully qualified file path, Revit opens the file and creates a Document instance. Use this method to open a file on other computers by assigning the files Universal Naming Conversion (UNC) name to this method.

The file can be a project file with the extension .rvt, a family file with the extension .rfa, or a template file with the extension .rte. The method throws Autodesk.Revit.Exceptions.InvalidOperationException if you try to open unsupported file types. If the document is opened successfully, the DocumentOpened event is raised.

4.3.4 Create a Document

Create new documents using the Application methods in the following table.

Method	Event
`Document NewProjectDocument(string templateFileName);`	`DocumentCreated`
`Document NewFamilyDocument(string templateFileName);`	`DocumentCreated`
`Document NewProjectTemplateDocument(string templateFilename);`	`DocumentCreated`

Table 4: Create Document in the API

Each method requires a template file name as the parameter. The created document is returned based on the template file.

4.3.5 Save and Close a Document

The Document class provides methods to save or close instances.

Method	Event
`Save()`	`DocumentSaved`
`SaveAs()`	`DocumentSavedAs`
`Close()`	`DocumentClosed`

Table 5: Save and Close Document in API

SaveAs() has 3 overloads. Two overloads take the filename as an argument and do not require input from the user. The third overload takes no arguments and the user will be prompted for a file name with the SaveAs dialog.

Close() has two overloads. One takes a Boolean argument that indicates whether to save the file before closing it. The second overload takes no arguments and if the document was modified, the user will be asked if they want to save the file before closing.

Note: The Close() method does not affect the active document or raise the DocumentClosed event, because the document is used by an external application. You can only call this method on non-active documents.

4.3.6 **Load Family**

The Document class provides you with the ability to load an entire family and all of its symbols into the project. Because loading an entire family can take a long time and a lot of memory, the Document class provides a similar method, LoadFamilySymbol() to load only specified symbols.

For more details, see Loading Families.

4.4 Settings

The following table identifies the commands in the Revit Platform UI Manage tab, and corresponding APIs.

UI command	Associated API	Reference
Project Information	Document.ProjectInformation	See the following note
Project Parameters	Document.ParameterBindings (Only for Shared Parameter)	See Shared Parameter
Project Location panel	Document.ProjectLocations Document.ActiveProjectLocation	See Place and Locations
Settings > Fill Patterns	Document.Settings.FillPatterns	See the following note
Materials	Document.Settings.Materials	See Materials Management
Settings > Object Styles	Document.Settings.Categories	See the following note
Phases...	Document.Phases	See the following note
Structural Settings	Load related structural settings are available in the API	See Revit Structure
Project Units	Document.ProjectUnit	See Unit
Area and Volume Calculations (on the Room & Area panel)	Document.Settings.VoumeCalculationSetting	See the following note

Table 6: Settings in API and UI

Note:

- Project Information - The API provides the ProjectInfo class which is retrieved using Document.ProjectInformation to represent project information in the Revit project. The following table identifies the corresponding APIs for the Project Information parameters.

Parameters	Corresponding API	Built-in Parameters
Project Issue Date	ProjectInfo.IssueDate	PROJECT_ISSUE_DATE
Project Status	ProjectInfo.Status	PROJECT_STATUS

Parameters	Corresponding API	Built-in Parameters
Client Name	ProjectInfo.ClientName	CLIENT_NAME
Project Address	ProjectInfo.Address	PROJECT_ADDRESS
Project Name	ProjectInfo.Name	PROJECT_NAME
Project Number	ProjectInfo.Number	PROJECT_NUMBER

Table 7: ProjectInformation

Use the properties exposed by ProjectInfo to retrieve and set all strings. These properties are implemented by the corresponding built-in parameters. You can get or set the values through built-in parameters directly. For more information about how to gain access to these parameters through the built-in parameters, see the Parameter section in the Elements Essentials chapter. The recommended way to get project information is to use the ProjectInfo properties.

- Fill Patterns - Retrieve all Fill Patterns in the current document using Document.Settings.FillPatterns.
- Object Styles - Use Settings.Categories to retrieve all information in Category objects except for Line Style. For more details, see the Categories section in the Elements Essentials chapter and Material chapter.
- Phases - Revit maintains the element lifetime by phases, which are distinct time periods in the project lifecycle. All phases in a document are retrieved using the Document.Phases property. The property returns an array containing Phase class instances. However, the Revit API does not expose functions from the Phase class.
- Options - The Options command configures project global settings. You can retrieve an Options.Application instance using the Application.Options property. Currently, the Options.Application class only supports access to library paths and shared parameters file.
- Area and Volume Calculations – The Document.Settings.VolumeCalculationSetting allows you to enable or disable volume calculations, and to change the room boundary location.

4.5 Units

The Revit Unit System is based on seven base units for seven base quantities that are independent. The base units are identified in the following table.

Base Unit	Unit In Revit	Unit System
Length	Feet (ft)	Imperial
Angle	Radian	Metric
Mass	Kilogram (kg)	Metric
Time	Seconds (s)	Metric
Electric Current	Ampere (A)	Metric
Temperature	Kelvin (K)	Metric
Luminous Intensity	Candela (cd)	Metric

Table 8: 7 Base Units in Revit Unit System

Note: Since Revit stores lengths in feet and other basic quantities in metric units, a derived unit involving length will be a non-standard unit based on both the Imperial and the Metric systems. For example, since a force is measured in "mass-length per time squared", it is stored in kg-ft / s^2.

The Revit Platform API provides access to project units and format via the Document.ProjectUnit. The APIs that provide access to Project units and format settings include the following:

UI Command	Corresponding Type in API	Access API
Slope	Enums.RiseRunOrAngleType	ProjectUnit.Slope
Decimal symbol	Enums.DecimalSymbolType	ProjectUnit.DecimalSymbolType
Discipline	Enums.UnitDiscipline	ProjectUnit.FormatOptions.Discipline
Units	Enums.DisplayUnitType	ProjectUnit.FormatOptions.Units
Rounding	System.Double	ProjectUnit.FormatOptions.Rounding
Unit suffix	Enums.UnitSuffixType	ProjectUnit.FormatOptions.Unitsuffix

Table 9: Project Unit Properties

5 Elements Essentials

An Element corresponds to a single building or drawing component, such as a door, a wall, or a dimension. In addition, an Element can be a door type, a view, or a material definition.

5.1 Element Classification

Revit Elements are divided into six groups: Model, Sketch, View, Group, Annotation and Information. Each group contains related Elements and their corresponding symbols.

5.1.1 Model Elements

Model Elements represent physical items that exist in a building project. Elements in the Model Elements group can be subdivided into the following:

- Family Instances - Family Instances contain family instance objects. You can load family objects into your project or create them from family templates. For more information, see the Family Instances chapter.
- Host Elements - Host Elements contain system family objects that can contain other model elements, such as wall, roof, ceiling, and floor. For more information about Host Elements, see the Walls, Floors, Roofs and Openings chapter.
- Structure Elements. - Structure Elements contain elements that are only used in Revit Structure. For more information about Structure Elements, see the Revit Structure chapter.

5.1.2 View Elements

View Elements represent the way you view and interact with other objects in Revit. For more information, see the Views chapter.

5.1.3 Group Elements

Group Elements represent the assistant Elements such as Array and Group objects in Revit. For more information, see the Editing Elements chapter.

5.1.4 Annotation and Datum Elements

Annotation and Datum Elements contain non-physical items that are visible.

- Annotation Elements represent 2D components that maintain scale on paper and are only visible in one view. For more information about Annotation Elements, see the Annotation Elements chapter.

Note: Annotation Elements representing 2D components do not exist only in 2D views. For example, dimensions can be drawn in 3D view while the shape they refer to only exists in a 2D planar face.

- Datum Elements represent non-physical items that are used to establish project context. These elements can exist in views. Datum Elements are further divided into the following:
- Common Datum Elements - Common Datum Elements represent non-physical visible items used to store data for modeling.
- Datum FamilyInstance - Datum FamilyInstance represents non-physical visible items loaded into your project or created from family templates.
- **Note:** For more information about Common Datum Elements and Datum FamilyInstance, see the Datum and Information Elements chapter; for ModelCurve related contents, see the Sketching chapter.

- Structural Datum Elements - Structural Datum Elements represent non-physical visible items used to store data for structure modeling. For more information about Structural Datum Elements, see the Revit Structure chapter.

5.1.5 Sketch Elements

Sketch Elements represent temporary items used to sketch 2D/3D form. This group contains the following objects used in family modeling and massing:

- SketchPlane
- Sketch
- Path3D
- GenericForm.

For Sketch details, see the Sketching chapter.

5.1.6 Information Elements

Information Elements contain non-physical invisible items used to store project and application data. Information Elements are further separated into the following:

- Project Datum Elements
- Project Datum Elements (Unique).

For more information about Datum Elements, see the Datum and Information Elements chapter.

5.2 Other Classifications

Elements are also classified by the following:

- Category
- Family
- Symbol
- Instance

There are some relationships between the classifications. For example:

- You can distinguish different kinds of FamilyInstances by the category. Items such as structural columns are in the Structural Columns category, beams and braces are in the Structural Framing category, and so on.
- You can differentiate structural FamilyInstance Elements by their symbol.

5.2.1 Category

The Element.Category property represents the category or subcategory to which an Element belongs. It is used to identify the Element type. For example, anything in the walls Category is considered a wall. Other categories include doors and rooms.

Categories are the most general class. The Document.Settings.Categories property is a map that contains all Category objects in the document and is subdivided into the following:

- Model Categories - Model Categories include beams, columns, doors, windows, and walls.
- Annotation Categories. Annotation Categories include dimensions, grids, levels, and textnotes.

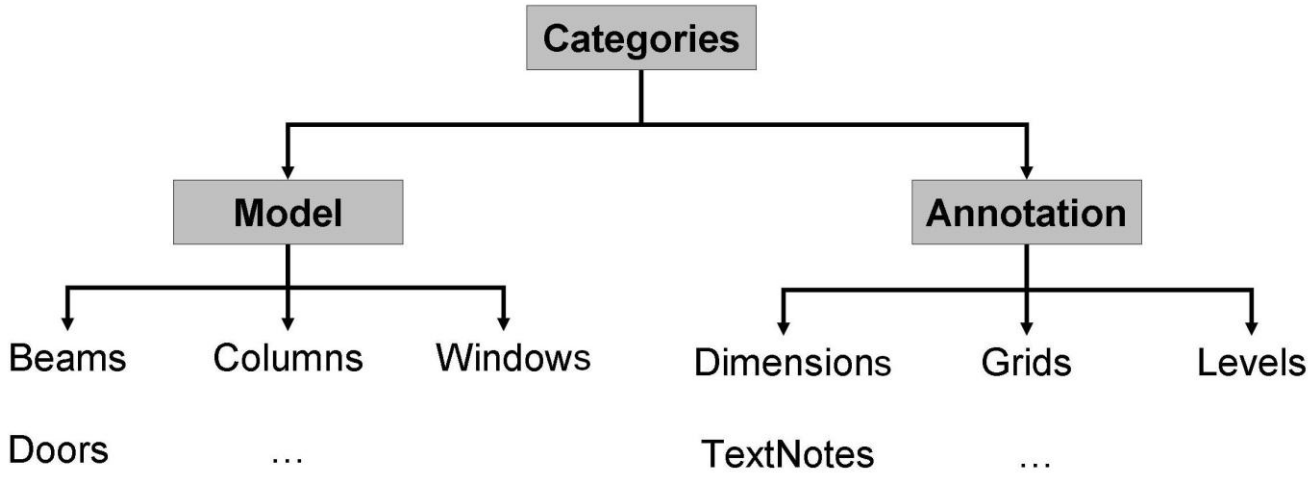


Figure 19: Categories

Note: The following guidelines apply to categories:

- In general, the following rules apply to categories:
  - Each family object belongs to a category
  - Non-family objects, like materials and views, do not belong to a category
  - There are exceptions such as ProjectInfo, which belongs to the Project Information category.
- An element and its corresponding symbols are usually in the same category. For example, a basic wall and its wall type Generic – 8" are all in the Walls category.
- The same type of Elements can be in different categories. For example, SpotDimensions has the SpotDimensionType, but it can belong to two different categories: Spot Elevations and Spot Coordinates.
- Different Elements can be in the same category because of their similarity or for architectural reasons. ModelLine and DetailLine are in the Lines category.

To gain access to the categories in a document's Setting class (for example, to insert a new category set), use one of the following techniques:

- Get the Categories from the document properties.
- Get a specific category quickly from the categories map using the BuiltInCategory enumerated type.

Code Region 5-1: Getting categories from document settings

```
// Get settings of current document
Settings documentSettings = document.Settings;

// Get all categories of current document
Categories groups = documentSettings.Categories;

// Show the number of all the categories to the user
String prompt = "Number of all categories in current Revit document:" + groups.Size;

// get Floor category according to OST_Floors and show its name
```

```
Category floorCategory = groups.get_Item(BuiltInCategory.OST_Floors);
prompt += floorCategory.Name;

// Give the user some information
MessageBox.Show(prompt, "Revit", MessageBoxButtons.OK);
```

Category is used in the following manner:

- Category is used to classify elements. The element category determines certain behaviors. For example, all elements in the same category can be included in the same schedule.
- Elements have parameters based on their categories.
- Categories are also used for controlling visibility and graphical appearance in Revit.

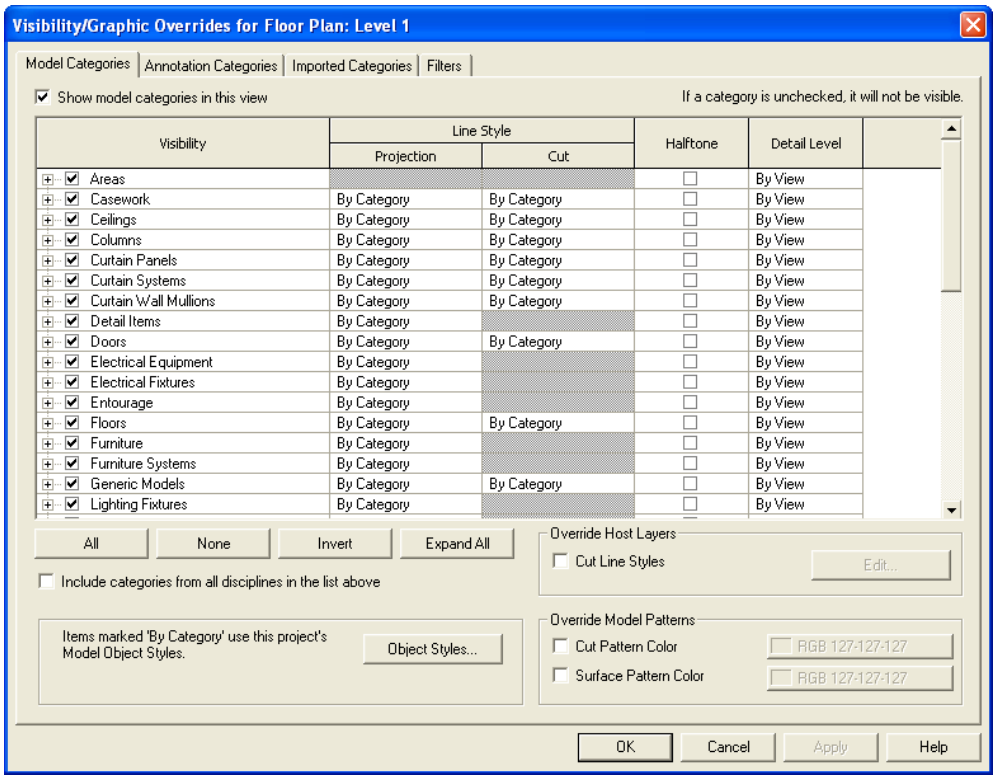


Figure 20: Visibility by Category

An element's category is determined by the Category ID.

- Category IDs are represented by the ElementId class.
- Imported Category IDs correspond to elements in the document.
- Most categories are built-in and their IDs are constants stored in ElementIds.
- Each built-in category ID has a corresponding value in the BuiltInCategory Enumeration. They can be converted to corresponding BuiltInCategory enumerated types.
- If the category is not built-in, the ID is converted to a null value.

Code Region 5-2: Getting element category

```
Element selectedElement = null;
```

```
foreach (Element e in document.Selection.Elements)
{
    selectedElement = e;
    break;  // just get one selected element
}

// Get the category instance from the Category property
Category category = selectedElement.Category;

BuiltInCategory enumCategory = (BuiltInCategory)category.Id.Value;
```

Note: To avoid Globalization problems when using Category.Name, BuiltInCategory is a better choice. Category.Name can be different in different languages.

5.2.2 Family

Families are classes of Elements within a category. Families can group Elements by the following:

- A common set of parameters (properties).
- Identical use.
- Similar graphical representation.

Most families are component Family files, meaning that you can load them into your project or create them from Family templates. You determine the property set and the Family graphical representation.

Another family type is the system Family. System Families are not available for loading or creating. Revit predefines the system Family properties and graphical representation; they include walls, dimensions, roofs, floors (or slabs), and levels.

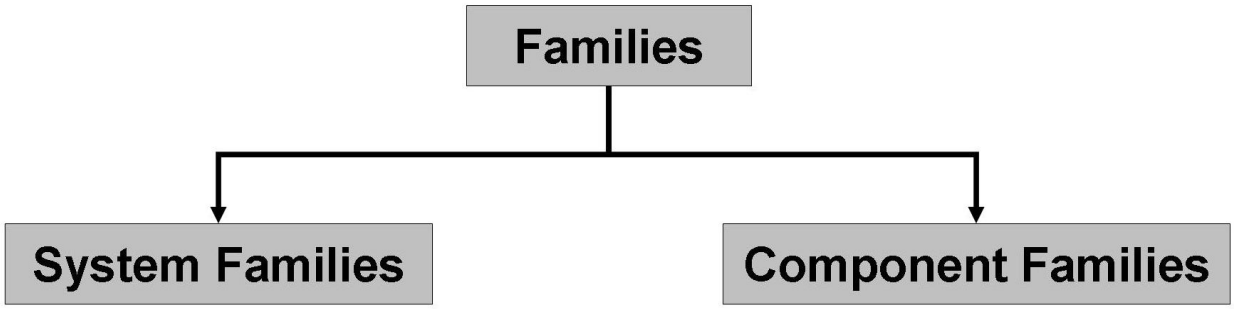


Figure 21: Families

In addition to functioning as an Element class, Family is also a template used to generate new items that belong to the Family.

5.2.2.1 Family in the Revit Platform API

In the Revit Platform API, both the Family class and FamilyInstance belong to the Component Family. Other Elements include System Family.

Families in the Revit Platform API are represented by three objects:

- Family
- FamilySymbol
- FamilyInstance.

Each object plays a significant role in the Family structure.

The Family object has the following characteristics:

- Represents an entire family such as a beam.
- Represents the entire family file on a disk.
- Contains a number of FamilySymbols.

The FamilySymbol object represents a specific set of family settings in the Family such as the Type, Concrete-Rectangular Beam: 16x32.

The FamilyInstance object is a FamilySymbol instance representing a single instance in the Revit project. For example, the FamilyInstance can be a single instance of a 16x32 Concrete-Rectangular Beam in the project.

Note: Remember that the FamilyInstance exists in FamilyInstance Elements, Datum Elements, and Annotation Elements.

Consequently, the following rules apply:

- Each FamilyInstance has one FamilySymbol.
- Each FamilySymbol belongs to one Family.
- Each Family contains one or more FamilySymbols.

For more detailed information, see the Family Instances chapter.

5.2.3 ElementType

In the Revit Platform API, Symbols are usually non-visible elements used to define instances. Symbols are called Types in the user interface.

- A type can be a specific size in a family, such as a 1730 X 2032 door, or an 8x4x1/2 angle.
- A type can be a style, such as default linear or default angular style for dimensions.

Symbols represent Elements that contain shared data for a set of similar elements. In some cases, Symbols represent building components that you can get from a warehouse, such as doors or windows, and can be placed many times in the same building. In other cases, Symbols contain host object parameters or other elements. For example, a WallType Symbol contains the thickness, number of layers, material for each layer, and other properties for a particular wall type.

FamilySymbol is a symbol in the API. It is also called Family Type in the Revit user interface. FamilySymbol is a class of elements in a family with the exact same values for all properties. For example, all 32x78 six-panel doors belong to one type, while all 24x80 six-panel doors belong to another type. Like a Family, a FamilySymbol is also a template. The FamilySymbol object is derived from the ElementType object and the Element object.

5.2.4 Instance

Instances are items with specific locations in the building (model instances) or on a drawing sheet (annotation instances). Instance represents transformed identical copies of an ElementType. For example, if a building contains 20 windows of a particular type, there is one ElementType with 20 Instances. Instances are called Components in the user interface.

Note: For FamilyInstance, the Symbol property can be used instead of the GetTypeId() method to get the corresponding FamilySymbol. It is convenient and safe since you do not need to do a type conversion.

5.3 Element Retrieval

Elements in Revit are very common. Retrieving the elements that you want from Revit is necessary before using the API for any Element command. There are several ways to retrieve elements with the Revit API:

- ElementId - If the ElementId of the element is known, the element can be retrieved from the document.
- Element filtering and iteration - this is a good way to retrieve a set of related elements in the document.
- Selected elements – retrieves the set of elements that the user has selected
- Specific elements – some elements are available as properties of the document

Each of these methods of element retrieval is discussed in more details in the following sections.

5.3.1 Getting an Element by ID

When the ElementId of the desired element is known, use the Document.Element property to get the element.

5.3.2 Filtering the Elements Collection

The most common way to get elements in the document is to use filtering to retrieve a collection of elements. The Revit API provides the FilteredElementCollector class, and supporting classes, to create filtered collections of element which can then be iterated. See the chapter on Filtering for more information.

5.3.3 Selection

Rather than getting a filtered collection of all of the elements in the model, you can access just the elements that have been selected. You can get the selected objects from the current active document using the UIDocument.Selection.Elements property. For more information on using the active selection, see the Selection chapter.

5.3.4 Accessing Specific Elements from Document

In addition to using the general way to access Elements, the Revit Platform API has properties in the Document class to get the specified Elements from the current active document without iterating all Elements. The specified Elements you can retrieve are listed in the following table.

Element	Access in property of Document
ProjectInfo	Document.ProjectInformation
ProjectUnit	Document.ProjectUnit
ProjectLocation	Document.ProjectLocations Document.ActiveProjectLocation
SiteLocation	Document.SiteLocation
Phase	Document.Phases
FillPattern	Document.Settings.FillPatterns
Material	Document.Settings.Materials
FamilySymbol relative to the deck profile of the layer within a structure	Document.DeckProfiles
FamilySymbol in the Title Block category	Document.TitleBlocks

Element	Access in property of Document
All Element types	Document.AnnotationSymbolTypes / BeamSystemTypes / ContFootingTypes / DimensionTypes / FloorTypes / GridTypes / LevelTypes / RebarBarTypes / RebarHookTypes / RoomTagTypes / SpotDimensionTypes / WallTypes / TextNoteTypes

Table 10: Retrieving Elements from document properties

5.4 General Properties

The following properties are common to each Element created using Revit.

5.4.1 ElementId

Every element in an active document has a unique identifier represented by the ElementId storage type. ElementId objects are project wide. It is a unique number that is never changed in the element model, which allows it to be stored externally to retrieve the element when needed.

To view an element ID in Revit, complete the following steps:

1. From the Modify tab, on the Inquiry panel, select Element ID. The Element ID drop down menu appears.

Select IDs of Selection to get the ID number for one element.

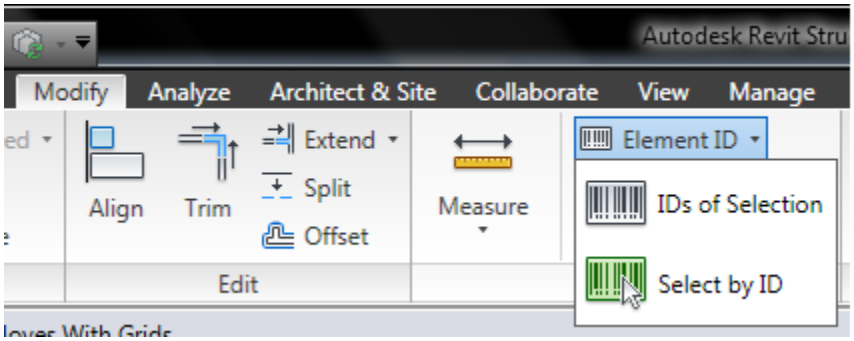


Figure 22: ElementId

In the Revit Platform API, you can create an ElementId directly, and then associate a unique integer value to the new ElementId. The new ElementId value is 0 by default.

Code Region 5-3: Setting ElementId

```
// Get the id of the element
Autodesk.Revit.DB.ElementId selectedId = element.Id;
int idInteger = selectedId.IntegerValue;

// create a new id and set the value
Autodesk.Revit.DB.ElementId id = new Autodesk.Revit.DB.ElementId(idInteger);
```

ElementId has the following uses:

- Use ElementId to retrieve a specific element from Revit. From the Revit Application class, gain access to the active document, and then get the specified element using the Document.Element(ElementId) property.

Code Region 5-4: Using ElementId

```
// Get the id of the element
Autodesk.Revit.DB.ElementId selectedId = element.Id;
int idInteger = selectedId.IntegerValue;

// create a new id and set the value
Autodesk.Revit.DB.ElementId id = new Autodesk.Revit.DB.ElementId(idInteger);

// Get the element
Autodesk.Revit.DB.Element first = document.get_Element(id);
```

If the ID number does not exist in the project, the element you retrieve is null.

- Use ElementId to check whether two Elements in one project are equal or not. It is not recommended to use the Object.Equal() method.

5.4.2 UniqueId

Every element has a UniqueId, represented by the String storage type. The UniqueId corresponds to the ElementId. However, unlike ElementId, UniqueId functions like a GUID (Globally Unique Identifier), which is unique across separate Revit projects. UniqueId can help you to track elements when you export Revit project files to other formats.

Code Region 5-5: UniqueId

```
String uniqueId = element.UniqueId;
```

Note: The ElementId is only unique in the current project. It is not unique across separate Revit projects. UniqueId is always unique across separate projects.

5.4.3 Location

The location of an object is important in the building modeling process. In Revit, some objects have a point location. For example a table has a point location. Other objects have a line location, representing a location curve or no location at all. A wall is an element that has a line location.

The Revit Platform API provides the Location class and location functionality for most elements. For example, it has the Move() and Rotate() methods to translate and rotate the elements. However, the Location class has no property from which you can get information such as a coordinate. In this situation, downcast the Location object to its subclass—like LocationPoint or LocationCurve—for more detailed location information and control using object derivatives.

Retrieving an element's physical location in a project is useful when you get the geometry of an object. The following rules apply when you retrieve a location:

- Wall, Beam, and Brace are curve-driven using LocationCurve.
- Room, RoomTag, SpotDimension, Group, FamilyInstances that are not curve-driven, and all In-Place-FamilyInstances use LocationPoint.

In the Revit Platform API, curve-driven means that the geometry or location of an element is determined by one or more associated curves. Almost all analytical model elements are curve-driven – linear and area loads, walls, framing elements, and so on.

Other Elements cannot retrieve a LocationCurve or LocationPoint. They return Location with no information.

Location Information	Elements
LocationCurve	Wall, Beam, Brace, Structural Truss, LineLoad(without host)

Location Information	Elements
LocationPoint	Room, RoomTag, SpotDimension, Group, Column, Mass
Only Location	Level, Floor, some Tags, BeamSystem, Rebar, Reinforcement, PointLoad, AreaLoad(without Host), Span Direction(IndependentTag)
No Location	View, LineLoad(with host), AreaLoad(with Host), BoundaryCondition

Table 11: Elements Location Information

Note: There are other Elements without Location information. For example a LineLoad (with host) or an AreaLoad (with host) have no Location.

Some FamilyInstance LocationPoints, such as all in-place-FamilyInstances and masses, are specified to point (0, 0, 0) when they are created. The LocationPoint coordinate is changed if you transform or move the instance.

To change a Group's LocationPoint, do one of the following:

- Drag the Group origin in the Revit UI to change the LocationPoint coordinate. In this situation, the Group LocationPoint is changed while the Group's location is not changed.
- Move the Group using the Document.Move() method to change the LocationPoint. This changes both the Group location and the LocationPoint.

For more information about LocationCurve and LocationPoint, see the Editing Elements chapter Move section.

5.4.4 Level

Levels are finite horizontal planes that act as a reference for level-hosted or level-based elements, such as roofs, floors, and ceilings. The Revit Platform API provides a Level class to represent level lines in Revit. Get the Level object to which the element is assigned using the API if the element is level-based.

Code Region 5-6: Assigning Level

```
// Get the level object to which the element is assigned.
Level level = element.Level;
```

A number of elements, such as a column, use a level as a basic reference. When you get the column level, the level you retrieve is the Base Level.

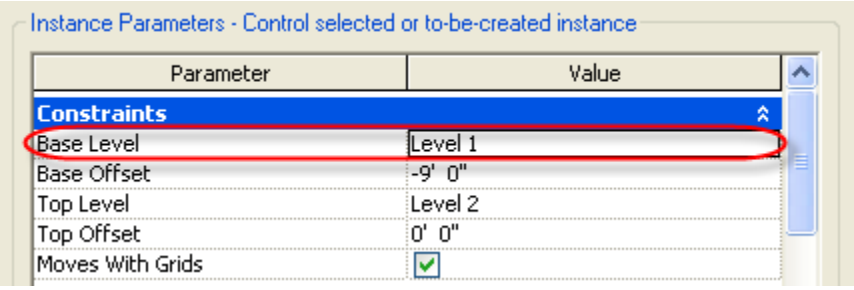


Figure 23: Column Base Level parameter

Note: Get the Beam or Brace level using the Reference Level parameter. From the Level property, you only get null instead of the reference level information.

Level is the most commonly used element in Revit. In the Revit Platform API, all levels in the project are located by iterating over the entire project and searching for Elements.Level objects.

For more Level details, see the Datum and Information Elements chapter.

5.4.5 **Parameter**

Every element has a set of parameters that users can view and edit in Revit. The parameters are visible in the Element Properties dialog box (select any element and click the Properties button next to the type selector). For example, the following image shows Room parameters.

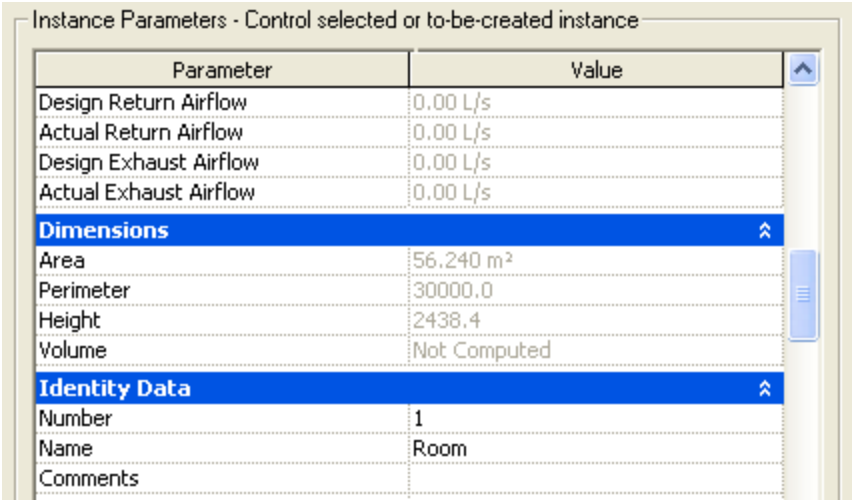


Figure 24: Room parameters

In the Revit Platform API, each Element object has a Parameters property, which is a collection of all the properties attached to the Element. You can change the property values in the collection. For example, you can get the area of a room from the room object parameters; additionally, you can set the room number using the room object parameters. The Parameter is another way to provide access to property information not exposed in the element object.

In general, every element parameter has an associated parameter ID. Parameter IDs are represented by the ElementId class. For user-created parameters, the IDs correspond to real elements in the document. However, most parameters are built-in and their IDs are constants stored in ElementIds.

Parameter is a generic form of data storage in elements. In the Revit Platform API, it is best to use the built-in parameter ID to get the parameter. Revit has a large number of built-in parameters available using the BuiltInParameter enumerated type.

For more details, see the Parameter Chapter.

6 Filtering

The Revit API provides a mechanism for filtering and iterating elements in a Revit document. This is the best way to get a set of related elements, such as all walls or doors in the document. Filters can also be used to find a very specific set of elements, such as all beams of a specific size.

The basic steps to get elements passing a specified filter are as follows:

1. Create a new FilteredElementCollector
2. Apply one or more filters to it
3. Get filtered elements or element ids (using one of several methods)

The following sample covers the basic steps to filtering and iterating elements in the document.

Code Region 6-1: Use element filtering to get all wall instances in document

```
// Find all Wall instances in the document by using category filter
ElementCategoryFilter filter = new ElementCategoryFilter(BuiltInCategory.OST_Walls);

// Apply the filter to the elements in the active document,
// Use shortcut WhereElementIsNotElementType() to find wall instances only
FilteredElementCollector collector = new FilteredElementCollector(document);
IList<Element> walls =
collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();
String prompt = "The walls in the current document are:\n";
foreach (Element e in walls)
{
    prompt += e.Name + "\n";
}
TaskDialog.Show("Revit", prompt);
```

6.1 Create a FilteredElementCollector

The main class used for element iteration and filtering is called FilteredElementCollector. It is constructed in one of three ways:

1. From a document - will search and filter the set of elements in a document
2. From a document and set of ElementIds - will search and filter a specified set of elements
3. From a document and a view - will search and filter the visible elements in a view

Note: Always check that a view is valid for element iteration when filtering elements in a specified view by using the static FilteredElementCollector.IsViewValidForElementIteration().

When the object is first created, there are no filters applied. This class requires that at least one condition be set before making at attempt to access the elements, otherwise an exception will be thrown.

6.2 Applying Filters

Filters can be applied to a FilteredElementCollector using ElementFilters. An ElementFilter is a class that examines an element to see if it meets a certain criteria. The ElementFilter base class has three derived classes that divide element filters into three categories.

- **ElementQuickFilter** - Quick filters operate only on the ElementRecord, a low-memory class which has a limited interface to read element properties. Elements which are rejected by a quick filter will not be expanded in memory.
- **ElementSlowFilter** - Slow filters require that the Element be obtained and expanded in memory first. Thus it is preferable to couple slow filters with at least one ElementQuickFilter, which should minimize the number of Elements that are expanded in order to evaluate against the criteria set by this filter.
- **ElementLogicalFilter** – Logical filters combine two or more filters logically. The component filters may be reordered by Revit to cause the quickest acting filters to be evaluated first.

Most filters may be inverted using an overload constructor that takes a Boolean argument indicating to invert the filter so that elements that would normally be accepted by the filter will be rejected, and elements that would normally be rejected will be accepted. Filters that cannot be inverted are noted in their corresponding sections below.

There is a set of predefined filters available for common uses. Many of these built-in filters provide the basis for the FilteredElementCollector shortcut methods mentioned in the FilteredElementCollector section above. The next three sections provide more information on the built-in filters.

Once a filter is created, it needs to be applied to the FilteredElementCollector. The generic method WherePasses() is used to apply a single ElementFilter to the FilteredElementCollector.

Filters can also be applied using a number of shortcut methods provided by FilteredElementCollector. Some apply a specific filter without further input, such as WhereElementIsCurveDriven(), while others apply a specific filter with a simple piece of input, such as the OfCategory() method which takes a BuiltInCategory as a parameter. And lastly there are methods such as UnionWith() that join filters together. All of these methods return the same collector allowing filters to be easily chained together.

6.2.1 Quick filters

Quick filters operate only on the ElementRecord, a low-memory class which has a limited interface to read element properties. Elements which are rejected by a quick filter will not be expanded in memory. The following table summarizes the built-in quick filters, and some examples follow for a few of the filters.

Built-in Filter	What it passes	Shortcut Method(s)
BoundingBoxContainsPointFilter	Elements which have a bounding box that contains a given point	None
BoundingBoxIntersectsFilter	Elements which have a bounding box which intersects a given outline	None
BoundingBoxIsInsideFilter	Elements which have a bounding box inside a given outline	None
ElementCategoryFilter	Elements matching the input category id	OfCategoryId()
ElementClassFilter	Elements matching the input runtime class (or derived classes)	OfClass()
ElementDesignOptionFilter	Elements in a particular design option	ContainedInDesignOption()

Built-in Filter	What it passes	Shortcut Method(s)
ElementIsCurveDrivenFilter	Elements which are curve driven	WhereElementIsCurveDriven()
ElementIsElementTypeFilter	Elements which are "Element types"	WhereElementIsElementType() WhereElementIsNotElementType()
ElementOwnerViewFilter	Elements which are view-specific	OwnedByView() WhereElementIsViewIndependent()
ElementStructuralTypeFilter	Elements matching a given structural type	None
ExclusionFilter	All elements except the element ids input to the filter	Excluding()
FamilySymbolFilter	Symbols of a particular family	None

Table 12: Built-in Quick Filters

Note: The FamilySymbolFilter cannot be inverted.

Note: The bounding box filters exclude all objects derived from View and objects derived from ElementType.

The following example creates an outline in the document and then uses a BoundingBoxIntersectsFilter to find the elements in the document with a bounding box that intersects that outline. It then shows how to use an inverted filter to find all walls whose bounding box do not intersect the given outline. Note that the use of the OfClass() method applies an ElementClassFilter to the collection as well.

Code Region 6-2: BoundingBoxIntersectsFilter example

```
// Use BoundingBoxIntersects filter to find elements with a bounding box that intersects the
// given Outline in the document.

// Create a Outline, uses a minimum and maximum XYZ point to initialize the outline.
Outline myOutLn = new Outline(new XYZ(0, 0, 0), new XYZ(100, 100, 100));

// Create a BoundingBoxIntersects filter with this Outline
BoundingBoxIntersectsFilter filter = new BoundingBoxIntersectsFilter(myOutLn);

// Apply the filter to the elements in the active document
// This filter excludes all objects derived from View and objects derived from ElementType
FilteredElementCollector collector = new FilteredElementCollector(document);
IList<Element> elements = collector.WherePasses(filter).ToElements();

// Find all walls which don't intersect with BoundingBox: use an inverted filter
// to match elements
// Use shortcut command OfClass() to find walls only
BoundingBoxIntersectsFilter invertFilter = new BoundingBoxIntersectsFilter(myOutLn, true);
collector = new FilteredElementCollector(document);
IList<Element> notIntersectWalls =
        collector.OfClass(typeof(Wall)).WherePasses(invertFilter).ToElements();
```

The next example uses an exclusion filter to find all walls that are not currently selected in the document.

Code Region 6-3: Creating an exclusion filter

```
// Find all walls that are not currently selected,
// Get all element ids which are current selected by users, exclude these ids when filtering
ICollection<ElementId> excludes = new List<ElementId>();
ElementSetIterator elemIter = uiDocument.Selection.Elements.ForwardIterator();
elemIter.Reset();
while (elemIter.MoveNext())
{
    Element curElem = elemIter.Current as Element;
    excludes.Add(curElem.Id);
}

// Create filter to exclude all selected element ids
ExclusionFilter filter = new ExclusionFilter(excludes);

// Apply the filter to the elements in the active document,
// Use shortcut method OfClass() to find Walls only
FilteredElementCollector collector = new FilteredElementCollector(uiDocument.Document);
IList<Element> walls = collector.WherePasses(filter).OfClass(typeof(Wall)).ToElements();
```

Note that the ElementClassFilter will match elements whose class is an exact match to the input class, or elements whose class is derived from the input class. The following example uses an ElementClassFilter to get all loads in the document.

Code Region 6-4: Using an ElementClassFilter to get loads

```
// Use ElementClassFilter to find all loads in the document
// Using typeof(LoadBase) will yield all AreaLoad, LineLoad and PointLoad
ElementClassFilter filter = new ElementClassFilter(typeof(LoadBase));

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector(document);
ICollection<Element> allLoads = collector.WherePasses(filter).ToElements();
```

There is a small subset of Element subclasses in the API which are not supported by the element class filter. These types exist in the API, but not in Revit's native object model, which means that this filter doesn't support them. In order to use a class filter to find elements of these types, it is necessary to use a higher level class and then process the results further to find elements matching only the subtype. Note that dedicated filters exist for some of these types. The following types are affected by this restriction:

Type	Dedicated Filter
Subclasses of Autodesk.Revit.DB.Material	None
Subclasses of Autodesk.Revit.DB.CurveElement	CurveElementFilter
Subclasses of Autodesk.Revit.DB.ConnectorElement	None
Subclasses of Autodesk.Revit.DB.HostedSweep	None
Autodesk.Revit.DB.Architecture.Room	RoomFilter
Autodesk.Revit.DB.Mechanical.Space	SpaceFilter

Type	Dedicated Filter
Autodesk.Revit.DB.Area	AreaFilter
Autodesk.Revit.DB.Architecture.RoomTag	RoomTagFilter
Autodesk.Revit.DB.Mechanical.SpaceTag	SpaceTagFilter
Autodesk.Revit.DB.AreaTag	AreaTagFilter
Autodesk.Revit.DB.CombinableElement	None
Autodesk.Revit.DB.Mullion	None
Autodesk.Revit.DB.Panel	None
Autodesk.Revit.DB.AnnotationSymbol	None
Autodesk.Revit.DB.Structure.AreaReinforcementType	None
Autodesk.Revit.DB.Structure.PathReinforcementType	None
Autodesk.Revit.DB.AnnotationSymbolType	None
Autodesk.Revit.DB.Architecture.RoomTagType	None
Autodesk.Revit.DB.Mechanical.SpaceTagType	None
Autodesk.Revit.DB.AreaTagType	None
Autodesk.Revit.DB.Structure.TrussType	None

6.2.2 **Slow Filters**

Slow filters require that the Element be obtained and expanded in memory first. Thus it is preferable to couple slow filters with at least one ElementQuickFilter, which should minimize the number of Elements that are expanded in order to evaluate against the criteria set by this filter. The following table summarizes the built-in slow filters, while a few examples follow to provide an in-depth look at some of the filters.

Built-in Filter	What it passes	Shortcut Method(s)
AreaFilter	Areas	None
AreaTagFilter	Area tags	None
CurveElementFilter	CurveElements	None
ElementLevelFilter	Elements associated with a given level id	None
ElementParameterFilter	Elements passing one or more parameter filter rules	None
FamilyInstanceFilter	Instances of a particular family instance	None
FamilyStructuralMaterialTypeFilter	Family elements of given structural material type	None
PrimaryDesignOptionMemberFilter	Elements owned by any primary design option	None
RoomFilter	Rooms	None
RoomTagFilter	Room tags	None
SpaceFilter	Spaces	None
SpaceTagFilter	Space tags	None

Built-in Filter	What it passes	Shortcut Method(s)
StructuralInstanceUsageFilter	FamilyInstances of given structural usage	None
StructuralMaterialTypeFilter	FamilyInstances of given structural material type	None
StructuralWallUsageFilter	Walls of given structural wall usage	None

Table 13: Built-in Slow Filters

The following slow filters cannot be inverted:

- RoomFilter
- RoomTagFilter
- AreaFilter
- AreaTagFilter
- SpaceFilter
- SpaceTagFilter
- FamilyInstanceFilter

As mentioned in the quick filters section, some classes do not work with the ElementClassFilter. Some of those classes, such as Room and RoomTag have their own dedicated filters.

Code Region 6-5: Using the Room filter

```
// Use a RoomFilter to find all room elements in the document. It is necessary to use the
// RoomFilter and not an ElementClassFilter or the shortcut method OfClass() because the Room
// class is not supported by those methods.
RoomFilter filter = new RoomFilter();

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector(document);
IList<Element> rooms = collector.WherePasses(filter).ToElements();
```

The ElementParameterFilter is a powerful filter that can find elements based on values of parameters they may have. It can find elements whose parameter values match a specific value or are greater or less than some value. ElementParameterFilter can also be used to find elements that support a specific shared parameter.

The example below uses an ElementParameterFilter to find rooms whose size is more than 100 square feet and rooms with less than 100 square feet.

Code Region 6-6: Using a parameter filter

```
// Creates an ElementParameter filter to find rooms whose area is
// greater than specified value
// Create filter by provider and evaluator
BuiltInParameter areaParam = BuiltInParameter.ROOM_AREA;
// provider
ParameterValueProvider pvp = new ParameterValueProvider(new ElementId((int)areaParam));
// evaluator
```

```
FilterNumericRuleEvaluator fnrv = new FilterNumericGreater();
// rule value
double ruleValue = 100.0f; // filter room whose area is greater than 100 SF
// rule
FilterRule fRule = new FilterDoubleRule(pvp, fnrv, ruleValue, 1E-6);

// Create an ElementParameter filter
ElementParameterFilter filter = new ElementParameterFilter(fRule);

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector(document);
IList<Element> rooms = collector.WherePasses(filter).ToElements();

// Find rooms whose area is less than or equal to 100:
// Use inverted filter to match elements
ElementParameterFilter lessOrEqualFilter = new ElementParameterFilter(fRule, true);
collector = new FilteredElementCollector(document);
IList<Element> lessOrEqualFounds = collector.WherePasses(lessOrEqualFilter).ToElements();
```

The following example shows how to use the FamilyStructuralMaterialTypeFilter to find all families whose material type is wood. It also shows how to use an inverted filter to find all families whose material type is not wood.

Code Region 6-7: Find all families with wood material

```
// Use FamilyStructuralMaterialType filter to find families whose material type is Wood
FamilyStructuralMaterialTypeFilter filter =
    new FamilyStructuralMaterialTypeFilter(StructuralMaterialType.Wood);

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector(document);
ICollection<Element> woodFamiles = collector.WherePasses(filter).ToElements();

// Find families are not Wood: Use inverted filter to match families
FamilyStructuralMaterialTypeFilter notWoodFilter =
    new FamilyStructuralMaterialTypeFilter(StructuralMaterialType.Wood, true);
collector = new FilteredElementCollector(document);
ICollection<Element> notWoodFamilies = collector.WherePasses(notWoodFilter).ToElements();
```

6.2.3 **Logical filters**

Logical filters combine two or more filters logically. The following table summarizes the built-in logical filters.

Built-in Filter	What it passes	Shortcut Method(s)
LogicalAndFilter	Elements that pass 2 or more filters	WherePasses()- adds one additional filter IntersectWith() - joins two sets of independent filters
LogicalOrFilter	Elements that pass at least one of 2 or more filters	UnionWith() - joins two sets of independent filters

Table 14: Built-in Logical Filters

In the example below, two quick filters are combined using a logical filter to get all door FamilyInstance elements in the document.

Code Region 6-8: Using LogicalAndFilter to find all door instances

```
// Find all door instances in the project by finding all elements that both belong to the
// door category and are family instances.
ElementClassFilter familyInstanceFilter = new ElementClassFilter(typeof(FamilyInstance));

// Create a category filter for Doors
ElementCategoryFilter doorsCategoryfilter =
        new ElementCategoryFilter(BuiltInCategory.OST_Doors);

// Create a logic And filter for all Door FamilyInstances
LogicalAndFilter doorInstancesFilter = new LogicalAndFilter(familyInstanceFilter,
        doorsCategoryfilter);

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector(document);
IList<Element> doors = collector.WherePasses(doorInstancesFilter).ToElements();
```

6.3 Getting filtered elements or element ids

Once one or more filters have been applied to the FilteredElementCollector, the filtered set of elements can be retrieved in one of three ways:

1. Obtain a collection of Elements or ElementIds.
   - ToElements() – returns all elements that pass all applied filters
   - ToElementIds() – returns ElementIds of all elements which pass all applied filters
2. Obtain the first Element or ElementId that matches the filter.
   - FirstElement() – returns first element to pass all applied filters
   - FirstElementId() – returns id of first element to pass all applied filters
3. Obtain an ElementId or Element iterator.
   - GetElementIdIterator() – returns FilteredElementIdIterator to the element ids passing the filters
   - GetElementIterator() – returns FilteredElementIterator to the elements passing the filters
   - GetEnumerator() – returns an IEnumerator<Element> that iterates through collection of passing elements

You should only use one of the methods from these groups at a time; the collector will reset if you call another method to extract elements. Thus, if you have previously obtained an iterator, it will be stopped and traverse no more elements if you call another method to extract elements.

Which method is best depends on the application. If just one matching element is required, FirstElement() or FirstElementId() is the best choice. If all the matching elements are required, use ToElements(). If a variable number are needed, use an iterator.

If the application will be deleting elements or making significant changes to elements in the filtered list, ToElementIds() or an element id iterator are the best options. This is because deleting elements or making significant changes to an element can invalidate an element handle. With

element ids, the call to Document.Element with the ElementId will always return a valid Element (or a null reference if the element has been deleted).

Using the ToElements() method to get the filter results as a collection of elements allows for the use of foreach to examine each element in the set, as is shown below:

Code Region 6-9: Using ToElements() to get filter results

```
// Use ElementClassFilter to find all loads in the document
// Using typeof(LoadBase) will yield all AreaLoad, LineLoad and PointLoad
ElementClassFilter filter = new ElementClassFilter(typeof(LoadBase));

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector(document);
collector.WherePasses(filter);
ICollection<Element> allLoads = collector.ToElements();

String prompt = "The loads in the current document are:\n";
foreach (Element loadElem in allLoads)
{
     LoadBase load = loadElem as LoadBase;
     prompt += load.GetType().Name +  ": " +
               load.Name + "\n";
}

TaskDialog.Show("Revit", prompt);
```

When just one passing element is needed, use FirstElement():

Code Region 6-10: Get the first passing element

```
// Create a filter to find all columns
StructuralInstanceUsageFilter columnFilter =
     new StructuralInstanceUsageFilter(StructuralInstanceUsage.Column);

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector(document);
collector.WherePasses(columnFilter);

// Get the first column from the filtered results
// Element will be a FamilyInstance
FamilyInstance column = collector.FirstElement() as FamilyInstance;
```

In some cases, FirstElement() is not sufficient. This next example shows how to use extension methods to get the first non-template 3D view (which is useful for input to FindReferencesByDirection()).

Code Region 6-11: Get first passing element using extension methods

```
// Use filter to find a non-template 3D view
// This example does not use FirstElement() since first filterd view3D might be a template
FilteredElementCollector collector = new FilteredElementCollector(document);
```

```
Func<View3D, bool> isNotTemplate = v3 => !(v3.IsTemplate);

// apply ElementClassFilter
collector.OfClass(typeof(View3D));

// use extension methods to get first non-template View3D
View3D view3D = collector.Cast<View3D>().First<View3D>(isNotTemplate);
```

The following example demonstrates the use of the FirstElementId() method to get one passing element (a 3d view in this case) and the use of ToElementIds() to get the filter results as a collection of element ids (in order to delete a set of elements in this case).

Code Region 6-12: Using Getting filter results as element ids

```
FilteredElementCollector collector = new FilteredElementCollector(document);

// Use shortcut OfClass to get View elements
collector.OfClass(typeof(View3D));

// Get the Id of the first view
ElementId viewId = collector.FirstElementId();

// Test if the view is valid for element filtering
if (FilteredElementCollector.IsViewValidForElementIteration(document, viewId))
{
    FilteredElementCollector viewCollector = new FilteredElementCollector(document, viewId);

    // Get all FamilyInstance items in the view
    viewCollector.OfClass(typeof(FamilyInstance));
    ICollection<ElementId> familyInstanceIds = viewCollector.ToElementIds();

    document.Delete(familyInstanceIds);
}
```

The GetElementIterator() method is used in the following example that iterates through the filtered elements to check the flow state of some pipes.

Code Region 6-13: Getting the results as an element iterator

```
FilteredElementCollector collector = new FilteredElementCollector(document);

// Apply a filter to get all pipes in the document
collector.OfClass(typeof(Autodesk.Revit.DB.Plumbing.Pipe));

// Get results as an element iterator and look for a pipe with
// a specific flow state
FilteredElementIterator elemItr = collector.GetElementIterator();
elemItr.Reset();
while (elemItr.MoveNext())
{
```

```
    Pipe pipe = elemItr.Current as Pipe;
    if (pipe.FlowState == PipeFlowState.LaminarState)
    {
        TaskDialog.Show("Revit", "Model has at least one pipe with Laminar flow state.");
        break;
    }
}
```

Alternatively, the filter results can be returned as an element id iterator:

Code Region 6-14: Getting the results as an element id iterator

```
// Use a RoomFilter to find all room elements in the document.
RoomFilter filter = new RoomFilter();

// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector(document);
collector.WherePasses(filter);

// Get results as ElementId iterator
FilteredElementIdIterator roomIdItr = collector.GetElementIdIterator();
roomIdItr.Reset();
while (roomIdItr.MoveNext())
{
    ElementId roomId = roomIdItr.Current;
    // Warn rooms smaller than 50 SF
    Room room = document.get_Element(roomId) as Room;
    if (room.Area < 50.0)
    {
        String prompt = "Room is too small: id = " + roomId.ToString();
        TaskDialog.Show("Revit", prompt);
        break;
    }
}
```

6.4 LINQ Queries

In .NET, the FilteredElementCollector class supports the IEnumerable interface for Elements. You can use this class with LINQ queries and operations to process lists of elements. Note that because the ElementFilters and the shortcut methods offered by this class process elements in native code before their managed wrappers are generated, better performance will be obtained by using as many native filters as possible on the collector before attempting to process the results using LINQ queries.

The following example uses an ElementClassFilter to get all FamilyInstance elements in the document, and then uses a LINQ query to narrow down the results to those FamilyInstances with a specific name.

Code Region 6-15: Using LINQ query

```
// Use ElementClassFilter to find family instances whose name is 60" x 30" Student
ElementClassFilter filter = new ElementClassFilter(typeof(FamilyInstance));
```

```
// Apply the filter to the elements in the active document
FilteredElementCollector collector = new FilteredElementCollector(document);
collector.WherePasses(filter);

// Use Linq query to find family instances whose name is 60" x 30" Student
var query = from element in collector
            where element.Name == "60\" x 30\" Student"
            select element;

// Cast found elements to family instances,
// this cast to FamilyInstance is safe because ElementClassFilter for FamilyInstance was used
List<FamilyInstance> familyInstances = query.Cast<FamilyInstance>().ToList<FamilyInstance>();
```

7 Selection

You can get the selected objects from the current active document using the UIDocument.Selection.Elements property. The selected objects are in an ElementSet in Revit. From this Element set, all selected Elements are retrieved. The Selection object can also be used to change the current selection programmatically.

7.1 Changing the Selection

To modify the Selection.Elements:

1. Create a new SelElementSet.
2. Put Elements in it.
3. Set the Selection.Elements to the new SelElementSet instance.

The following example illustrates how to change the selected Elements.

Code Region 7-1: Changing selected elements

```
private void ChangeSelection(Document document)
{
    // Get selected elements form current document.
    UIDocument uidoc = new UIDocument(document);
    Autodesk.Revit.UI.Selection.SelElementSet collection = uidoc.Selection.Elements;

    // Display current number of selected elements
    TaskDialog.Show("Revit","Number of selected elements: " + collection.Size.ToString());

    //Create a new SelElementSet
    SelElementSet newSelectedElementSet = SelElementSet.Create();

    // Add wall into the created element set.
    foreach (Autodesk.Revit.DB.Element elements in collection)
    {
         if (elements is Wall)
        {
            newSelectedElementSet.Add(elements);
        }
    }

    // Set the created element set as current select element set.
    uidoc.Selection.Elements = newSelectedElementSet;

    // Give the user some information.
    if (0 != newSelectedElementSet.Size)
    {
        TaskDialog.Show("Revit",uidoc.Selection.Elements.Size.ToString() +
            " Walls are selected!");
    }
    else
    {
```

```
            TaskDialog.Show("Revit","No Walls have been selected!");
        }
    }
```

7.2 User Selection

The Selection class also has methods for allowing the user to select new objects, or even a point on screen. This allows the user to select one or more Elements (or other objects, such as an edge or a face) using the cursor and then returns control to your application. These functions do not automatically add the new selection to the active selection collection.

- The PickObject() method prompts the user to select an object in the Revit model.
- The PickObjects() method prompts the user to select multiple objects in the Revit model.
- The PickElementsByRectangle() method prompts the user to select multiple elements using a rectangle.
- The PickPoint() method prompts the user to pick a point in the active sketch plane.

The type of object to be selected is specified when calling PickObject() or PickObjects. Types of objects that can be specified are: Element, PointOnElement, Edge or Face.

The StatusbarTip property shows a message in the status bar when your application prompts the user to pick objects or elements. Each of the Pick functions has an overload that has a String parameter in which a custom status message can be provided.

Code Region 7-2: Adding selected elements with PickObject() and PickElementsByRectangle()

```
UIDocument uidoc = new UIDocument(document);
Selection choices = uidoc.Selection;
// Pick one object from Revit.
Reference hasPickOne = choices.PickObject(ObjectType.Element);
if (hasPickOne != null)
{
    TaskDialog.Show("Revit", "One element added to Selection.");
}

int selectionCount = choices.Elements.Size;
// Choose objects from Revit.
IList<Element> hasPickSome = choices.PickElementsByRectangle("Select by rectangle");
if (hasPickSome.Count > 0)
{
    int newSelectionCount = choices.Elements.Size;
    string prompt = string.Format("{0} elements added to Selection.",
        newSelectionCount - selectionCount);
    TaskDialog.Show("Revit", prompt);
}
```

The PickPoint() method has 2 overloads with an ObjectSnapTypes parameter which is used to specify the type of snap types used for the selection. More than one can be specified, as shown in the next example.

Code Region 7-3: Snap points

```
public void PickPoint(UIDocument uidoc)
{
    ObjectSnapTypes snapTypes = ObjectSnapTypes.Endpoints | ObjectSnapTypes.Intersections;
    XYZ point = uidoc.Selection.PickPoint(snapTypes, "Select an end point or intersection");

    string strCoords = "Selected point is " + point.ToString();

    TaskDialog.Show("Revit", strCoords);
}
```

7.3 Filtered User Selection

PickObject(), PickObjects() and PickElementsByRectangle() all have overloads that take an ISelectionFilter as a parameter. ISelectionFilter is an interface that can be implemented to filter objects during a selection operation. It has two methods that can be overridden: AllowElement() which is used to specify if an element is allowed to be selected, and AllowReference() which is used to specify if a reference to a piece of geometry is allowed to be selected.

The following example illustrates how to use an ISelectionFilter interface to limit the user's selection to elements in the Mass category. It does not allow any references to geometry to be selected.

Code Region 7-4: Using ISelectionFilter to limit element selection

```
public static IList<Element> GetManyRefByRectangle(UIDocument doc)
{
    ReferenceArray ra = new ReferenceArray();
    ISelectionFilter selFilter = new MassSelectionFilter();
    IList<Element> eList = doc.Selection.PickElementsByRectangle(selFilter,
        "Select multiple faces") as IList<Element>;
    return eList;
}

public class MassSelectionFilter : ISelectionFilter
{
    public bool AllowElement(Element element)
    {
        if (element.Category.Name == "Mass")
        {
            return true;
        }
        return false;
    }

    public bool AllowReference(Reference refer, XYZ point)
    {
        return false;
    }
}
```

The next example demonstrates the use of ISelectionFilter to allow only planar faces to be selected.

Code Region 7-5: Using ISelectionFilter to limit geometry selection

```
public void SelectPlanarFaces(Autodesk.Revit.DB.Document document)
{
    UIDocument uidoc = new UIDocument(document);
    ISelectionFilter selFilter = new PlanarFacesSelectionFilter();
    IList<Reference> faces = uidoc.Selection.PickObjects(ObjectType.Face,
        selFilter, "Select multiple planar faces");
}

public class PlanarFacesSelectionFilter : ISelectionFilter
{
    public bool AllowElement(Element element)
    {
        return true;
    }

    public bool AllowReference(Reference refer, XYZ point)
    {
        if (refer.GeometryObject is PlanarFace)
        {
            return true;
        }

        return false;
    }
}
```

For more information about retrieving Elements from selected Elements, see the Walkthrough: Retrieve Selected Elements section in the Getting Started chapter.

8 Parameters

Revit provides a general mechanism for giving each element a set of parameters that you can edit. In the Revit UI, parameters are visible in the Element Properties dialog box. This chapter describes how to get and use built-in parameters using the Revit Platform API. For more information about user-defined shared parameters, see the Shared Parameters chapter.

In the Revit Platform API, Parameters are managed in the Element class. You can access Parameters in these ways:

- By iterating through the Element.Parameters collection of all parameters for an Element (for an example, see the sample code in Walkthrough: Get Selected Element Parameters below).
- By accessing the parameter directly through the overloaded Element.Parameter property. If the Parameter doesn't exist, the property returns null.
- By accessing a parameter by name via the Element.ParametersMap collection.

You can retrieve the Parameter object from an Element if you know the name string, built-in ID, definition, or GUID. The Parameter[String] property overload gets a parameter based on its localized name, so your code should handle different languages if it's going to look up parameters by name and needs to run in more than one locale.

The Parameter[GUID] property overload gets a shared parameter based on its Global Unique ID (GUID), which is assigned to the shared parameter when it's created.

8.1 Walkthrough: Get Selected Element Parameters

The Element Parameters are retrieved by iterating through the Element ParameterSet. The following code sample illustrates how to retrieve the Parameter from a selected element.

Note: This example uses some Parameter members, such as AsValueString and StorageType, which are covered later in this chapter.

Code Region 8-1: Getting selected element parameters

```
void GetElementParameterInformation(Document document, Element element)
{
     // Format the prompt information string
     String prompt = "Show parameters in selected Element:";

     StringBuilder st = new StringBuilder();
     // iterate element's parameters
     foreach (Parameter para in element.Parameters)
     {
         st.AppendLine(GetParameterInformation(para, document));
     }

     // Give the user some information
     MessageBox.Show(prompt, "Revit", MessageBoxButtons.OK);
}

String GetParameterInformation(Parameter para, Document document)
{
      string defName = para.Definition.Name + @"\t";
      // Use different method to get parameter data according to the storage type
      switch (para.StorageType)
      {
           case StorageType.Double:
               //covert the number into Metric
               defName += " : " + para.AsValueString();
               break;
           case StorageType.ElementId:
               //find out the name of the element
```

```
                ElementId id = para.AsElementId();
                if (id.Value >= 0)
                {
                    defName += " : " + document.get_Element(ref id).Name;
                }
                else
                {
                    defName += " : " + id.Value.ToString();
                }
                break;
            case StorageType.Integer:
                if (ParameterType.YesNo == para.Definition.ParameterType)
                {
                    if (para.AsInteger() == 0)
                    {
                        defName += " : " + "False";
                    }
                    else
                    {
                        defName += " : " + "True";
                    }
                }
                else
                {
                    defName += " : " + para.AsInteger().ToString();
                }
                break;
            case StorageType.String:
                defName += " : " + para.AsString();
                break;
            default:
                defName = "Unexposed parameter.";
            break;
    }

    return defName;
}
```

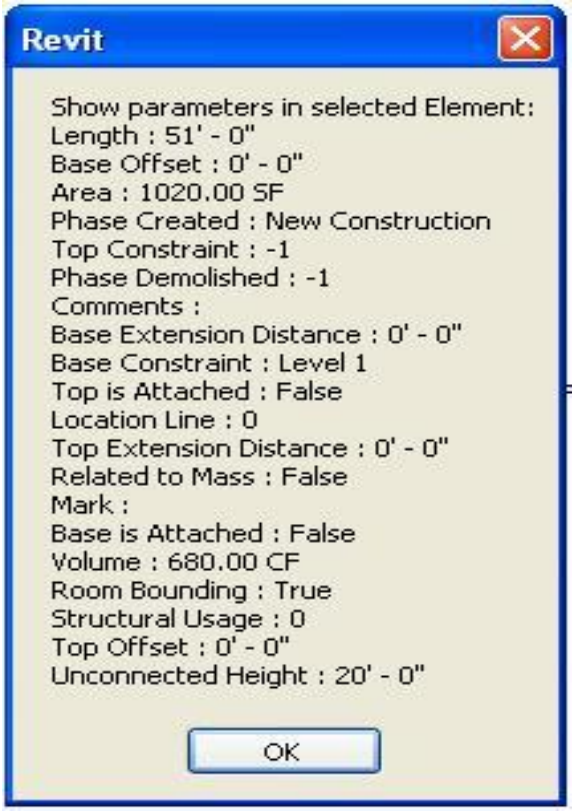


Figure 25: Get wall parameters result

Note: In Revit, some parameters have values in the drop-down list in the Element Properties dialog box. You can get the numeric values corresponding to the enumerated type for the Parameter using the Revit Platform API, but you cannot get the string representation for the values using the Parameter.AsValueString() method.

8.2 Definition

The Definition object describes the data type, name, and other Parameter details. There are two kinds of definition objects derived from this object.

- InternalDefinition represents all kinds of definitions existing entirely in the Revit database.
- ExternalDefinition represents definitions stored on disk in a shared parameter file.

You should write the code to use the Definition base class so that the code is applicable to both internal and external parameter Definitions. The following code sample shows how to find a specific parameter using the definition type.

Code Region 8-2: Finding a parameter based on definition type

```
//Find parameter using the Parameter's definition type.
public Parameter FindParameter(Element element)
{
    Parameter foundParameter = null;
    // This will find the first parameter that measures length
    foreach (Parameter parameter in element.Parameters)
    {
        if (parameter.Definition.ParameterType == ParameterType.Length)
        {
            foundParameter = parameter;
            break;
        }
     }
     return foundParameter;
}
```

8.2.1 **ParameterType**

This property returns parameter data type, which affects how the parameter is displayed in the Revit UI. The ParameterType enumeration members are:

Member name	Description
Number	The parameter data should be interpreted as a real number, possibly including decimal points.
Moment	The data value will be represented as a moment.
AreaForce	The data value will be represented as an area force.
LinearForce	The data value will be represented as a linear force.
Force	The data value will be represented as a force.
YesNo	A boolean value that will be represented as Yes or No.
Material	The value of this property is considered to be a material.
URL	A text string that represents a web address.
Angle	The parameter data represents an angle. The internal representation will be in radians. The user visible representation will be in the units that the user has chosen.
Volume	The parameter data represents a volume. The internal representation will be in decimal cubic feet. The user visible representation will be in the units that the user has chosen.
Area	The parameter data represents an area. The internal representation will be in decimal square feet. The user visible representation will be in the units that the user has chosen.
Integer	The parameter data should be interpreted as a whole number, positive or negative.
Invalid	The parameter type is invalid. This value should not be used.
Length	The parameter data represents a length. The internal representation will be in decimal feet. The user visible representation will be in the units system that the user has chosen.
Text	The parameter data should be interpreted as a string of text.

For more details about ParameterType.Material, see the Material chapter.

8.2.2 ParameterGroup

The Definition class ParameterGroup property returns the parameter definition group ID. The BuiltInParameterGroup is an enumerated type listing all built-in parameter groups supported by Revit. Parameter groups are used to sort parameters in the Element Properties dialog box.

8.3 BuiltInParameter

The Revit Platform API has a large number of built-in parameters, defined in the Autodesk.Revit.Parameters.BuiltInParameter enumeration (see the RevitAPI Help.chm file for the definition of this enumeration). The parameter ID is used to retrieve the specific parameter from an element, if it exists, using the Element.Parameter property. However, not all parameters can be retrieved using the ID. For example, family parameters are not exposed in the Revit Platform API, therefore, you cannot get them using the built-in parameter ID.

The following code sample shows how to get the specific parameter using the BuiltInParameter Id:

Code Region 8-3: Getting a parameter based on BuiltInParameter

```
public Parameter FindWithBuiltinParameterID(Wall wall)
{
    // Use the WALL_BASE_OFFSET paramametId
    // to get the base offset parameter of the wall.
    BuiltInParameter paraIndex = BuiltInParameter.WALL_BASE_OFFSET;
    Parameter parameter = wall.get_Parameter(paraIndex);

    return parameter;
}
```

Note: With the Parameter overload, you can use an Enumerated type BuiltInParameter as the method parameter. For example, use BuiltInParameter.GENERIC_WIDTH.

If you do not know the exact BuiltInParameter ID, get the parameter by iterating the ParameterSet collection.

Another approach for testing or identification purposes is to test each BuiltInParameter using the get_Parameter() method. Use the Enum class to iterate as illustrated in the following code. When you use this method, it is possible that the ParameterSet collection may not contain all parameters returned from the get_Parameter() method, though this is infrequent.

8.4 StorageType

StorageType describes the type of parameter values stored internally.

Based on the property value, use the corresponding get and set methods to retrieve and set the parameter data value.

The StorageType is an enumerated type that lists all internal parameter data storage types supported by Revit:

Member Name	Description
String	Internal data is stored as a string of characters.
ElementId	Data type represents an element and is stored as an Element ID.
Double	Data is stored internally as an 8-byte floating point number.
Integer	Internal data is stored as a signed 32-bit integer.

Member Name	Description
None	None represents an invalid storage type. For internal use only.

Table 15: Storage Type

In most cases, the ElementId value is a positive number. However, it can be a negative number. When the ElementId value is negative, it does not represent an Element but has another meaning. For example, the storage type parameter for a beam's Vertical Projection is ElementId. When the parameter value is Level 1 or Level 2, the ElementId value is positive and corresponds to the ElementId of that level. However, when the parameter value is set to Auto-detect, Center of Beam or Top of Beam, the ElementId value is negative.

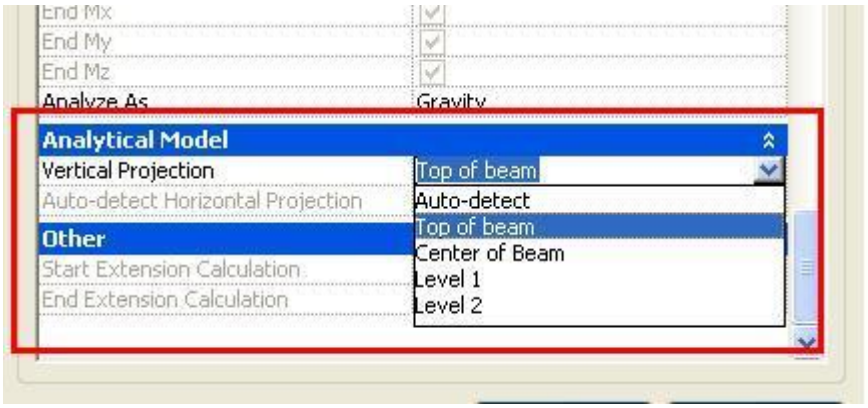


Figure 26: Storage type sample

The following code sample shows how to check whether a parameter's value can be set to a double value, based on its StorageType:

Code Region 8-4: Checking a parameter's StorageType

```
public bool SetParameter(Parameter parameter, double value)
{
     bool result = false;
     //if the parameter is readonly, you can't change the value of it
     if (null != parameter && !parameter.IsReadOnly)
     {
          StorageType parameterType = parameter.StorageType;
          if (StorageType.Double != parameterType)
          {
               throw new Exception("The storagetypes of value and parameter are different!");
          }

          //If successful, the result is true
          result = parameter.Set(value);
      }

     return result;
}
```

The Set() method return value indicates that the Parameter value was changed. The Set() method returns true if the Parameter value was changed, otherwise it returns false.

Not all Parameters are writable. An Exception is thrown if the Parameter is read-only.

8.5 AsValueString() and SetValueString()

AsValueString() and SetValueString() are Parameter class methods. The two methods are only applied to value type parameters, which are double or integer parameters representing a measured quantity.

Use the AsValueString() method to get the parameter value as a string with the unit of measure. For example, the Base Offset value, a wall parameter, is a Double value. Usually the value is shown as a string like -20’0” in the Element Properties dialog box:

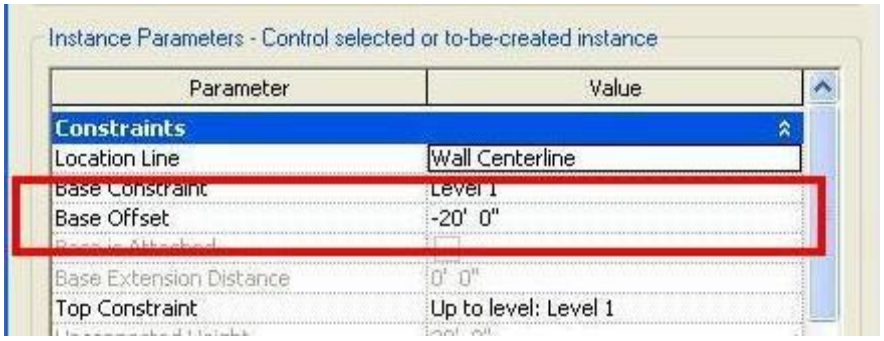


Figure 27: AsValueString and SetValueString sample

Using the AsValueString() method, you get the -20’0” string value directly. Otherwise you get a double value like -20 without the units of measure if you use the AsDouble() method.

Use the SetValueString() method to change the value of a value type parameter instead of using the Set() method. The following code sample illustrates how to change the parameter value using the SetValueString() method:

Code Region 8-5: Using Parameter.SetValueString()

```
public bool SetWithValueString(Parameter foundParameter)
{
    bool result = false;
    if (!foundParameter.IsReadOnly)
    {
        //If successful, the result is true
        result = foundParameter.SetValueString("-22\'3\"");
    }
    return result;
}
```

8.6 Parameter Relationships

There are relationships between Parameters where the value of one Parameter can affect:

- whether another Parameter can be set, or is read-only
- what parameters are valid for the element
- the computed value of another parameter

Additionally, some parameters are always read-only.

Some parameters are computed in Revit, such as wall Length and Area parameter. These parameters are always read-only because they depend on the element’s internal state.

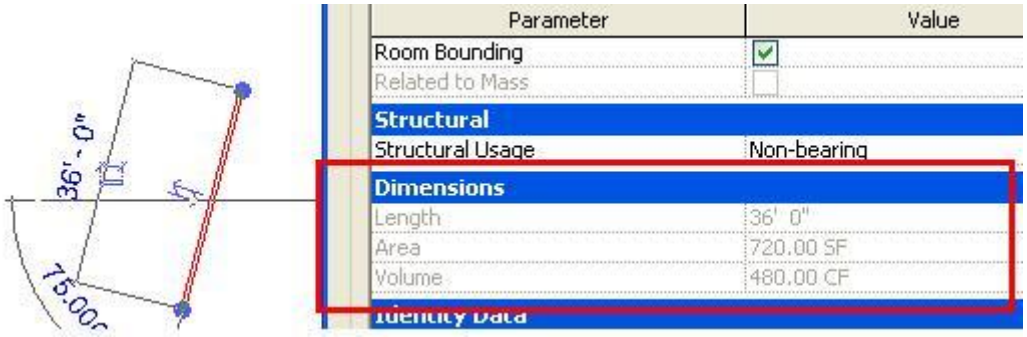


Figure 28: Wall computed parameters

In this code sample, the Sill Height parameter for an opening is adjusted, which results in the Head Height parameter being re-computed:

Code Region 8-6: Parameter relationship example

```
// opening should be an opening such as a window or a door
public void ShowParameterRelationship(FamilyInstance opening)
{
    // get the original Sill Height and Head Height parameters for the opening
    Parameter sillPara = opening.get_Parameter(BuiltInParameter.INSTANCE_SILL_HEIGHT_PARAM);
    Parameter headPara = opening.get_Parameter(BuiltInParameter.INSTANCE_HEAD_HEIGHT_PARAM);
    double sillHeight = sillPara.AsDouble();
    double origHeadHeight = headPara.AsDouble();

    // Change the Sill Height only and notice that Head Height is recalculated
    sillPara.Set(sillHeight + 2.0);
    double newHeadHeight = headPara.AsDouble();
    MessageBox.Show("Old head height: " + origHeadHeight + "; new head height: "
                + newHeadHeight);
}
```

8.7 Adding Parameters to Elements

In the Revit Platform API, you can use all of the defined parameters and you can add custom parameters that you define using the Revit user interface and the Revit Platform API.

For more details, refer to the Shared Parameter chapter.

9 Collections

Most Revit Platform API properties and methods use customized collection classes defined in the Revit Platform API when providing access to a group of related items. These collections are of one of three basic types:

- Array
- Set
- Map

These collection classes were written before generic type-safe .NET classes like List<T> existed.

The IEnumerable and IEnumerator interfaces implemented in Revit collection types are defined in the System.Collection namespace.

9.1 Interface

The following sections discuss interface-related collection types.

9.1.1 IEnumerable

The IEnumerable interface is in the System.Collections namespace. It exposes the enumerator, which supports a simple iteration over a non-generic collection. The GetEnumerator() method gets an enumerator that implements this interface. The returned IEnumerator object is iterated throughout the collection. The GetEnumerator() method is used implicitly by foreach loops in C#.

9.1.2 IEnumerator

The IEnumerator interface is in the System.Collections namespace. It supports a simple iteration over a non-generic collection. IEnumerator is the base interface for all non-generic enumerators. The foreach statement in C# hides the enumerator's complexity.

Note: Using foreach is recommended instead of directly manipulating the enumerator.

Enumerators are used to read the collection data, but they cannot be used to modify the underlying collection.

Use IEnumerator as follows:

- Initially, the enumerator is positioned in front of the first element in the collection. However, it is a good idea to always call Reset() when you first obtain the enumerator.
  - The Reset() method moves the enumerator back to the original position. At this position, calling the Current property throws an exception.
  - Call the MoveNext() method to advance the enumerator to the collection's first element before reading the current iterator value.
- The Current property returns the same object until either the MoveNext() method or Reset() method is called. The MoveNext() method sets the current iterator to the next element.
- If MoveNext passes the end of the collection, the enumerator is positioned after the last element in the collection and MoveNext returns false.
  - When the enumerator is in this position, subsequent calls to the MoveNext also return false.
  - If the last call to the MoveNext returns false, calling the Current property throws an exception.

    - To set the current iterator to the first element in the collection again, call the Reset() method followed by MoveNext().
- An enumerator remains valid as long as the collection remains unchanged.
    - If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is invalidated and the next call to the MoveNext() or the Reset() method throws an InvalidOperationException.
    - If the collection is modified between the MoveNext and the current iterator, the Current property returns to the specified element, even if the enumerator is already invalidated.

Note: All calls to the Reset() method must result in the same state for the enumerator. The preferred implementation is to move the enumerator to the collection beginning, before the first element. This invalidates the enumerator if the collection is modified after the enumerator was created, which is consistent with the MoveNext() and the Current properties.

9.2 Collections and Iterators

In the Revit Platform API, the Collections namespace mainly contains Array, Map, Set collections and the relevant Iterators. API Collections and Iterators are generic and type safe. For example, ElementSet and ElementIterator always contain Element and can be used as follows:

Code Region 9-1: Using ElementSet and ElementIterator

```
UIDocument uidoc = new UIDocument(document);
ElementSet elems = uidoc.Selection.Elements;

string info = "Selected elements:\n";
foreach (Autodesk.Revit.DB.Element elem in elems)
{
    info += elem.Name + "\n";
}

TaskDialog.Show("Revit",info);

info = "Levels in document:\n";

FilteredElementCollector collector = new FilteredElementCollector(document);
ICollection<Element> collection = collector.OfClass(typeof(BoundaryConditions)).ToElements();
foreach (Element elem in collection)
{
    // you need not check null for elem
    info += elem.Name + "\n";
}

TaskDialog.Show("Revit",info);
```

All collections implement the IEnumerable interface and all relevant iterators implement the IEnumerator interface. As a result, all methods and properties are implemented in the Revit Platform API and can play a role in the relevant collections.

The following table compares the Array, Map, and Set methods and properties.

Function	Array	Map	Set
Add the item to the end of the collection.	Append(Object)		
Removes every item from the collection rendering it empty.	Clear()	Clear()	Clear()
Tests for the existence of a key within the collection		Contains(Object)	Contains(Object)
Removes an object with the specified key from the collection		Erase(Object)	Erase(Object)
Retrieve a forward moving iterator to the collection.	ForwardIterator()	ForwardIterator()	ForwardIterator()
Retrieve a forward moving iterator to the collection.	GetEnumerator()	GetEnumerator()	GetEnumerator()
Insert the specified item into the collection.	Insert(Object, Int32)	Insert(Object, Object)	Insert(Object)
Retrieve a backward moving iterator to the collection.	ReverseIterator()	ReverseIterator()	ReverseIterator()
Test to see if the collection is empty.	IsEmpty{get;}	IsEmpty{get;}	IsEmpty{get;}
Gets or sets an item at a specified index (key) within the array (map)	Item(Int32) {get; set;}	Item(Object) {get; set;}	
Returns the number of objects in the collection.	Size{get;}	Size{get;}	Size{get;}

Table 16: Collections Methods and Properties

Note: The GetEnumerator() method uses the ForwardIterator() method in its core code. Using the ForwardIterator() method to get the corresponding Iterator works the same way as using GetEnumerator().

In the Revit Platform API, Collections have their own Iterators that implement the IEnumerator interface. Use the GetEnumerator() method (or ForwardIterator) to get the Iterator. To implement the IEnumerator interface, the Iterator requires the following:

- MoveNext() method
- Reset() method
- The Current property

Function	Array	Map	Set
Move the iterator one item forward.	MoveNext()	MoveNext()	MoveNext()
Bring the iterator back to the start of the array.	Reset()	Reset()	Reset()
Retrieve the current focus of the iterator.	Current{get;}	Current{get;}	Current{get;}

Table 17: Method and Property Iterators

In general, Array, Map, and Set are not used generically. Instead, use a collection that has an idiographic type, such as ElementSet.

Implementing all of the collections is similar. The following example uses ElementSet and ModelCurveArray to demonstrate how to use the main collection properties:

Code Region 9-2: Using collections

```
UIDocument uidoc = new UIDocument(document);
SelElementSet selection = uidoc.Selection.Elements;
// Store the ModelLine references
ModelCurveArray lineArray = new ModelCurveArray();

// … Store operation
//assume 131943 is a model line element id
Autodesk.Revit.DB.ElementId id = new Autodesk.Revit.DB.ElementId(131943);
lineArray.Append(document.get_Element(id) as ModelLine);

// use Size property of Array
TaskDialog.Show("Revit","Before Insert: " + lineArray.Size + " in lineArray.");

// use IsEmpty property of Array
if (!lineArray.IsEmpty)
{
    // use Item(int) property of Array
    ModelCurve modelCurve = lineArray.get_Item(0) as ModelCurve;

    // erase the specific element from the set of elements
    selection.Erase(modelCurve);

    // create a new model line and insert to array of model line
    SketchPlane sketchPlane = modelCurve.SketchPlane;

    XYZ startPoint = new XYZ(0, 0, 0);  // the start point of the line
    XYZ endPoint = new XYZ(10, 10, 0);  // the end point of the line
    // create geometry line
    Line geometryLine = document.Application.Create.NewLine(startPoint, endPoint, true);

    // create the ModelLine
    ModelLine line = document.Create.NewModelCurve(geometryLine, sketchPlane) as ModelLine;

    lineArray.Insert(line, lineArray.Size - 1);
}

TaskDialog.Show("Revit","After Insert: " + lineArray.Size + " in lineArray.");

// use the Clear() method to remove all elements in lineArray
lineArray.Clear();

TaskDialog.Show("Revit","After Clear: " + lineArray.Size + " in lineArray.");
```

10 Editing Elements

In Revit, you can move, rotate, delete, mirror, group, and array one element or a set of elements with the Revit Platform API. Using the editing functionality in the API is similar to the commands in the Revit UI.

10.1 Moving Elements

The Revit Platform API provides Document.Move() methods to move one or more elements from one place to another.

Member	Description
Move(Element, XYZ)	Move an element in the project by a specified vector.
Move(ElementId, XYZ)	Move an element by ID in the project by a specified vector.
Move(ICollection<ElementId>, XYZ)	Move several elements by a set of IDs in the project by a specified vector.
Move(ElementSet, XYZ)	Move several elements in the project by a specified vector.

Table 18: Move Members

Note: When you use the Move() method, the following rules apply.

- The Move() method cannot move a level-based element up or down from the level. When the element is level-based, you cannot change the Z coordinate value. However, you can place the element at any location in the same level. As well, some level based elements have an offset instance parameter you can use to move them in the Z direction.
- For example, if you create a new column at the original location (0, 0, 0) in Level1, and then move it to the new location (10, 20, 30), the column is placed at the location (10, 20, 0) instead of (10, 20, 30), and the Move() method returns success.

Code Region 10-1: Using Move()

```
public void MoveColumn(Autodesk.Revit.DB.Document document, FamilyInstance column)
{
    // get the column current location
    LocationPoint columnLocation = column.Location as LocationPoint;

    XYZ oldPlace = columnLocation.Point;

    // Move the column to new location.
    XYZ newPlace = new XYZ(10, 20, 30);
    document.Move(column, newPlace);

    // now get the column's new location
    columnLocation = column.Location as LocationPoint;
    XYZ newActual = columnLocation.Point;

    string info = "Original Z location: " + oldPlace.Z +
                          "\nNew Z location: " + newActual.Z;

    TaskDialog.Show("Revit", info);
```

```
}
```

- When you move one or more elements, associated elements are moved. For example, if a wall with windows is moved, the windows are also moved.
- If you pin the element in Revit, the Element.Pinned property is true. This means that the element cannot be moved or rotated.

Another way to move an element in Revit is to use Location and its derivative objects. In the Revit Platform API, the Location object provides the ability to translate and rotate elements. More location information and control is available using the Location object derivatives such as LocationPoint or LocationCurve. If the Location element is downcast to a LocationCurve object or a LocationPoint object, move the curve or the point to a new place directly.

Code Region 10-2: Moving using Location

```
bool MoveUsingLocationCurve(Autodesk.Revit.Application application, Wall wall)
{
    LocationCurve wallLine = wall.Location as LocationCurve;
    XYZ translationVec = new XYZ(10, 20, 0);
    return (wallLine.Move(translationVec));
}
```

When you move the element, note that the vector (10, 20, 0) is not the destination but the offset. The following picture illustrates the wall position before and after moving.

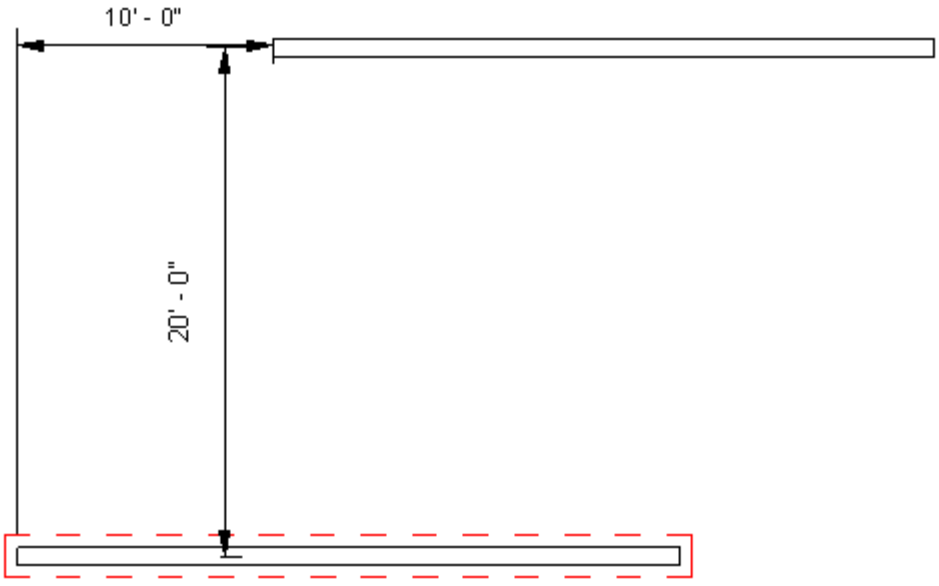


Figure 29: Move a wall using the LocationCurve

In addition, you can use the LocationCurve Curve property or the LocationPoint Point property to move one element in Revit.

Use the Curve property to move a curve-driven element to any specified position. Many elements are curve-driven, such as walls, beams, and braces. Also use the property to resize the length of the element.

Code Region 10-3: Moving using Curve

```
void MoveUsingCurveParam(Autodesk.Revit.ApplicationServices.Application application, Wall
wall)
{
    LocationCurve wallLine = wall.Location as LocationCurve;
    XYZ p1 = XYZ.Zero;
    XYZ p2 = new XYZ(10, 20, 0);
    Line newWallLine = application.Create.NewLineBound(p1, p2);

    // Change the wall line to a new line.
    wallLine.Curve = newWallLine;
}
```

You can also get or set a curve-based element's join properties with the LocationCurve.JoinType property.

Use the LocationPoint Point property to set the element's physical location.

Code Region 10-4: Moving using Point

```
void LocationMove(FamilyInstance column)
{
    LocationPoint columnPoint = column.Location as LocationPoint;
    if (null != columnPoint)
    {
        XYZ newLocation = new XYZ(10, 20, 0);
        // Move the column to the new location
        columnPoint.Point = newLocation;
    }
}
```

10.2 Rotating elements

The Revit Platform API uses the Document.Rotate() methods to rotate one or several elements in the project.

Member	Description
Rotate(Element, Line, Double)	Rotate an element in the project by a specified number of radians around a given axis.
Rotate(ElementId, Line, Double)	Rotate an element by ID in the project by a specified number of radians around a given axis.
Rotate(ICollection<ElementId>, Line, Double)	Rotate several elements by IDs in the project by a specified number of radians around a given axis.
Rotate(ElementSet, Line, Double)	Rotate several elements in the project by a specified number of radians around a given axis.

Table 19: Rotate Members

In the Rotate() methods, the angle of rotation is in radians. The positive radian means rotating counterclockwise around the specified axis, while the negative radian means clockwise, as the following pictures illustrates.

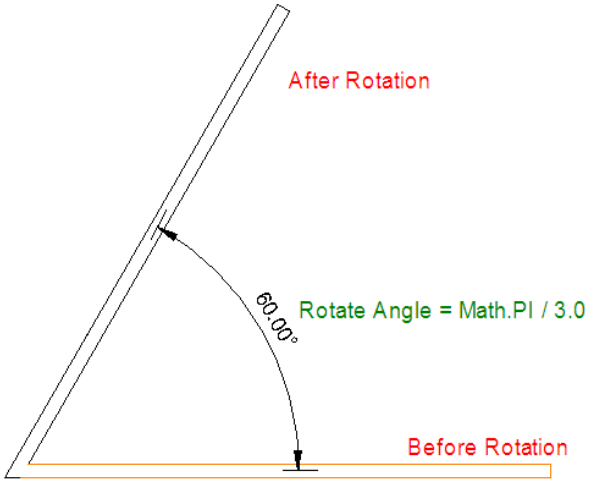


Figure 30: Counterclockwise rotation

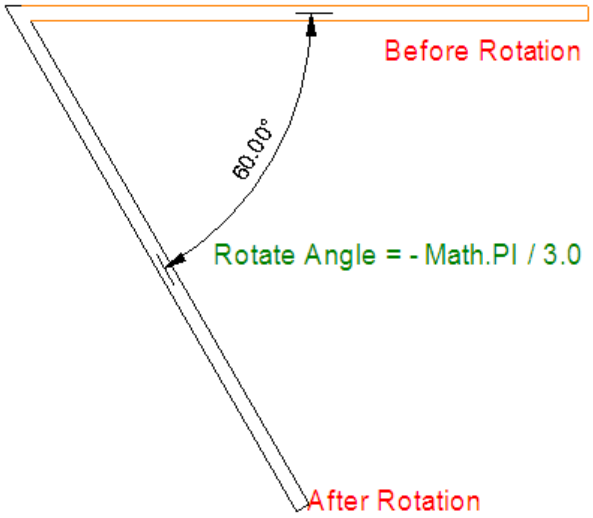


Figure 31: Clockwise rotation

Note: When you rotate an element, the axis line must be bound. An unbounded axis line results in a failed rotation even if Succeeded is returned to Revit. In this situation, the Rotate() method still returns a true value.

Code Region 10-5: Using Rotate()

```
public bool RotateColumn(Autodesk.Revit.DB.Document document, Autodesk.Revit.Element element)
{
    XYZ point1 = new XYZ(10, 20, 0);
    XYZ point2 = new XYZ(10, 20, 30);
    // The axis should be a bound line.
```

```
    Line axis = document.Application.Create.NewLineBound(point1, point2);
    bool successful = document.Rotate(element, axis, Math.PI / 3.0);

    return successful;
}
```

If the element Location can be downcast to a LocationCurve or a LocationPoint, you can rotate the curve or the point directly.

Code Region 10-6: Rotating based on location curve

```
bool LocationRotate(Autodesk.Revit.ApplicationServices.Application application,
Autodesk.Revit.Element element)
{
    bool rotated = false;
    // Rotate the element via its location curve.
    LocationCurve curve = element.Location as LocationCurve;
    if (null != curve)
    {
        Curve line = curve.Curve;
        XYZ aa = line.get_EndPoint(0);
        XYZ cc = new XYZ(aa.X, aa.Y, aa.Z + 10);
        Line axis = application.Create.NewLineBound(aa, cc);
        rotated = curve.Rotate(axis, Math.PI / 2.0);
    }

    return rotated;
}
```

Code Region 10-7: Rotating based on location point

```
bool LocationRotate(Autodesk.Revit.Application application, Autodesk.Revit.Element element)
{
    bool rotated = false;
    LocationPoint location = element.Location as LocationPoint;

    if (null != location)
    {
        XYZ aa = location.Point;
        XYZ cc = new XYZ(aa.X, aa.Y, aa.Z + 10);
        Line axis = application.Create.NewLineBound(aa, cc);
        rotated = location.Rotate(axis, Math.PI / 2.0);
    }

    return rotated;
}
```

10.3 Mirror

The Revit Platform API uses the Document.Mirror() method to mirror one or more elements in the project.

Member	Description
Mirror(Element, Reference)	Mirror one element by a reference.
Mirror(ElementId, Reference)	Mirror one element by a reference and the element ID.
Mirror(ICollection<ElementId>, Reference)	Mirror several elements by a reference and their IDs.
Mirror(ElementSet, Reference)	Mirror several elements by a reference.
Mirror(ElementSet, Line)	Mirror one or several elements by a line.

Table 20: Mirror Members

There are two ways to mirror one element and two ways to mirror several elements. After performing the mirror operation, you can access the new elements from the Selection ElementSet. The following code illustrates how to mirror a column using its reference line, then move the new column to a new location.

Note: This code only works in Revit Structure.

Code Region 10-8: Mirroring a column

```
public void MirrorColumn(Autodesk.Revit.DB.Document document, FamilyInstance column)
{
    UIDocument uidoc = new UIDocument(document);
    AnalyticalModel analytical = column.GetAnalyticalModel() as AnalyticalModel;
    Reference lineReference = analytical.GetCurve().Reference;
    document.Mirror(column, lineReference);

    //The selected element is changed after the Mirror operation.
    foreach (Autodesk.Revit.DB.FamilyInstance newColumn in uidoc.Selection.Elements)
    {
        bool isMirror = newColumn.Mirrored;

        TaskDialog.Show("Revit","Column is mirrored: " + isMirror);

        //Offset the new column
        XYZ translationVec = new XYZ(10, 20, 30);
        newColumn.Location.Move(translationVec);
    }
}
```

Every FamilyInstance has a Mirrored property. It indicates whether a FamilyInstance (for example a column) is mirrored. In the previous example, if the column Mirrored property returns true, then you have a mirrored column.

10.4 Group

The Revit Platform API uses the Creation.Document.NewGroup() method to select an element or multiple elements or groups and then combines them. With each instance of a group that you

place, there is associatively among them. For example, you create a group with a bed, walls, and window and then place multiple instances of the group in your project. If you modify a wall in one group, it changes for all instances of that group. This makes modifying your building model much easier because you can change several instances of a group in one operation.

Code Region 10-9: Creating a Group

```
Group group = null;
UIDocument uidoc = new UIDocument(document);
ElementSet selection = uidoc.Selection.Elements;
if (selection.Size > 0)
{
    // Group all selected elements
    group = document.Create.NewGroup(selection);
}
```

Initially, the group has a generic name, such as Group 1. It can be modified by changing the name of the group type as follows:

Code Region 10-10: Naming a Group

```
// Change the default group name to a new name "MyGroup"
group.GroupType.Name = "MyGroup";
```

There are three types of groups in Revit; Model Group, Detail Group, and Attached Detail Group. All are created using the NewGroup() method. The created Group's type depends on the Elements passed.

- If no detail Element is passed, a Model Group is created.
- If all Elements are detail elements, then a Detail Group is created.
- If both types of Elements are included, a Model Group that contains an Attached Detail Group is created and returned.

Note: When elements are grouped, they can be deleted from the project.

- When a model element in a model group is deleted, it is still visible when the mouse cursor hovers over or clicks the group, even if the application returns Succeeded to the UI. In fact, the model element is deleted and you cannot select or access that element.
- When the last member of a group instance is deleted, excluded, or removed from the project, the model group instance is deleted.

When elements are grouped, they cannot be moved or rotated. If you perform these operations on the grouped elements, nothing happens to the elements, though the Move() or Rotate() method returns true.

You cannot group dimensions and tags without grouping the elements they reference. If you do, the API call will fail.

You can group dimensions and tags that refer to model elements in a model group. The dimensions and tags are added to an attached detail group. The attached detail group cannot be moved, copied, rotated, arrayed, or mirrored without doing the same to the parent group.

10.5 Array

The Revit Platform API uses the overloaded Document.Array() methods to array one or more elements in the project. These methods create a linear or radial array of one or more selected

components. Linear arrays represent an array created along a line from one point, while radial arrays represent an array created along an arc.

As an example of using an array, you can select a door and windows located in the same wall and then create multiple instances of the door, wall, and window configuration.

Member	Description	Array Type
Array(Element, Int32, Boolean, XYZ)	Array one element in the project by a specified number.	Linear
Array(Element, Int32, Boolean, Double)	Array one or several elements in the project by a specified degree.	Radial
Array(ElementId, Int32, Boolean, XYZ)	Array one or several elements in the project by a specified number.	Linear
Array(ElementId, Int32, Boolean, Double)	Array one or several elements in the project by a specified degree.	Radial
Array(ICollection<ElementId>, Int32, Boolean, XYZ)	Array one or several elements in the project by a specified number.	Linear
Array(ICollection<ElementId>, Int32, Boolean, Double)	Array one or several elements in the project by a specified degree.	Radial
Array(ElementSet, Int32, Boolean, XYZ)	Array one or several elements in the project by a specified number.	Linear
Array(ElementSet, Int32, Boolean, Double)	Array one or several elements in the project by a specified degree.	Radial

Table 21 : **Document Array Members**

The Array() method is useful if you need to create several instances of a component and manipulate them simultaneously. Every instance in an array can be a member of a group.

Note: When using the Array() method, the following rules apply:

- When performing Linear and Radial Array operations, elements dependent on the arrayed elements are also arrayed.
- Some elements cannot be arrayed because they cannot be grouped. See the Revit User's Guide for more information about restrictions on groups and arrays.
- Arrays are not supported by most annotation symbols.

The Revit Platform API also provides ArrayWithoutAssociate() methods to array one or several elements without being grouped and associated. This method is similar to the Array() method, but each element is independent of the others, and can be manipulated without affecting the other elements.

10.6 Delete

The Revit Platform API provides Delete() methods to delete one or more elements in the project.

Member	Description
Delete(Element)	Delete an element from the project.
Delete(ElementId)	Delete an element from the project using the element ID
Delete(ICollection<ElementId>)	Delete several elements from the project by their IDs.
Delete(ElementSet)	Delete several elements from the project.

Table 22: Delete Members

You can delete the element specified using the element object or the ElementId. These methods delete a specific element and any elements dependent on it.

Code Region 10-11: Deleting an element based on object

```
private void DeleteElement(Autodesk.Revit.DB.Document document, Element element)
{
    // Delete a selected element.
    ICollection<Autodesk.Revit.DB.ElementId> deletedIdSet = document.Delete(element);

    if (0 == deletedIdSet.Count)
    {
        throw new Exception("Deleting the selected element in Revit failed.");
    }

    String prompt = "The selected element has been removed and ";
    prompt += deletedIdSet.Count - 1;
    prompt += " more dependent elements have also been removed.";

    // Give the user some information
    TaskDialog.Show("Revit",prompt);
}
```

Code Region 10-12: Deleting an element based on ElementId

```
private void DeleteElement(Autodesk.Revit.DB.Document document, Element element)
{
    // Delete an element via its id
    Autodesk.Revit.DB.ElementId elementId = element.Id;
    ICollection<Autodesk.Revit.DB.ElementId> deletedIdSet = document.Delete(elementId);

    if (0 == deletedIdSet.Count)
    {
        throw new Exception("Deleting the selected element in Revit failed.");
    }

    String prompt = "The selected element has been removed and ";
    prompt += deletedIdSet.Count - 1;
    prompt += " more dependent elements have also been removed.";

    // Give the user some information
    TaskDialog.Show("Revit", prompt);
}
```

Note: When an element is deleted, any child elements associated with that element are also deleted, as indicated in the samples above.

The API also provides two ways to delete several elements.

Code Region 10-13: Deleting an ElementSet

```
// Delete all the selected elements via the set of elements
UIDocument uidoc = new UIDocument(document);
ElementSet elementSet = uidoc.Selection.Elements;
ICollection<Autodesk.Revit.DB.ElementId> deletedIdSet = document.Delete(elementSet);

if (0 == deletedIdSet.Count)
{
    throw new Exception("Deleting the selected elements in Revit failed.");
}

TaskDialog.Show("Revit","The selected element has been removed.");
```

Code Region 10-14: Deleting an ElementSet based on Id

```
// Delete all the selected elements via the set of element ids.
ICollection<Autodesk.Revit.DB.ElementId> idSelection = null ;
UIDocument uidoc = new UIDocument(document);
foreach (Autodesk.Revit.DB.Element elem in uidoc.Selection.Elements)
{
    Autodesk.Revit.DB.ElementId id = elem.Id;
    idSelection.Add(id);
}

ICollection<Autodesk.Revit.DB.ElementId> deletedIdSet = document.Delete(idSelection);

if (0 == deletedIdSet.Count)
{
    throw new Exception("Deleting the selected elements in Revit failed.");
}

TaskDialog.Show("Revit","The selected element has been removed.");
```

Note: After you delete the elements, any references to the deleted elements become invalid and throw an exception if they are accessed.

Some elements have special deletion methods. For example, the Materials.Remove() method is specifically for deleting Material objects. For more information, see the Material chapter Material Management section.

10.7 SuspendUpdating

When using automatic regeneration mode, SuspendUpdating is a class that temporarily delays automatic updating and consistency operations in Revit. For example, if you create an instance of this object, it suspends certain updating and consistency operations to increase large-scale change performance using the API. Note that this does not apply when using manual regeneration mode.

Updating continues when the SuspendUpdating object is explicitly disposed. SuspendUpdating works primarily with object transformation operations such as move and rotate. For a complete list of operations which this class suspends, see the Revit API Help file.

Code Region 10-15: Using SuspendUpdating

```
public void MoveElements(Autodesk.Revit.DB.Document document)
{
    UIDocument uidoc = new UIDocument(document);
    Selection choices = uidoc.Selection;
    ElementSet collection = choices.Elements;
    // Using statement defines a scope at the end of which the suspandUpdating
    // object is disposed.
    using (SuspendUpdating aSuspendUpdating = new SuspendUpdating(document))
    {
        // Suspend these operations.
        foreach (FamilyInstance element in collection)
        {
            MoveRotateElement(document, element);
        }
    }
    // Move() operations only update when aSuspendUpdating and
    // bSuspendUpdating (in MoveRotateElement()) are disposed
}
```

SuspendUpdating objects can be nested. The Nested property returns true if the object is already nested. During nesting, the final update happens only when all SuspendUpdating objects are disposed. For example, if the following function is called by the previous example, the bSuspendUpdating object Nested property returns true. In this situation, the Move() operation update only happens after both bSuspendUpdating and aSuspendUpdating objects are disposed.

Code Region 10-16: Nesting SuspendUpdating

```
private bool MoveRotateElement(Document document, Autodesk.Revit.DB.Element elem)
{
    // Operations suspended until SuspendUpdating object is disposed
    using (SuspendUpdating bSuspendUpdating = new SuspendUpdating(document))
    {
        // Move Element
        XYZ translation3D = new XYZ(0.0, 0.0, 0.0);
        bool successful1 = document.Move(elem, translation3D);

        // Rotate Element
        XYZ point1 = new XYZ(10, 20, 0);
        XYZ point2 = new XYZ(10, 20, 30);
        Line axis = document.Application.Create.NewLineUnbound(point1, point2);
        bool successful2 = document.Rotate(elem, axis, Math.PI / 3.0);

        return (successful1 && successful2);
    }
}
```

11 Walls, Floors, Roofs and Openings

This chapter discusses Elements and the corresponding ElementTypes representing built-in place construction:

- HostObject - The first two sections focus on HostObject and corresponding HostObjAttributes subclasses
- Foundation - Different foundations in the API are represented as different classes, including Floor, ContFooting, and FamilyInstance. The Floor and Foundation section compares them in the API.
- CompoundStructure - This section describes the HostObject.CompoundStructure property and provides tips to access Material.

In addition to host Elements, the Opening class is introduced at the end of this chapter.

11.1 Walls

There are four kinds of Walls represented by the WallType.WallKind enumeration:

- Stacked
- Curtain
- Basic
- Unknown

The Wall and WallType class work with the Basic wall type while providing limited function to the Stacked and Curtain walls. On occasion you need to check a Wall to determine the wall type. For example, you cannot get sub-walls from a Stacked Wall using the API. WallKind is read only and set by System Family.

The Wall.Flipped property and Wall.flip() method gain access to and control Wall orientation. In the following examples, a Wall is compared before and after calling the flip() method.

- The Orientation property before is (0.0, 1.0, 0.0).
- The Orientation property after the flip call is (0.0, -1.0, 0.0).
- The Wall Location Line (WALL_KEY_REF_PARAM) parameter is 3, which represents Finish Face: Interior in the following table.
- Taking the line as reference, the Wall is moved but the Location is not changed.

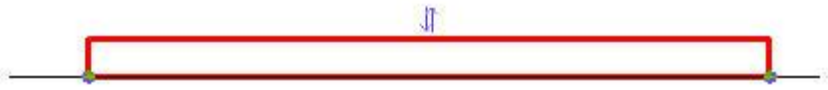


Figure 32: Original wall

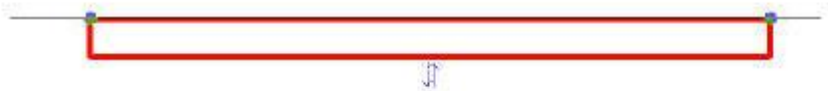


Figure 33: Wall after flip

Location Line Value	Description
0	Wall Centerline
1	Core Centerline
2	Finish Face: Exterior
3	Finish Face: Interior
4	Core Face: Exterior
5	Core Face: Interior

Table 23: Wall Location Line

There are five override methods in the Document class to create a Wall:

Name	Description
NewWall(Curve, WallType, Level, Double, Double, Boolean, Boolean)	Creates a new rectangular profile wall within the project.
NewWall(CurveArray, Boolean)	Creates a non rectangular profile wall within the project using the default wall style.
NewWall(Curve, Level, Boolean)	Creates a new rectangular profile wall within the project on the specified level using the default wall style.
NewWall(CurveArray, WallType, Level, Boolean)	Creates a non rectangular profile wall within the project.
NewWall(CurveArray, WallType, Level, Boolean, XYZ)	Creates a non rectangular profile wall within the project with a specific normal.

Table 24: NewWall() Overrides

The WallType Wall Function (WALL_ATTR_EXTERIOR) parameter influences the created wall instance Room Bounding and Structural Usage parameter. The WALL_ATTR_EXTERIOR value is an integer:

Wall Function	Interior	Exterior	Foundation	Retaining	Soffit
Value	0	1	2	3	4

Table 25: Wall Function

There are two override methods in the Document class to batch create multiple walls:

Name	Description
NewWalls(List(Of RectangularWallCreationData))	Creates rectangular walls within the project.
NewWalls(List(Of ProfiledWallCreationData))	Creates profile walls within the project.

Table 26: NewWalls() Overrides

The following rules apply to Walls created by the API:

- If the input structural parameter is true or the Wall Function (WALL_ATTR_EXTERIOR) parameter is Foundation, the Wall StructuralUsage parameter is Bearing; otherwise it is NonBearing.
- The created Wall Room Bounding (WALL_ATTR_ROOM_BOUNDING) parameter is false if the Wall Function (WALL_ATTR_EXTERIOR) parameter is Retaining.

For more information about structure-related functions such as the AnalyticalModel property, refer to the Revit Structure chapter.

11.2 Floors and Foundations

Floor and Foundation-related API items include:

Object	Element Type	ElementType Type	Element Creation	Other
Floor	Floor	FloorType	NewFloor()/ NewSlab()	StructuralUsage = InstanceUsage.Slab FloorType.IsFoundationSlab = false
Slab	Floor	FloorType	NewSlab()	StructuralUsage = InstanceUsage.Slab FloorType.IsFoundationSlab = false
Wall Foundation	ContFooting	ContFootingType	No	Category = OST_StructuralFoundation
Isolated Foundation	FamilyInstance	FamilySymbol	NewFamilyInstance()	Category = OST_StructuralFoundation
Foundation Slab	Floor	FloorType	NewFloor()	Category = OST_StructuralFoundation StructuralUsage = InstanceUsage.SlabFoundation FloorType.IsFoundationSlab = true

Table 27: Floors and Foundations in the API

The following rules apply to Floor:

- Elements created from the Foundation Design bar have the same category, OST_StructuralFoundation, but correspond to different Classes.
- The FloorType IsFoundationSlab property sets the FloorType category to OST_StructuralFoundation or not.

When you retrieve FloorType to create a Floor or Foundation Slab with NewFloor, use the following methods:

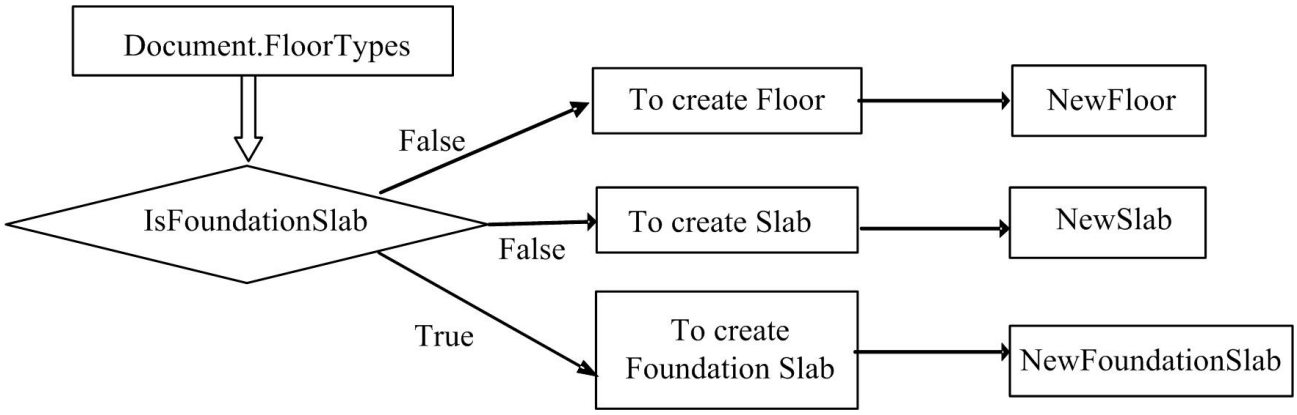


Figure 34: Create foundation and floor/slab

NewSlab() is an override NewFloor() method. Currently, the API does not provide access to the Floor Slope Arrow in the Floor class. However, in Revit Structure, you can create a sloped slab with NewSlab():

Code Region 11-1: NewSlab()

```
public Floor NewSlab(CurveArray profile, Level level, Line slopedArrow, double angle,
bool isStructural);
```

The Slope Arrow is created using the slopedArrow parameter.

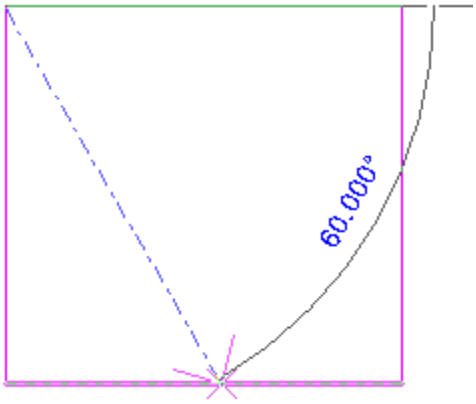


Figure 35: slopedArrow parameter in NewSlab

The Floor.FloorType property is an alternative to using the Floor.GetTypeId() method. For more information about structure-related functions such as SpanDirectionSymbols and SpanDirectionAngle, refer to the Revit Structure chapter.

When editing an Isolated Foundation in Revit, you can perform the following actions:

- You can pick a host, such as a floor. However, the FamilyInstance object Host property always returns null.
- When deleting the host floor, the Foundation is not deleted with it.
- The Foundation host is available from the Host (INSTANCE_FREE_HOST_PARAM) parameter.
- Use another related Offset (INSTANCE_FREE_HOST_OFFSET_PARAM) parameter to control the foundation offset from the host Element.

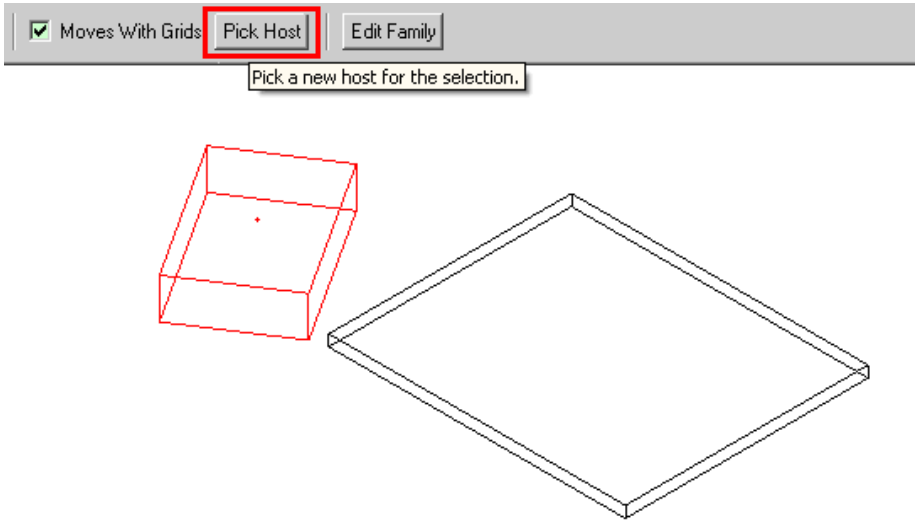


Figure 36: Pick Host for FoundationSlab (FamilyInstance)

Continuous footings are represented by the ContFooting class in the API. The API provides limited access to both ContFooting and ContFootingType except when using the GetAnalyticalModel() method (refer to the AnalyticalModel section in the Revit Structure chapter). For example, the attached wall is not available in Revit Architecture. In Revit Structure, the relationship between the Wall class and the ContFooting class is shown using the GetAnalyticalModelSupports() method in the AnalyticalModel class. For more details, refer to the AnalyticalModelSupport section in the Revit Structure chapter.

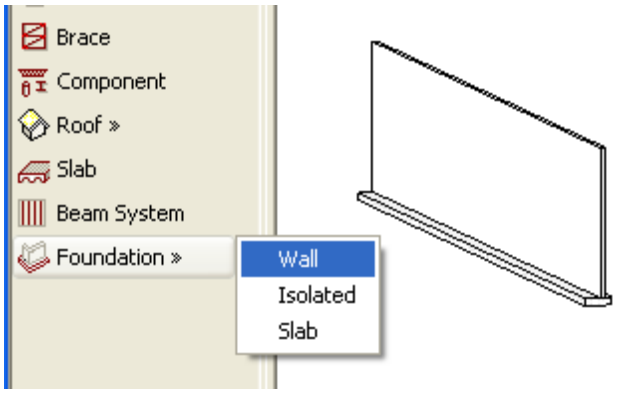


Figure 37: Wall ContFooting

11.2.1 Modifying Slabs

You can modify the form of slab-based elements using the SlabShapeEditor class. This class allows you to:

- Manipulate one or more of the points or edges on a selected slab-based element
- Add points on the element to change the element's geometry
- Add linear edges and split the existing face of a slab into smaller sub-regions
- Remove the shape modifier and reset the element geometry back to the unmodified shape.

Here's an example of reverting a selected modified floor back to its original shape:

Code Region 11-2: Reverting a slab's shape

```
private void ResetSlabShapes(Autodesk.Revit.DB.Document document)
{
    UIDocument uidoc = new UIDocument(document);
    Selection choices = uidoc.Selection;
    ElementSet collection = choices.Elements;
    foreach (Autodesk.Revit.DB.Element elem in collection)
    {
        Floor floor = elem as Floor;
        if (floor != null)
        {
            SlabShapeEditor slabShapeEditor = floor.SlabShapeEditor;
            slabShapeEditor.ResetSlabShape();
        }
    }
}
```

For more detailed examples of using the SlabShapeEditor and related classes, see the SlabShapeEditing sample application included in the Revit SDK.

11.3 Roofs

Roofs in the Revit Platform API all derive from the RoofBase object. There are two classes:

- FootPrintRoof – represents a roof made from a building footprint
- ExtrusionRoof – represents roof made from an extruded profile

Both have a RoofType property that gets or sets the type of roof. This example shows how you can create a footprint roof based on some selected walls:

Code Region 11-3: Creating a footprint roof

```
// Before invoking this sample, select some walls to add a roof over.
// Make sure there is a level named "Roof" in the document.

// find the Roof level
FilteredElementCollector collector = new FilteredElementCollector(document);
collector.WherePasses(new ElementClassFilter(typeof(Level)));
var elements = from element in collector where element.Name == "Roof" select element;
Level level = elements.Cast<Level>().ElementAt<Level>(0);

RoofType rooftype = null;
// select the first rooftype
foreach (RoofType rt in document.RoofTypes)
{
    rooftype = rt;
    break;
}

// Get the handle of the application
Autodesk.Revit.ApplicationServices.Application application = document.Application;

// Define the footprint for the roof based on user selection
CurveArray footprint = application.Create.NewCurveArray();
UIDocument uidoc = new UIDocument(document);
if (uidoc.Selection.Elements.Size != 0)
{
    foreach (Autodesk.Revit.DB.Element element in uidoc.Selection.Elements)
    {
        Wall wall = element as Wall;
        if (wall != null)
        {
            LocationCurve wallCurve = wall.Location as LocationCurve;
            footprint.Append(wallCurve.Curve);
            continue;
        }

        ModelCurve modelCurve = element as ModelCurve;
        if (modelCurve != null)
        {
            footprint.Append(modelCurve.GeometryCurve);
        }
    }
}
else
{
     throw new Exception("You should select a curve loop, or a wall loop, or loops
```

```
combination \nof walls and curves to create a footprint roof.");
}


ModelCurveArray footPrintToModelCurveMapping = new ModelCurveArray();
FootPrintRoof footprintRoof = document.Create.NewFootPrintRoof(footprint, level, rooftype,
out footPrintToModelCurveMapping);
ModelCurveArrayIterator iterator = footPrintToModelCurveMapping.ForwardIterator();
iterator.Reset();
while (iterator.MoveNext())
{
    ModelCurve modelCurve = iterator.Current as ModelCurve;
    footprintRoof.set_DefinesSlope(modelCurve, true);
    footprintRoof.set_SlopeAngle(modelCurve, 0.5);
}
```

For an example of how to create an ExtrusionRoof, see the NewRoof sample application included with the Revit API SDK.

11.3.1 Gutter and Fascia

Gutter and Fascia elements are derived from the HostedSweep class, which represents a roof. They can be created, deleted or modified via the API. To create these elements, use one of the Document.Create.NewFascia() or Document.Create.NewGutter() overrides. For an example of how to create new gutters and fascia, see the NewHostedSweep application included in the SDK samples. Below is a code snippet showing you can modify a gutter element's properties.

Code Region 11-4: Modifying a gutter

```
public void ModifyGutter(Autodesk.Revit.DB.Document document)
{
    UIDocument uidoc = new UIDocument(document);
    ElementSet collection = uidoc.Selection.Elements;

    foreach (Autodesk.Revit.DB.Element elem in collection)
    {
        if (elem is Gutter)
        {
            Gutter gutter = elem as Gutter;
            // convert degrees to rads:
            gutter.Angle = 45.00 * Math.PI / 180;
            TaskDialog.Show("Revit","Changed gutter angle");
        }
    }
}
```

11.4 Curtains

Curtain walls, curtain systems, and curtain roofs are host elements for CurtainGrid objects. A curtain wall can have only one CurtainGrid, while curtain systems and curtain roofs may contain one or more CurtainGrids. For an example of how to create a CurtainSystem, see the

CurtainSystem sample application included with the Revit SDK. For an example of creating a curtain wall and populating it with grid lines, see the CurtainWallGrid sample application.

11.5 Other Elements

The following Elements are not HostObjects (and don't have a specific class), but are special cases that can host other objects.

11.5.1 Stair and Ramp

Stair, Ramp, and the associated element types do not have specific classes in the API.

- Stair and its associated element type are represented as Element and ElementType in the OST_Stairs category.
- Ramp and its element type are represented as Element and ElementType in the OST_Ramps category.

11.5.2 Ceiling

Ceiling does not have a specific class, so you can't create it using the API. The Ceiling object is an Element in the OST_Ceilings category.

- The Ceiling element type is HostObjAttributes using the OST_Ceilings category with its CompoundStructure available.

11.6 CompoundStructure

Some host elements, such as Walls, Floors, Ceilings, and Roofs, include parallel layers. The parallel layers are available from the HostObjAttributes.CompoundStructure property.

Note: The following items are important when using CompoundStructure:

- The total thickness of the element is the sum of each CompoundStructureLayer's thickness. You cannot change the element's total thickness directly but you can change it via changing the CompoundStructureLayer thickness.
- CompoundStructure is a property of HostObjAttributes. Changing the CompoundStructureLayer changes every element instance in the current document.
- The CompoundStructureLayerArray ReadOnly property retrieved from the element CompoundStructure is always true. You cannot insert, remove, or reorder CompoundStructureLayers within the array.
- The CompoundStructureLayer DeckProfile, DeckUsage, and Variable properties only work with Slab in Revit Structure. For more details, refer to the Revit Structure chapter.

11.6.1 Material

Each CompoundStructureLayer in HostObjAttributes is typically displayed with some type of material. If CompoundStructureLayer.Material returns null, it does not mean the layer does not have a Material; it means the Material is Category-related. For more details, refer to the Material chapter. Getting the CompoundStructureLayer Material is illustrated in the following sample code:

Code Region 11-5: Getting the CompoundStructureLayer Material

```
public void GetWallLayerMaterial(Autodesk.Revit.DB.Document document, Wall wall)
{
    // get WallType of wall
    WallType aWallType = wall.WallType;
    // Only Basic Wall has compoundStructure
```

```
    if (WallKind.Basic == aWallType.Kind)
    {
        // Get CompoundStructure
        CompoundStructure comStruct = aWallType.CompoundStructure;
        Categories allCategories = document.Settings.Categories;

        // Get the category OST_Walls default Material;
        // use if that layer's default Material is <By Category>
        Category wallCategory = allCategories.get_Item(BuiltInCategory.OST_Walls);
        Autodesk.Revit.DB.Material wallMaterial = wallCategory.Material;

        foreach (CompoundStructureLayer structLayer in comStruct.Layers)
        {
            Autodesk.Revit.DB.Material layerMaterial = structLayer.Material;
            // If CompoundStructureLayer's Material is specified, use default
            // Material of its Category
            if (null == layerMaterial)
            {
                switch (structLayer.Function)
                {
                    case CompoundStructureLayerFunction.Finish1:
                        layerMaterial =
                           allCategories.get_Item(BuiltInCategory.OST_WallsFinish1).Material;
                        break;
                    case CompoundStructureLayerFunction.Finish2:
                        layerMaterial =
                           allCategories.get_Item(BuiltInCategory.OST_WallsFinish2).Material;
                        break;
                    case CompoundStructureLayerFunction.MembraneLayer:
                        layerMaterial =
                          allCategories.get_Item(BuiltInCategory.OST_WallsMembrane).Material;
                        break;
                    case CompoundStructureLayerFunction.Structure:
                        layerMaterial =
                         allCategories.get_Item(BuiltInCategory.OST_WallsStructure).Material;
                        break;
                    case CompoundStructureLayerFunction.Substrate:
                        layerMaterial =
                         allCategories.get_Item(BuiltInCategory.OST_WallsSubstrate).Material;
                        break;
                    case CompoundStructureLayerFunction.ThermalOrAir:
                        layerMaterial =
                        allCategories.get_Item(BuiltInCategory.OST_WallsInsulation).Material;
                        break;
                    default:
                        // It is impossible to reach here
                        break;
                }
```

```
                if (null == layerMaterial)
                {
                    // CompoundStructureLayer's default Material is its SubCategory
                    layerMaterial = wallMaterial;
                }
            }
            TaskDialog.Show("Revit","Layer Material: " + layerMaterial);
        }
    }
}
```

11.7 Opening

In the Revit Platform API, the Opening object is derived from the Element object and contains all of the Element object properties and methods. To retrieve all Openings in a project, use Document.ElementIterator to find the Elements.Opening objects.

11.7.1 General Properties

This section explains how to use the Opening properties.

- IsRectBoundary - Identifies whether the opening has a rectangular boundary.
  - If true, it means the Opening has a rectangular boundary and you can get an IList<XYZ> collection from the Opening BoundaryRect property. Otherwise, the property returns null.
  - If false, you can get a CurveArray object from the BoundaryCurves property.
- BoundaryCurves - If the opening boundary is not a rectangle, this property retrieves geometry information; otherwise it returns null. The property returns a CurveArray object containing the curves that represent the Opening object boundary.

  For more details about Curve, refer to the Geometry chapter.
- BoundaryRect - If the opening boundary is a rectangle, you can get the geometry information using this property; otherwise it returns null.
  - The property returns an IList<XYZ> collection containing the XYZ coordinates.
  - The IList<XYZ> collection usually contains the rectangle boundary minimum (lower left) and the maximum (upper right) coordinates.
- Host - The host property retrieves the Opening host element. The host element is the element cut by the Opening object.

  Note: If the Opening object's category is Shaft Openings, the Opening host is null.

The following example illustrates how to retrieve the existing Opening properties.

Code Region 11-6: Retrieving existing opening properties

```
private void Getinfo_Opening(Opening opening)
{
    string message = "Opening:";
    //get the host element of this opening
    message += "\nThe id of the opening's host element is : " + opening.Host.Id.Value;

    //get the information whether the opening has a rect boundary
    //If the opening has a rect boundary, get the geom info from BoundaryRect property.
    //Otherwise we should get the geometry information from BoundaryCurves property
    if (opening.IsRectBoundary)
    {
        message += "\nThe opening has a rectangular boundary.";
        //array contains two XYZ objects: the max and min coords of boundary
        XYZArray boundaryRect = opening.BoundaryRect;

        //get the coordinate value of the min coordinate point
        XYZ point = opening.BoundaryRect.get_Item(0);
        message += "\nMin coordinate point:(" + point.X + ", "
                                + point.Y + ", " + point.Z + ")";
        //get the coordinate value of the Max coordinate point
        point = opening.BoundaryRect.get_Item(1);
        message += "\nMax coordinate point: (" + point.X + ", "
                                + point.Y + ", " + point.Z + ")";
    }
    else
    {
        message += "\nThe opening doesn't have a rectangular boundary.";
        // Get curve number
        int curves = opening.BoundaryCurves.Size;
        message += "\nNumber of curves is : " + curves;
        for (int i = 0; i < curves; i++)
        {
            Autodesk.Revit.Geometry.Curve curve = opening.BoundaryCurves.get_Item(i);
            // Get curve start and end points
            message += "\nCurve start point: " + XYZToString(curve.get_EndPoint(0));
            message += "; Curve end point: " + XYZToString(curve.get_EndPoint(1));
        }
    }
    MessageBox.Show(message, "Revit");
}
// output the point's three coordinates
string XYZToString(XYZ point)
{
    return "(" + point.X + ", " + point.Y + ", " + point.Z + ")";
}
```

11.7.2 Create Opening

In the Revit Platform API, use the Document.NewOpening() method to create an opening in your project. There are four method overloads you can use to create openings in different host elements:

Code Region 11-7: NewOpening()

```
//Create a new Opening in a beam, brace and column.
public Opening NewOpening(Element famInstElement, CurveArray profile, eRefFace iFace);
```

```
//Create a new Opening in a roof, floor and ceiling.
public Opening NewOpening(Element hostElement, CurveArray profile, bool bPerpendicularFace);
```

```
//Create a new Opening Element.
public Opening NewOpening(Level bottomLevel, Level topLevel, CurveArray  profile);
```

```
//Create an opening in a straight wall or arc wall.
public Opening NewOpening(Wall, XYZ pntStart, XYZ pntEnd);
```

- Create an Opening in a Beam, Brace, or Column - Use to create an opening in a family instance. The iFace parameter indicates the face on which the opening is placed.
- Create a Roof, Floor, or Ceiling Opening - Use to create an opening in a roof, floor, or ceiling.
- The bPerpendicularFace parameter indicates whether the opening is perpendicular to the face or vertical.
- If the parameter is true, the opening is perpendicular to the host element face. See the following picture:

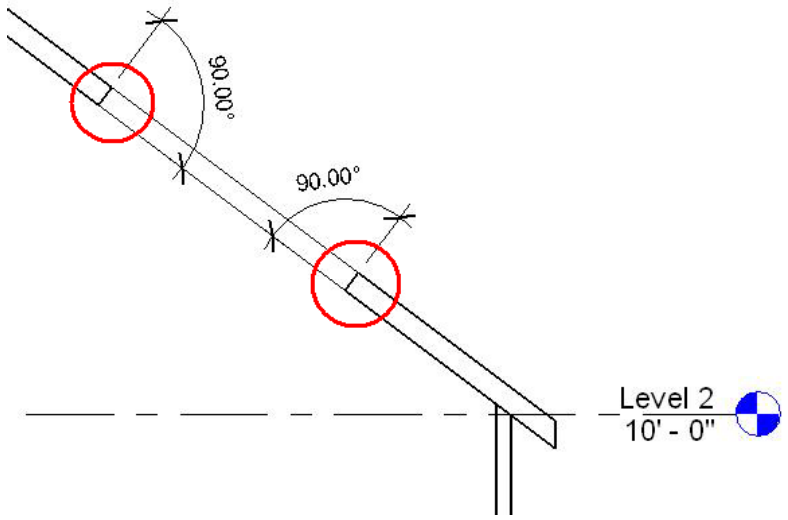


Figure 38: Opening cut perpendicular to the host element face

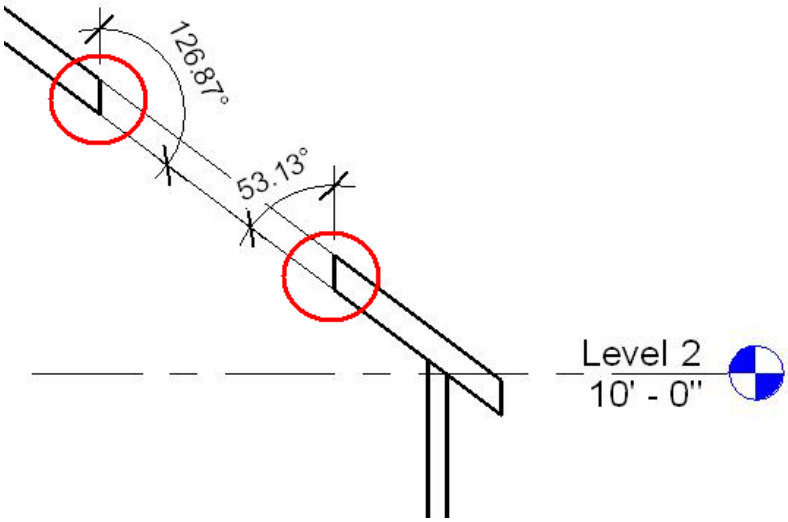


Figure 39: Opening cut vertically to the host element

- Create a New Opening Element - Use to create a shaft opening in your project. However, make sure the topLevel is higher than the bottomLevel; otherwise an exception is thrown.
- Create an Opening in a Straight Wall or Arc Wall - Use to create a rectangle opening in a wall. The coordinates of pntStart and pntEnd should be corner coordinates that can shape a rectangle. For example, the lower left corner and upper right corner of a rectangle. Otherwise an exception is thrown.

Note: Using the Opening command you can only create a rectangle shaped wall opening. To create some holes in a wall, edit the wall profile instead of the Opening command.

12 Family Instances

In this chapter, you will learn about the following:

- The relationship between family and family instance
- Family and family instance features
- How to load or create family and family instance features.

12.1 Identifying Elements

In Revit, the easiest way to judge whether an element is a FamilyInstance or not is by using the properties dialog box.

- If the family name starts with System Family and the Load button is disabled, it belongs to System Family.

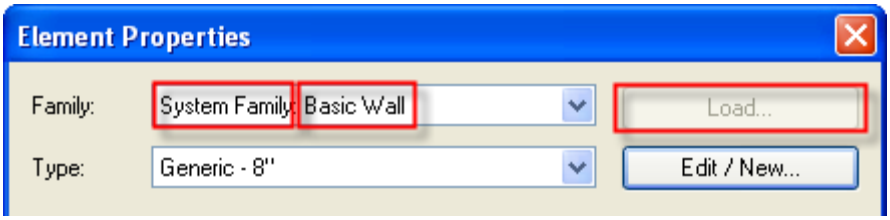


Figure 40: System Family

- A general FamilyInstance, which belongs to the Component Family, does not start with System Family.
- For example, in the following picture the family name for the desk furniture is Desk. In addition, the Load button is enabled.

Figure 41: Component Family

- There are some exceptions, for example: Mass and in-place member. The Family and Type fields are blank.

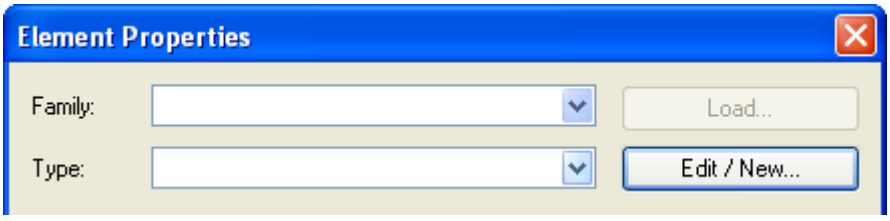


Figure 42: Mass or in-place member example

Families in the Revit Platform API are represented by three objects:

- Family
- FamilySymbol
- FamilyInstance

Each object plays a significant role in the family structure.

The Family object represents an entire family such as Single-Flush doors. For example, the Single-Flush door Family corresponds to the Single-Flush.rfa file. The Family object contains several FamilySymbols that are used to get all family symbols to facilitate swapping instances from one symbol to another.

The FamilySymbol object represents a specific set of family settings corresponding to a Type in the Revit UI, such as 34”X80”.

The FamilyInstance object represents an actual Type (FamilySymbol) instance in the Revit project. For example，in the following picture, the FamilyInstance is a single door in the project.

- Each FamilyInstance has one FamilySymbol. The door is an instance of a 34”X80”.
- Each FamilySymbol belongs to one Family. The 34”X80” symbol belongs to a Single-Flush family.
- Each Family contains one or more FamilySymbols. The Single-Flush family contains a 34”X80” symbol, a 34”X84” symbol, a 36”X84” and so on.

Note that while most component elements are exposed through the API classes FamilySymbol and FamilyInstance, some have been wrapped with specific API classes. For example, AnnotationSymbolType wraps FamilySymbol and AnnotationSymbol wraps FamilyInstance.

12.2 Family

The Family class represents an entire Revit family. It contains the FamilySymbols used by FamilyInstances.

12.2.1 Loading Families

The Document class contains the LoadFamily() and LoadFamilySymbol() methods.

- LoadFamily() loads an entire family and all of its types or symbols into the project.
- LoadFamilySymbol() loads only the specified family symbol from a family file into the project.

Note: To improve the performance of your application and reduce memory usage, if possible load specific FamilySymbols instead of entire Family objects.

- The family file path is retrieved using the Options.Application object LibraryPaths property.
- The Options.Application object is retrieved using the Application object Options property.
- In LoadFamilySymbol(), the input argument Name is the same string value returned by the FamilySymbol object Name property.

For more information, refer to the Code Samples in this chapter.

12.2.2 Categories

The FamilyBase.FamilyCategory property indicates the category of the Family such as Columns, Furniture, Structural Framing, or Windows.

12.3 FamilyInstances

Examples of categories of FamilyInstance objects in Revit are Beams, Braces, Columns, Furniture, Massing, and so on. The FamilyInstance object provides more detailed properties so that the family instance type and appearance in the project can be changed.

12.3.1 Location-Related Properties

Location-related properties show the physical and geometric characteristics of FamilyInstance objects, such as orientation, rotation and location.

12.3.1.1 Orientation

The face orientation or hand orientation can be changed for some FamilyInstance objects. For example, a door can face the outside or the inside of a room or wall and it can be placed with the handle on the left side or the right side. The following table compares door, window, and desk family instances.

Boolean Property	Door	Window (Fixed: 36”w x 72”h)	Desk
CanFlipFacing	True	True	False
CanFlipHand	True	False	False

Table 28: Compare Family Instances

If CanFlipFacing or CanFlipHand is true, you can call the flipFacing() or flipHand() methods respectively. These methods can change the facing orientation or hand orientation respectively. Otherwise, the methods do nothing and return False.

When changing orientation, remember that some types of windows can change both hand orientation and facing orientation, such as a Casement 3x3 with Trim family.

There are four different facing orientation and hand orientation combinations for doors. See the following picture for the combinations and the corresponding Boolean values are in the following table.

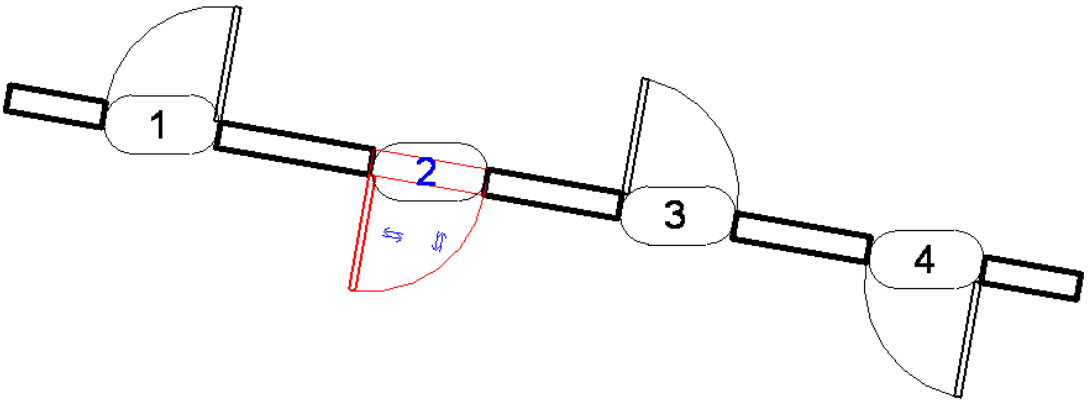


Figure 43: Doors with different Facing and Hand Orientations

Boolean Property	Door 1	Door 2	Door 3	Door 4

Boolean Property	Door 1	Door 2	Door 3	Door 4
FacingFlipped	False	True	False	True
HandFlipped	False	True	True	False

Table 29: Different Instances of the Same Type

12.3.1.2**Rotation -** Mirrored

The Mirrored property indicates whether the FamilyInstance object has been mirrored.

Boolean Property	Door 1	Door 2	Door 3	Door 4
Mirrored	False	False	True	True

Table 30: Door Mirrored Property

In the previous door example, the Mirrored property for Door 1 and Door 2 is False, while for both Door 3 and Door 4 it is True. This is because when you create a door in the Revit project, the default result is either Door 1 or Door 2. To create a door like Door 3 or Door 4, you must flip the Door 1 and Door 2 hand orientation respectively. The flip operation is like a mirror transformation, which is why the Door 3 and Door 4 Mirrored property is True.

For more information about using the Mirror() method in Revit, refer to the Editing Elements chapter.

12.3.1.3**Rotation – CanRotate and rotate()**

The family instance Boolean CanRotate property is used to test whether the family instance can be rotated 180 degrees. This depends on the family to which the instance belongs. For example, in the following picture, the CanRotate properties for Window 1 (Casement 3x3 with Trim: 36”x72”) and Door 1 (Double-Glass 2: 72”x82”) are true, while Window 2 (Fixed: 36”w x 72”h) is false.

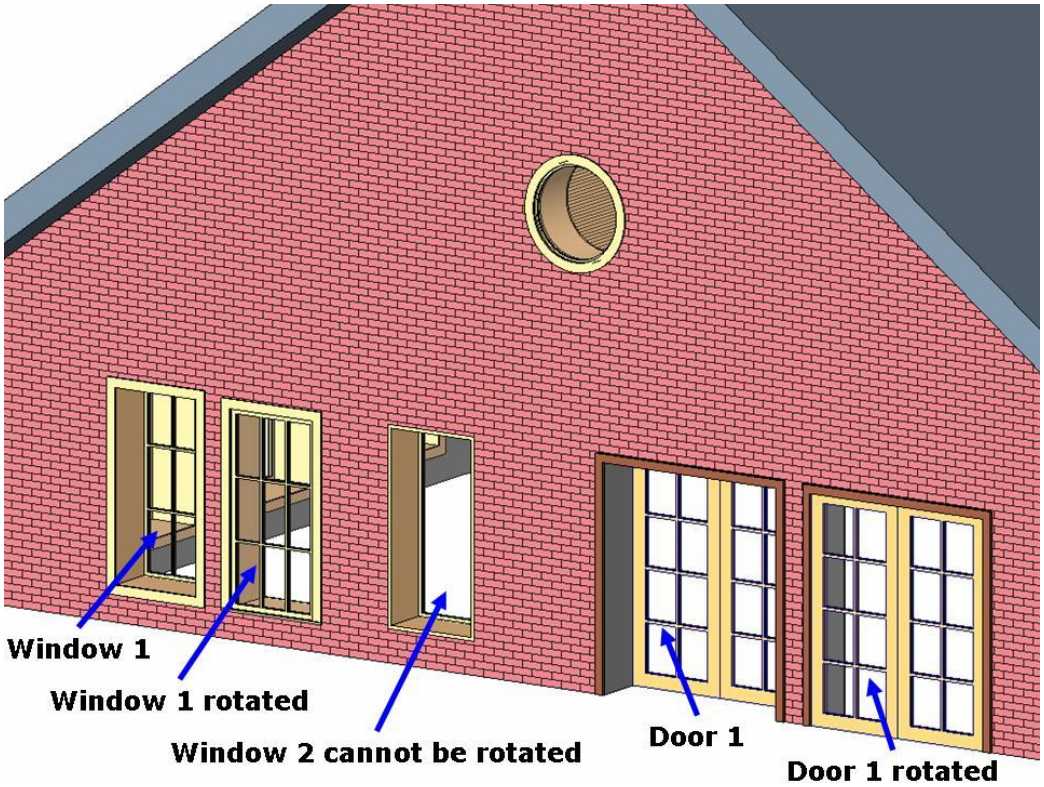


Figure 44: Changes after rotate()

If CanRotate is true, you can call the family instance rotate() method, which flips the family instance by 180 degrees. Otherwise, the method does nothing and returns False. The previous picture also shows the Window 1 and Door 1 states after executing the rotate() method.

Recall from the Rotating elements section earlier in this document, that family instances (and other elements) can be rotated a user-specified angle using Document.Rotate().

12.3.1.4**Location**

The Location property determines the physical location of an instance in a project. An instance can have a point location or a line location.

The following characteristics apply to Location:

- A point location is a LocationPoint class object - A footing, a door, or a table has a point location
- A line location is a LocationCurve class object - A beam has a line location.
- They are both subclasses of the Location class.

For more information about Location, refer to the Editing Elements chapter.

12.3.2**Host and Component**

Host and Component are both FamilyInstance properties. Component can be further divided into Subcomponent and Supercomponent.

12.3.2.1**Host**

A FamilyInstance object has a Host property that returns its hosting element.

Some FamilyInstance objects do not have host elements, such as Tables and other furniture, so the Host property returns nothing because no host elements are created. However, other objects, such as doors and windows, must have host elements. In this case the Host property returns a wall Element in which the window or the door is located. See the following picture.

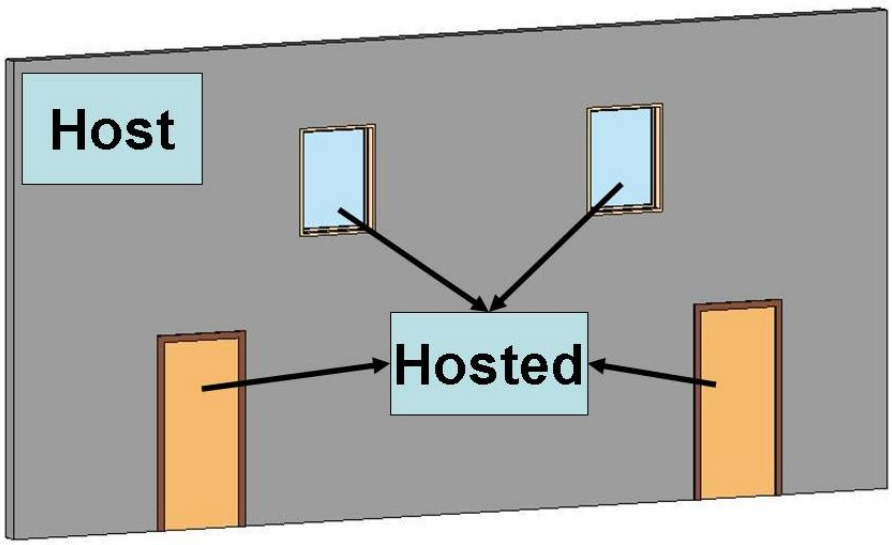


Figure 45: Doors and windows hosted in a wall

12.3.3 Subcomponent and Supercomponent

The FamilyInstance.SubComponents property returns the set of family instances loaded into that family. When an instance of 'Table-Dining Round w Chairs.rfa' is placed in a project, the set of chairs are returned by the SubComponent property.

The SuperComponent property returns the family instance's parent component. In 'Table-Dining Round w Chairs.rfa', the family instance supercomponent for each nested chair is the instance of 'Table-Dining Round w Chairs.rfa'.

Code Region 12-1: Getting SubComponents and SuperComponent from FamilyInstance

```
public void GetSubAndSuperComponents(FamilyInstance familyInstance)
{
    ElementSet subElemSet = familyInstance.SubComponents;
    if (subElemSet != null)
    {
        string subElems = "";
        foreach (Autodesk.Revit.Element ee in subElemSet)
        {
            FamilyInstance f = ee as FamilyInstance;
            subElems = subElems + f.Name + "\n";
        }
        MessageBox.Show("Subcomponent count = " + subElemSet.Size + "\n" + subElems);
    }
    FamilyInstance super = familyInstance.SuperComponent as FamilyInstance;
    if (super != null)
    {
        MessageBox.Show("SUPER component: " + super.Name);
    }
}
```

12.3.4 Other Properties

The properties in this section are specific to Revit Architecture and Revit Structure. They are covered thoroughly in their respective chapters.

12.3.4.1 Room Information

FamilyInstance properties include Room, FromRoom, and ToRoom. For more information about Room, refer to the Revit Architecture chapter.

12.3.4.2 Space Information

FamilyInstance has a Space property for identifying the space that holds an instance in MEP.

12.3.4.3 Revit Structure Related Analytical Model

The GetAnalyticalModel() method retrieves the family instance structural analytical model.

For more information about AnalyticalModel refer to the Revit Structure chapter.

12.3.5 Creating FamilyInstance Objects

Typically a FamilyInstance object is created using one of the eight overload methods of Autodesk.Revit.Creation.Document called NewFamilyInstance(). The choice of which overload to use depends not only on the category of the instance, but also other characteristics of the placement like whether it should be hosted, placed relative to a reference level, or placed directly on a particular face. The details are included in Table 31 - Options for creating instance with NewFamilyInstance() below.

Some FamilyInstance objects require more than one location to be created. In these cases, it is more appropriate to use the more detailed creation method provided by this object (see Table 32 - Options for creating instances with other methods below). If the instance is not created, an exception is thrown. The type/symbol used must be loaded into the project before the method is called.

Category	NewFamilyInstance() parameters	Comments
Air Terminals Boundary Conditions Casework Communication Devices Data Devices Electrical Equipment Electrical Fixtures Entourage Fire Alarm Devices Furniture Furniture Systems Generic Models Lighting Devices	XYZ, FamilySymbol, StructuralType	Creates the instance in an arbitrary location without reference to a level or host element.
	XYZ, FamilySymbol, Element, StructuralType	If it is to be hosted on a wall, floor or ceiling
	XYZ, FamilySymbol, XYZ, Element, StructuralType	If it is to be hosted on a wall, floor, or ceiling, and needs to be oriented in a non-default direction
	XYZ, FamilySymbol, Element, Level, StructuralType	If it is to be hosted on a wall, floor or ceiling and associated to a reference level

Lighting Fixtures Mass Mechanical Equipment Nurse Call Devices Parking Planting Plumbing Fixtures Security Devices Site Specialty Equipment Sprinklers Structural Connections Structural Foundations Structural Stiffeners Telephone Devices	XYZ, FamilySymbol, Level, StructuralType	If it is to be associated to a reference level
	Face, XYZ, XYZ, FamilySymbol	If it is face-based and needs to be oriented in a non-default direction
	Face, Line, FamilySymbol	If it is face-based and linear
Columns Structural Columns	XYZ, FamilySymbol, Level, StructuralType	Creates the column so that its base lies on the reference level. The column will extend to the next available level in the model, or will extend the default column height if there are no suitable levels above the reference level.
Doors Windows	XYZ, FamilySymbol, Element, StructuralType	Doors and windows must be hosted by a wall. Use this method if they can be placed with the default orientation.
	XYZ, FamilySymbol, XYZ, Element, StructuralType	If the created instance needs to be oriented in a non-default direction
	XYZ, FamilySymbol, Element, Level, StructuralType	If the instance needs to be associated to a reference level
Structural Framing (Beams, Braces)	Curve, FamilySymbol, Level, StructuralType	Creates a level based brace or beam given its curve. This is the recommended method to create Beams and Braces
	XYZ, FamilySymbol, StructuralType	Creates instance in an arbitrary location[1]
	XYZ, FamilySymbol, Element, Level, StructuralType	If it is hosted on an element (floor etc.) and associated to a reference level [1]
	XYZ, FamilySymbol, Level, StructuralType	If it is associated to a reference level [1]

	XYZ, FamilySymbol, Element, StructuralType	If it is hosted on an element (floor etc.) [1]

Table 31 - Options for creating instance with NewFamilyInstance()

[1] The structural instance will be of zero-length after creation. Extend it by setting its curve (FamilyInstance.Location as LocationCurve) using LocationCurve.Curve property.

You can simplify your code and improve performance by creating more than one family instance at a time using Document.NewFamilyInstances(). This method has a single parameter, which is a list of FamilyInstanceCreationData objects describing the family instances to create.

Code Region 12-2: Batch creating family instances

```
ElementSet BatchCreateColumns(Autodesk.Revit.DB.Document document, Level level)
{
    List<FamilyInstanceCreationData> fiCreationDatas =
        new List<FamilyInstanceCreationData>();

    ElementSet elementSet = null;

    //Try to get a FamilySymbol
    FamilySymbol familySymbol = null;
    FilteredElementCollector collector = new FilteredElementCollector(document);
    ICollection<Element> collection = collector.OfClass(typeof(FamilySymbol)).ToElements();
    foreach (Element e in collection)
    {
        familySymbol = e as FamilySymbol;
        if (null != familySymbol.Category)
        {
            if ("Structural Columns" == familySymbol.Category.Name)
            {
                break;
            }
        }
    }

    if (null != familySymbol)
    {
        //Create 10 FamilyInstanceCreationData items for batch creation
        for (int i = 1; i < 11; i++)
        {
            XYZ location = new XYZ(i * 10, 100, 0);
            FamilyInstanceCreationData fiCreationData =
                new FamilyInstanceCreationData(location, familySymbol, level,
                    StructuralType.Column);

            if (null != fiCreationData)
            {
```

```
                fiCreationDatas.Add(fiCreationData);
            }
        }

        if (fiCreationDatas.Count > 0)
        {
            // Create Columns
            elementSet = document.Create.NewFamilyInstances(fiCreationDatas);
        }
        else
        {
            throw new Exception("Batch creation failed.");
        }
    }
    else
    {
        throw new Exception("No column types found.");
    }

    return elementSet;
}
```

Instances of some family types are better created through methods other than Autodesk.Revit.Creation.Document.NewFamilyInstance(). These are listed in the table below.

Category	Creation method	Comments
Air Terminal Tags Area Load Tags Area Tags Casework Tags Ceiling Tags Communication Device Tags Curtain Panel Tags Data Device Tags Detail Item Tags Door Tags Duct Accessory Tags Duct Fitting Tags Duct Tags Electrical Equipment Tags Electrical Fixture Tags Fire Alarm Device Tags	NewTag(View, Element, Boolean, TagMode, TagOrientation, XYZ)	TagMode should be TM_ADDBY_CATEGORY and there should be a related tag family loaded when try to create a tag, otherwise exception will be thrown

Family Instances

Flex Duct Tags		
Flex Pipe Tags		
Floor Tags		
Furniture System Tags		
Furniture Tags		
Generic Model Tags		
Internal Area Load Tags		
Internal Line Load Tags		
Internal Point Load Tags		
Keynote Tags		
Lighting Device Tags		
Lighting Fixture Tags		
Line Load Tags		
Mass Floor Tags		
Mass Tags		
Mechanical Equipment Tags		
Nurse Call Device Tags		
Parking Tags		
Pipe Accessory Tags		
Pipe Fitting Tags		
Pipe Tags		
Planting Tags		
Plumbing Fixture Tags		
Point Load Tags		
Property Line Segment Tags		
Property Tags		
Railing Tags		
Revision Cloud Tags		
Roof Tags		
Room Tags		
Security Device Tags		
Site Tags		
Space Tags		
Specialty Equipment Tags		
Spinkler Tags		
Stair Tags		
Structural Area		

Reinforcement Tags Structural Beam System Tags Structural Column Tags Structural Connection Tags Structural Foundation Tags Structural Framing Tags Structural Path Reinforcement Tags Structural Rebar Tags Structural Stiffener Tags Structural Truss Tags Telephone Device Tags Wall Tags Window Tags Wire Tag Zone Tags		
Material Tags	NewTag(View, Element, Boolean, TagMode, TagOrientation, XYZ)	TagMode should be TM_ADDBY_MATERIAL and there should be a material tag family loaded, otherwise exception will be thrown
Multi-Category Tags	NewTag(View, Element, Boolean, TagMode, TagOrientation, XYZ)	TagMode should be TM_ADDBY_MULTICATEGORY, and there should be a multi-category tag family loaded, otherwise exception will be thrown
Generic Annotations	NewAnnotationSymbol(XYZ, AnnotationSymbolType, View)	
Title Blocks	NewViewSheet(FamilySymbol)	The titleblock will be added to the newly created sheet.

Table 32 - Options for creating instances with other methods

Families and family symbols are loaded using the Document.LoadFamily() or Document.LoadFamilySymbol() methods. Some families, such as Beams, have more than one endpoint and are inserted in the same way as a single point instance. Once the linear family instances are inserted, their endpoints can be changed using the Element.Location property. For more information, refer to the Code Samples section in this chapter.

12.4 Code Samples

Review the following code samples for more information about working with Family Instances. Please note that in the NewFamilyInstance() method, a StructuralType argument is required to specify the type of the family instance to be created. Here are some examples:

Type of Family Instance	Value of StructuralType
Doors, tables, etc.	NonStructural
Beams	Beam
Braces	Brace
Columns	Column
Footings	Footing

Table 33: The value of StructuralType argument in the NewFamilyInstance() method

12.4.1 Create Tables

The following function demonstrates how to load a family of Tables into a Revit project and create instances from all symbols in this family.

The LoadFamily() method returns false if the specified family was previously loaded. Therefore, in the following case, do not load the family, Table-Dining Round w Chairs.rfa, before this function is called. In this example, the tables are created at Level 1 by default.

Code Region 12-3: Creating tables

```
String fileName = @"C:\Documents and Settings\All Users\Application Data\Autodesk\RST
2011\Imperial Library\Furniture\Table-Dining Round w Chairs.rfa";

// try to load family
Family family = null;
if (!document.LoadFamily(fileName, out family))
{
    throw new Exception("Unable to load " + fileName);
}

// Loop through table symbols and add a new table for each
FamilySymbolSetIterator symbolItor = family.Symbols.ForwardIterator();
double x = 0.0, y = 0.0;
while (symbolItor.MoveNext())
{
    FamilySymbol symbol = symbolItor.Current as FamilySymbol;
    XYZ location = new XYZ(x, y, 10.0);
    // Do not use the overloaded NewFamilyInstance() method that contains
    // the Level argument, otherwise Revit cannot show the instances
    // correctly in 3D View, for the table is not level-based component.
    FamilyInstance instance = document.Create.NewFamilyInstance(location, symbol,
                                                    StructuralType.NonStructural);
    x += 10.0;
}
```

The result of loading the Tables family and placing one instance of each FamilySymbol:

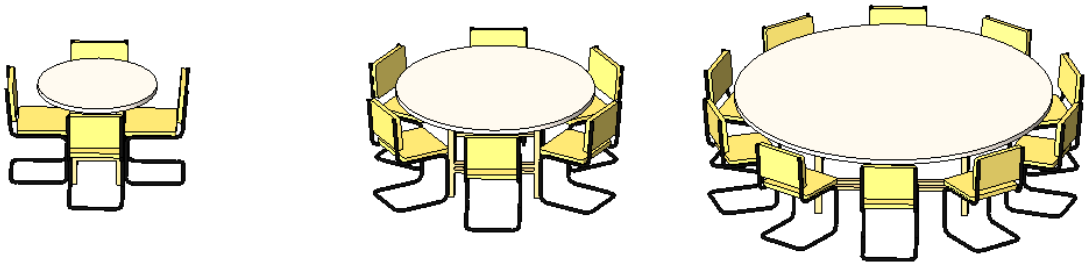


Figure 46: Load family and create tables in the Revit project

12.4.2 Create a Beam

In this sample, a family symbol is loaded instead of a family, because loading a single FamilySymbol is faster than loading a Family that contains many FamilySymbols.

Code Region 12-4: Creating a beam

```
// get the active view's level for beam creation
Level level = document.ActiveView.Level;

// load a family symbol from file
FamilySymbol gotSymbol = null;
String fileName = @"C:\Documents and Settings\All Users\Application Data\Autodesk\RST 2011\Imperial Library\Structural\Framing\Steel\W-Wide Flange.rfa";
String name = "W10X54";

FamilyInstance instance = null;

if (document.LoadFamilySymbol(fileName, name, out gotSymbol))
{
    // look for a model line in the list of selected elements
    UIDocument uidoc = new UIDocument(document);
    ElementSet sel = uidoc.Selection.Elements;
    ModelLine modelLine = null;
    foreach (Autodesk.Revit.DB.Element elem in sel)
    {
        if (elem is ModelLine)
        {
            modelLine = elem as ModelLine;
            break;
        }
    }
    if (null != modelLine)
    {
        // create a new beam
        instance = document.Create.NewFamilyInstance(modelLine.GeometryCurve, gotSymbol,
                                                     level, StructuralType.Beam);
    }
    else
    {
        throw new Exception("Please select a model line before invoking this command");
```

```
    }


}
else
{
    throw new Exception("Couldn't load " + fileName);
}
```

12.4.3 **Create Doors**

Create a long wall about 180’ in length and select it before running this sample. The host object must support inserting instances; otherwise the NewFamilyInstance() method will fail. If a host element is not provided for an instance that must be created in a host, or the instance cannot be inserted into the specified host element, the method NewFamilyInstance() does nothing.

Code Region 12-5: Creating doors

```
void CreateDoorsInWall(Autodesk.Revit.DB.Document document, Wall wall)
{
    String fileName = @"C:\Documents and Settings\All Users\Application Data\Autodesk\RST 2011\Imperial Library\Doors\Single-Decorative 2.rfa";


    Family family = null;
    if (!document.LoadFamily(fileName, out family))
    {
        throw new Exception("Unable to load " + fileName);
    }

    // get the active view's level for beam creation
    Level level = document.ActiveView.Level;

    FamilySymbolSetIterator symbolItor = family.Symbols.ForwardIterator();
    double x = 0, y = 0, z = 0;
    while (symbolItor.MoveNext())
    {
        FamilySymbol symbol = symbolItor.Current as FamilySymbol;
        XYZ location = new XYZ(x, y, z);
        FamilyInstance instance = document.Create.NewFamilyInstance(location, symbol,
            wall, level, StructuralType.NonStructural);
        x += 10;
        y += 10;
        z += 1.5;
    }
}
```

The result of the previous code in Revit is shown in the following picture. Notice that if the specified location is not at the specified level, the NewFamilyInstance() method uses the location elevation instead of the level elevation.

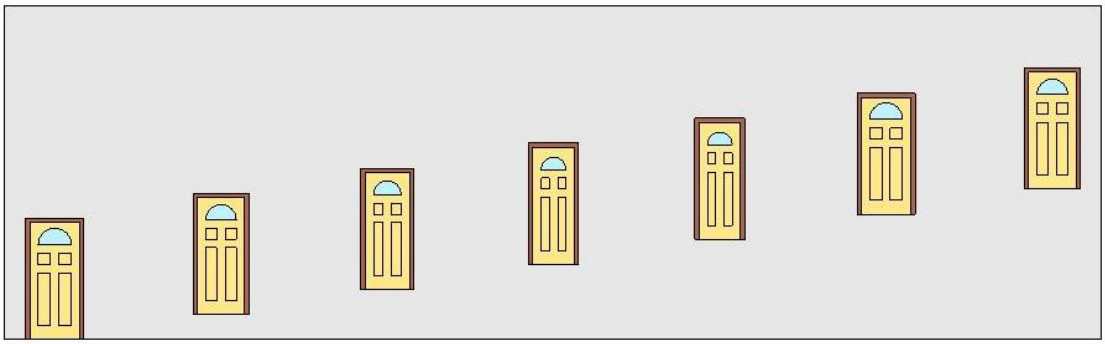


Figure 47: Insert doors into a wall

12.4.4 Create FamilyInstances Using Reference Directions

Use reference directions to insert an item in a specific direction.

Code Region 12-6: Creating FamilyInstances using reference directions

```
String fileName = @"C:\Documents and Settings\All Users\Application Data\Autodesk\RST 2011\Imperial Library\Furniture\Bed-Box.rfa";

Autodesk.Revit.DB.Family family = null;
if (!document.LoadFamily(fileName, out family))
{
    throw new Exception("Couldn't load " + fileName);
}

FilteredElementCollector collector = new FilteredElementCollector(document);
Floor floor = collector.OfClass(typeof(Floor)).FirstElement() as Floor;
if (floor != null)
{
    FamilySymbolSetIterator symbolItor = family.Symbols.ForwardIterator();
    int x = 0, y = 0;
    int i = 0;
    while (symbolItor.MoveNext())
    {
        FamilySymbol symbol = symbolItor.Current as FamilySymbol;
        XYZ location = new XYZ(x, y, 0);
        XYZ direction = new XYZ();
        switch (i % 3)
        {
            case 0:
                direction = new XYZ(1, 1, 0);
                break;
            case 1:
                direction = new XYZ(0, 1, 1);
                break;
            case 2:
                direction = new XYZ(1, 0, 1);
                break;
        }
```

```
        FamilyInstance instance = document.Create.NewFamilyInstance(location, symbol,
            direction, floor, StructuralType.NonStructural);
        x += 10;
        i++;
    }
}
else
{
    throw new Exception("Please open a model with at least one floor element before invoking
this command.");
}
```

The result of the previous code appears in the following picture:

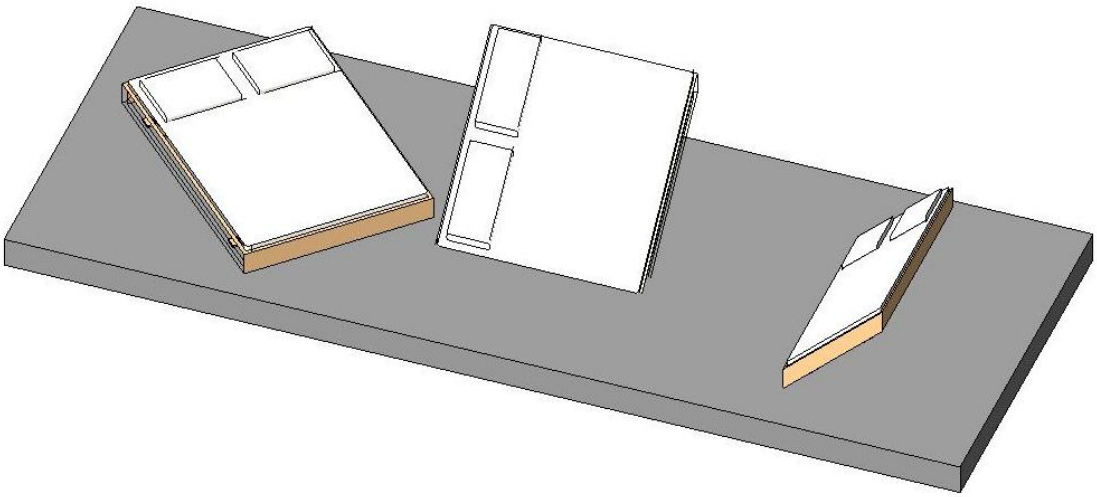


Figure 48: Create family instances using different reference directions

13 Family Creation

This chapter discusses families and how to:

- Create and modify Family documents
- Access family types and parameters

13.1 Family Documents

13.1.1 Family

The Family object represents an entire Revit family. A Family Document is a Document that represents a Family rather than a Revit project.

Using the Family Creation functionality of the Revit API, you can create and edit families and their types. This functionality is particularly useful when you have pre-existing data available from an external system that you want to convert to a Revit family library.

API access to system family editing is not available.

13.1.2 Categories

As noted in the previous chapter, the FamilyBase.FamilyCategory property indicates the category of the Family such as Columns, Furniture, Structural Framing, or Windows.

The following code can be used to determine the category of the family in an open Revit Family document.

Code Region 13-1: Category of open Revit Family Document

```
string categoryName = familyDoc.OwnerFamily.FamilyCategory.Name;
```

13.1.3 Parameters

Family parameters can be accessed from the OwnerFamily property of a Family Document as the following example shows.

Code Region 13-2: Category of open Revit Family Document

```
// get the owner family of the family document.
Family family = familyDoc.OwnerFamily;
Parameter param = family.get_Parameter(BuiltInParameter.FAMILY_WORK_PLANE_BASED);
// this param is a Yes/No parameter in UI, but an integer value in API
// 1 for true and 0 for false
int isTrue = param.AsInteger();
// param.Set(1); // set value to true.
```

13.1.4 Creating a Family Document

The ability to modify Revit Family documents and access family types and parameters is available from the Document class if the Document is a Family document, as determined by the IsFamilyDocument property. To edit an existing family while working in a Project document, use the EditFamily() functions available from the Document class, and then use LoadFamily() to reload the family back into the owner document after editing is complete. To create a new family document use Application.NewFamilyDocument():

Code Region 13-3: Creating a new Family document

```
// create a new family document using Generic Model.rft template
string templateFileName = @"C:\Documents and Settings\All Users\Application Data\Autodesk\RST 2011\Imperial Templates\Generic Model.rft";


Document familyDocument = application.NewFamilyDocument(templateFileName);
if (null == familyDocument)
{
    throw new Exception("Cannot open family document");
}
```

13.1.5 Nested Family Symbols

You can filter a Family Document for FamilySymbols to get all of the FamilySymbols loaded into the Family. In this code sample, all the nested FamilySymbols in the Family for a given FamilyInstance are listed.

Code Region 13-4: Getting nested Family symbols in a Family

```
public void GetLoadedSymbols(Autodesk.Revit.DB.Document document, FamilyInstance familyInstance)
{
    if (null != familyInstance.Symbol)
    {
        // Get family associated with this
        Family family = familyInstance.Symbol.Family;

        // Get Family document for family
        Document familyDoc = document.EditFamily(family);
        if (null != familyDoc && familyDoc.IsFamilyDocument == true)
        {
            String loadedFamilies = "FamilySymbols in " + family.Name + ":\n";
            FilteredElementCollector collector = new FilteredElementCollector(document);
            ICollection<Element> collection =
                collector.OfClass(typeof(FamilySymbol)).ToElements();
            foreach (Element e in collection)
            {
                FamilySymbol fs = e as FamilySymbol;
                loadedFamilies += "\t" + fs.Name + "\n";
            }

            TaskDialog.Show("Revit",loadedFamilies);
        }
    }
}
```

13.2 Family Item Factory

The FamilyItemFactory class provides the ability to create elements in family documents. It is accessed through the Document.FamilyCreate property. FamilyItemFactory is derived from the ItemFactoryBase class which is a utility to create elements in both Revit project documents and family documents.

13.2.1 Creating Forms

The FamilyItemFactory provides the ability to create form elements in families, such as extrusions, revolutions, sweeps, and blends. See section 17.2 for more information on these 3D sketch forms.

The following example demonstrates how to create a new Extrusion element. It creates a simple rectangular profile and then moves the newly created Extrusion to a new location.

Code Region 13-5: Creating a new Extrusion

```
private Extrusion CreateExtrusion(Autodesk.Revit.DB.Document document, SketchPlane
sketchPlane)
{
    Extrusion rectExtrusion = null;
    // make sure we have a family document
    if (true == document.IsFamilyDocument)
    {
        // define the profile for the extrusion
        CurveArrArray curveArrArray = new CurveArrArray();
        CurveArray curveArray1 = new CurveArray();
        CurveArray curveArray2 = new CurveArray();
        CurveArray curveArray3 = new CurveArray();
        // create a rectangular profile
        XYZ p0 = XYZ.Zero;
        XYZ p1 = new XYZ(10, 0, 0);
        XYZ p2 = new XYZ(10, 10, 0);
        XYZ p3 = new XYZ(0, 10, 0);
        Line line1 = document.Application.Create.NewLineBound(p0, p1);
        Line line2 = document.Application.Create.NewLineBound(p1, p2);
        Line line3 = document.Application.Create.NewLineBound(p2, p3);
        Line line4 = document.Application.Create.NewLineBound(p3, p0);
        curveArray1.Append(line1);
        curveArray1.Append(line2);
        curveArray1.Append(line3);
        curveArray1.Append(line4);
        curveArrArray.Append(curveArray1);
        // create solid rectangular extrusion
        rectExtrusion = document.FamilyCreate.NewExtrusion(true, curveArrArray,
                                                           sketchPlane, 10);
        if (null != rectExtrusion)
        {
            // move extrusion to proper place
            XYZ transPoint1 = new XYZ(-16, 0, 0);
            document.Move(rectExtrusion, transPoint1);
        }
        else
        {
            throw new Exception("Create new Extrusion failed.");
        }
    }
```

```
    else
    {
       throw new Exception("Please open a Family document before invoking this command.");
    }
    return rectExtrusion;
}
```

The following sample shows how to create a new Sweep from a solid ovoid profile in a Family Document.

Code Region 13-6: Creating a new Sweep

```
private Sweep CreateSweep(Autodesk.Revit.DB.Document document, SketchPlane sketchPlane)
{
    Sweep sweep = null;
    // make sure we have a family document
    if (true == document.IsFamilyDocument)
    {
        // Define a profile for the sweep
        CurveArrArray arrarr = new CurveArrArray();
        CurveArray arr = new CurveArray();
        // Create an ovoid profile
        XYZ pnt1 = new XYZ(0, 0, 0);
        XYZ pnt2 = new XYZ(2, 0, 0);
        XYZ pnt3 = new XYZ(1, 1, 0);
        arr.Append(document.Application.Create.NewArc(pnt2, 1.0d, 0.0d, 180.0d, XYZ.BasisX,
                                                      XYZ.BasisY));
        arr.Append(document.Application.Create.NewArc(pnt1, pnt3, pnt2));
        arrarr.Append(arr);
        SweepProfile profile = document.Application.Create.NewCurveLoopsProfile(arrarr);
        // Create a path for the sweep
        XYZ pnt4 = new XYZ(10, 0, 0);
        XYZ pnt5 = new XYZ(0, 10, 0);
        Curve curve = document.Application.Create.NewLineBound(pnt4, pnt5);
        CurveArray curves = new CurveArray();
        curves.Append(curve);
        // create a solid ovoid sweep
        sweep = document.FamilyCreate.NewSweep(true, curves, sketchPlane, profile, 0,
                                               ProfilePlaneLocation.Start);
        if (null != sweep)
        {
            // move to proper place
            XYZ transPoint1 = new XYZ(11, 0, 0);
            document.Move(sweep, transPoint1);
        }
        else
        {
            throw new Exception("Failed to create a new Sweep.");
        }
```

```
    }
    else
    {
        throw new Exception("Please open a Family document before invoking this command.");
    }
    return sweep;
}
```

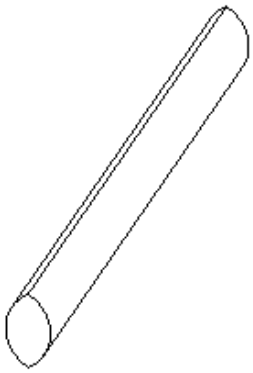


Figure 49: Ovoid sweep created by previous example

13.2.1.1 Assigning Subcategories to forms

After creating a new form in a family, you may want to change the subcategory for the form. For example, you may have a Door family and want to create multiple subcategories of doors and assign different subcategories to different door types in your family.

The following example shows how to create a new subcategory, assign it a material, and then assign the subcategory to a form.

Code Region 13-7: Assigning a subcategory

```
public void AssignSubCategory(Document document, GenericForm extrusion)
{
    // create a new subcategory
    Category cat = document.OwnerFamily.FamilyCategory;
    Category subCat = document.Settings.Categories.NewSubcategory(cat, "NewSubCat");

    // create a new material and assign it to the subcategory
    MaterialWood woodMaterial = document.Settings.Materials.AddWood("Wood Material");
    subCat.Material = woodMaterial;

    // assign the subcategory to the element
    extrusion.Subcategory = subCat;
}
```

13.2.2 Creating Annotations

New annotations such as Dimensions and ModelText and TextNote objects can also be created in families, as well as curve annotation elements such as SymbolicCurve, ModelCurve, and DetailCurve. See Chapter 16 for more information on Annotation elements.

Additionally, a new Alignment can be added, referencing a View that determines the orientation of the alignment, and two geometry references.

The following example demonstrates how to create a new arc length Dimension.

Code Region 13-8: Creating a Dimension

```
public Dimension CreateArcDimension(Document document, SketchPlane sketchPlane)
{
   Autodesk.Revit.Creation.Application appCreate = document.Application.Create;
   Line gLine1 = appCreate.NewLine(new XYZ(0, 2, 0), new XYZ(2, 2, 0), true);
   Line gLine2 = appCreate.NewLine(new XYZ(0, 2, 0), new XYZ(2, 4, 0), true);
   Arc arctoDim = appCreate.NewArc(new XYZ(1, 2, 0), new XYZ(-1, 2, 0), new XYZ(0, 3, 0));
   Arc arcofDim = appCreate.NewArc(new XYZ(0, 3, 0), new XYZ(1, 2, 0), new XYZ(0.8, 2.8, 0));

   Autodesk.Revit.Creation.FamilyItemFactory creationFamily = document.FamilyCreate;
   ModelCurve modelCurve1 = creationFamily.NewModelCurve(gLine1, sketchPlane);
   ModelCurve modelCurve2 = creationFamily.NewModelCurve(gLine2, sketchPlane);
   ModelCurve modelCurve3 = creationFamily.NewModelCurve(arctoDim, sketchPlane);
   //get their reference
   Reference ref1 = modelCurve1.GeometryCurve.Reference;
   Reference ref2 = modelCurve2.GeometryCurve.Reference;
   Reference arcRef = modelCurve3.GeometryCurve.Reference;

   Dimension newArcDim = creationFamily.NewArcLengthDimension(document.ActiveView, arcofDim,
                                                              arcRef, ref1, ref2);
   if (newArcDim == null)
   {
       throw new Exception("Failed to create new arc length dimension.");
   }

   return newArcDim;
}
```

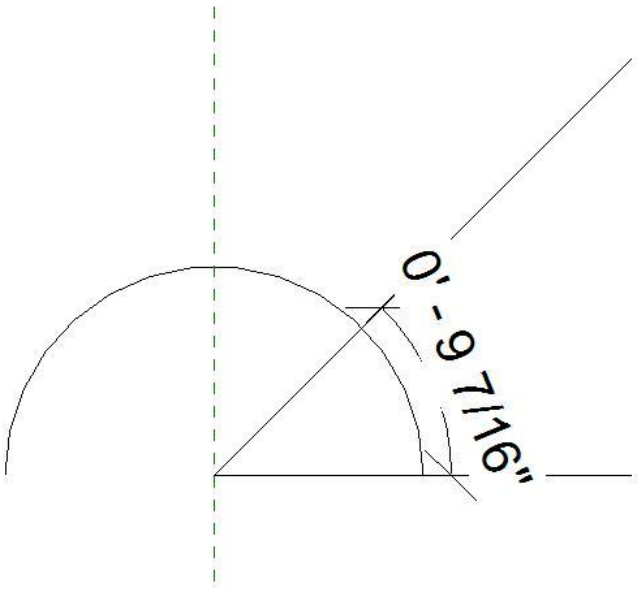


Figure 50: Resulting arc length dimension

Some types of dimensions can be labeled with a FamilyParameter. Dimensions that cannot be labeled will throw an Autodesk.Revit.Exceptions.InvalidOperationException if you try to get or set the Label property. In the following example, a new linear dimension is created between two lines and labeled as "width".

Code Region 13-9: Labeling a dimension

```
public Dimension CreateLinearDimension(Document document)
{
    // first create two lines
    XYZ pt1 = new XYZ(5, 5, 0);
    XYZ pt2 = new XYZ(5, 10, 0);
    Line line = document.Application.Create.NewLine(pt1, pt2, true);
    Plane plane = document.Application.Create.NewPlane(pt1.Cross(pt2), pt2);
    SketchPlane skplane = document.FamilyCreate.NewSketchPlane(plane);
    ModelCurve modelcurve1 = document.FamilyCreate.NewModelCurve(line, skplane);

    pt1 = new XYZ(10, 5, 0);
    pt2 = new XYZ(10, 10, 0);
    line = document.Application.Create.NewLine(pt1, pt2, true);
    plane = document.Application.Create.NewPlane(pt1.Cross(pt2), pt2);
    skplane = document.FamilyCreate.NewSketchPlane(plane);
    ModelCurve modelcurve2 = document.FamilyCreate.NewModelCurve(line, skplane);

    // now create a linear dimension between them
    ReferenceArray ra = new ReferenceArray();
    ra.Append(modelcurve1.GeometryCurve.Reference);
    ra.Append(modelcurve2.GeometryCurve.Reference);

    pt1 = new XYZ(5, 10, 0);
    pt2 = new XYZ(10, 10, 0);
```

```
    line = document.Application.Create.NewLine(pt1, pt2, true);

    Dimension dim = document.FamilyCreate.NewLinearDimension(document.ActiveView, line, ra);

    // create a label for the dimension called "width"
    FamilyParameter param = document.FamilyManager.AddParameter("width",
        Autodesk.Revit.Parameters.BuiltInParameterGroup.PG_CONSTRAINTS,
        Autodesk.Revit.Parameters.ParameterType.Length, false);

    dim.Label = param;

    return dim;
}
```

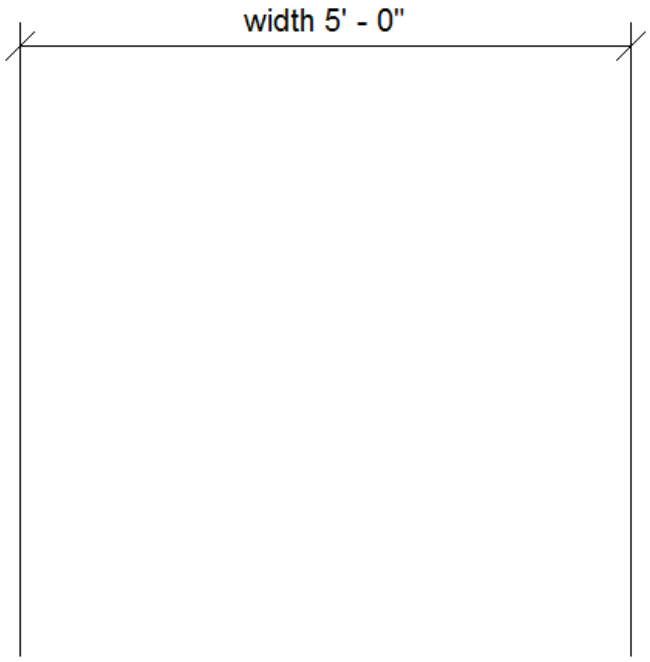


Figure 51: Labeled linear dimension

13.3 Family Element Visibility

The FamilyElementVisibility class can be used to control the visibility of family elements in the project document. For example, if you have a door family, you may only want the door swing to be visible in plan views in the project document in which doors are placed, not 3D views. By setting the visibility on the curves of the door swing, you can control their visibility. FamilyElementVisibility is applicable to the following family element classes which have the SetVisibility() function:

- GenericForm, which is the base class for form classes such as Sweep and Extrusion
- SymbolicCurve
- ModelText

- CurveByPoints
- ModelCurve
- ReferencePoint
- ImportInstance

In the example below, the resulting family document will display the text "Hello World" with a line under it. When the family is loaded into a Revit project document and an instance is placed, in plan view, only the line will be visible. In 3D view, both the line and text will be displayed, unless the Detail Level is set to Course, in which case the line will disappear.

Code Region 13-10: Setting family element visibility

```
public void CreateAndSetVisibility(Autodesk.Revit.DB.Document familyDocument, SketchPlane
sketchPlane)
{
    // create a new ModelCurve in the family document
    XYZ p0 = new XYZ(1, 1, 0);
    XYZ p1 = new XYZ(5, 1, 0);
    Line line1 = familyDocument.Application.Create.NewLineBound(p0, p1);

    ModelCurve modelCurve1 = familyDocument.FamilyCreate.NewModelCurve(line1, sketchPlane);

    // create a new ModelText along ModelCurve line
    ModelText text = familyDocument.FamilyCreate.NewModelText("Hello World", null,
                     sketchPlane, p0, Autodesk.Revit.DB.HorizontalAlign.Center, 0.1);

    // set visibility for text
    FamilyElementVisibility textVisibility =
                            new FamilyElementVisibility(FamilyElementVisibilityType.Model);
    textVisibility.IsShownInTopBottom = false;
    text.SetVisibility(textVisibility);

    // set visibility for line
    FamilyElementVisibility curveVisibility =
                            new FamilyElementVisibility(FamilyElementVisibilityType.Model);
    curveVisibility.IsShownInCoarse = false;
    modelCurve1.SetVisibility(curveVisibility);
}
```

13.4 Family Manager

Family documents provide access to the FamilyManager property. The FamilyManager class provides access to family types and parameters. Using this class you can add and remove types, add and remove family and shared parameters, set the value for parameters in different family types, and define formulas to drive parameter values.

13.4.1 Getting Types in a Family

The FamilyManager can be used to iterate through the types in a family, as the following example demonstrates.

Code Region 13-11: Getting the types in a family

```
public void GetFamilyTypesInFamily(Document familyDoc)
{
    if (familyDoc.IsFamilyDocument == true)
    {
        FamilyManager familyManager = familyDoc.FamilyManager;

        // get types in family
        string types = "Family Types: ";
        FamilyTypeSet familyTypes = familyManager.Types;
        FamilyTypeSetIterator familyTypesItor = familyTypes.ForwardIterator();
        familyTypesItor.Reset();
        while (familyTypesItor.MoveNext())
        {
            FamilyType familyType = familyTypesItor.Current as FamilyType;
            types += "\n" + familyType.Name;
        }
        MessageBox.Show(types, "Revit");
    }
}
```

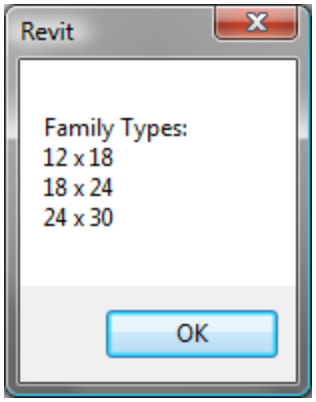


Figure 52: Family types in Concrete-Rectangular-Column family

13.4.2 Editing FamilyTypes

FamilyManager provides the ability to iterate through existing types in a family, and add and modify types and their parameters.

The following example shows how to add a new type, set its parameters and then assign the new type to a FamilyInstance. Type editing is done on the current type by using the Set() function. The current type is available from the CurrentType property. The CurrentType property can be used to set the current type before editing, or use the NewType() function which creates a new type and sets it to the current type for editing.

Note that once the new type is created and modified, Document.LoadFamily() is used to load the family back into the Revit project to make the new type available.

Code Region 13-12: Editing Family Types

```
public void EditFamilyTypes(Document document, FamilyInstance familyInstance)
{
    // example works best when familyInstance is a rectangular concrete element
```

```
    if (null != familyInstance.Symbol)
    {
        // Get family associated with this
        Family family = familyInstance.Symbol.Family;

        // Get Family document for family
        Document familyDoc = document.EditFamily(family);
        if (null != familyDoc)
        {
            FamilyManager familyManager = familyDoc.FamilyManager;

            // add a new type and edit its parameters
            FamilyType newFamilyType = familyManager.NewType("2X2");

            // look for 'b' and 'h' parameters and set them to 2 feet
            FamilyParameter familyParam = familyManager.get_Parameter("b");
            if (null != familyParam)
            {
                familyManager.Set(familyParam, 2.0);
            }
            familyParam = familyManager.get_Parameter("h");
            if (null != familyParam)
            {
                familyManager.Set(familyParam, 2.0);
            }

            // now update the Revit project with Family which has a new type
            family = familyDoc.LoadFamily(document);
            // find the new type and assign it to FamilyInstance
            FamilySymbolSetIterator symbolsItor = family.Symbols.ForwardIterator();
            symbolsItor.Reset();
            while (symbolsItor.MoveNext())
            {
                FamilySymbol familySymbol = symbolsItor.Current as FamilySymbol;
                if (familySymbol.Name == "2X2")
                {
                    familyInstance.Symbol = familySymbol;
                    break;
                }
            }
        }
    }
}
```

14 Conceptual Design

This chapter discusses the conceptual design functionality of the Revit API for the creation of complex geometry in a family document. Form-making is supported by the addition of new objects: points and spline curves that pass through these points. The resulting surfaces can be divided, patterned, and panelized to create buildable forms with persistent parametric relationships.

14.1 Point and curve objects

A reference point is an element that specifies a location in the XYZ work space of the conceptual design environment. You create reference points to design and plot lines, splines, and forms. A ReferencePoint can be added to a ReferencePointArray, then used to create a CurveByPoints, which in turn can be used to create a form.

The following example demonstrates how to create a CurveByPoints object. See the "Creating a loft form" example in the next section to see how to create a form from multiple CurveByPoints objects.

Code Region 14-1: Creating a new CurveByPoints

```
ReferencePointArray rpa = new ReferencePointArray();

XYZ xyz = document.Application.Create.NewXYZ(0, 0, 0);
ReferencePoint rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

xyz = document.Application.Create.NewXYZ(0, 30, 10);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

xyz = document.Application.Create.NewXYZ(0, 60, 0);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

xyz = document.Application.Create.NewXYZ(0, 100, 30);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

xyz = document.Application.Create.NewXYZ(0, 150, 0);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

CurveByPoints curve = document.FamilyCreate.NewCurveByPoints(rpa);

PointOnEdge poe =
document.Application.Create.NewPointOnEdge(curve.GeometryCurve.Reference, 0.5);
rp = document.FamilyCreate.NewReferencePoint(poe);
```

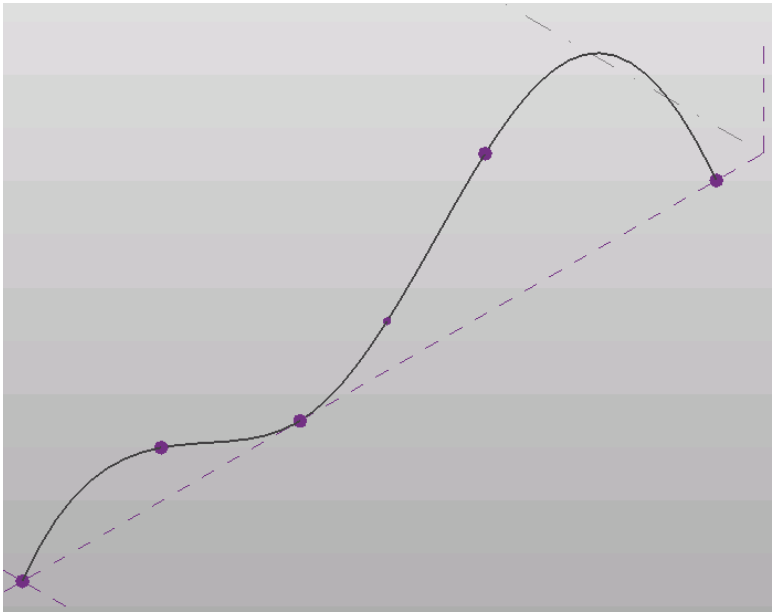


Figure 53: CurveByPoints curve

Reference points can be created based on XYZ coordinates as in the example above, or they can be created relative to other geometry so that the points will move when the referenced geometry changes. These points are created using the subclasses of the PointElementReference class. The subclasses are:

- PointOnEdge
- PointOnEdgeEdgeIntersection
- PointOnEdgeFaceIntersection
- PointOnFace
- PointOnPlane

For example, the last two lines of code in the previous example create a reference point in the middle of the CurveByPoints.

Forms can be created using model lines or reference lines. Model lines are "consumed" by the form during creation and no longer exist as separate entities. Reference lines, on the other hand, persist after the form is created and can alter the form if they are moved. Although the API does not have a ReferenceLine class, you can change a model line to a reference line using the ModelCurve.ChangeToReferenceLine() method.

Code Region 14-2: Using Reference Lines to create Form

```
private FormArray CreateRevolveForm(Document document)
{
    FormArray revolveForms = null;

    // Create one profile
    ReferenceArray ref_ar = new ReferenceArray();

    XYZ ptA = new XYZ(0, 0, 10);
    XYZ ptB = new XYZ(100, 0, 10);
```

```
    Line line = document.Application.Create.NewLine(ptA, ptB, true);
    ModelCurve modelcurve = MakeLine(document.Application, ptA, ptB);
    ref_ar.Append(modelcurve.GeometryCurve.Reference);

    ptA = new XYZ(100, 0, 10);
    ptB = new XYZ(100, 100, 10);
    modelcurve = MakeLine(document, ptA, ptB);
    ref_ar.Append(modelcurve.GeometryCurve.Reference);

    ptA = new XYZ(100, 100, 10);
    ptB = new XYZ(0, 0, 10);
    modelcurve = MakeLine(document, ptA, ptB);
    ref_ar.Append(modelcurve.GeometryCurve.Reference);

    // Create axis for revolve form
    ptA = new XYZ(-5, 0, 10);
    ptB = new XYZ(-5, 10, 10);
    ModelCurve axis = MakeLine(document, ptA, ptB);

    // make axis a Reference Line
    axis.ChangeToReferenceLine();

    // Typically this operation produces only a single form,
    // but some combinations of arguments will create multiple froms from a single profile.
    revolveForms = document.FamilyCreate.NewRevolveForms(true, ref_ar,
                                            axis.GeometryCurve.Reference, 0, Math.PI / 4);

    return revolveForms;
}

public ModelCurve MakeLine(Document doc, XYZ ptA, XYZ ptB)
{
    Autodesk.Revit.ApplicationServices.Application app = doc.Application;
    // Create plane by the points
    Line line = app.Create.NewLine(ptA, ptB, true);
    XYZ norm = ptA.Cross(ptB);
    if (norm.Length == 0) norm = XYZ.BasisZ;
    Plane plane = app.Create.NewPlane(norm, ptB);
    SketchPlane skplane = doc.FamilyCreate.NewSketchPlane(plane);
    // Create line here
    ModelCurve modelcurve = doc.FamilyCreate.NewModelCurve(line, skplane);
    return modelcurve;
}
```

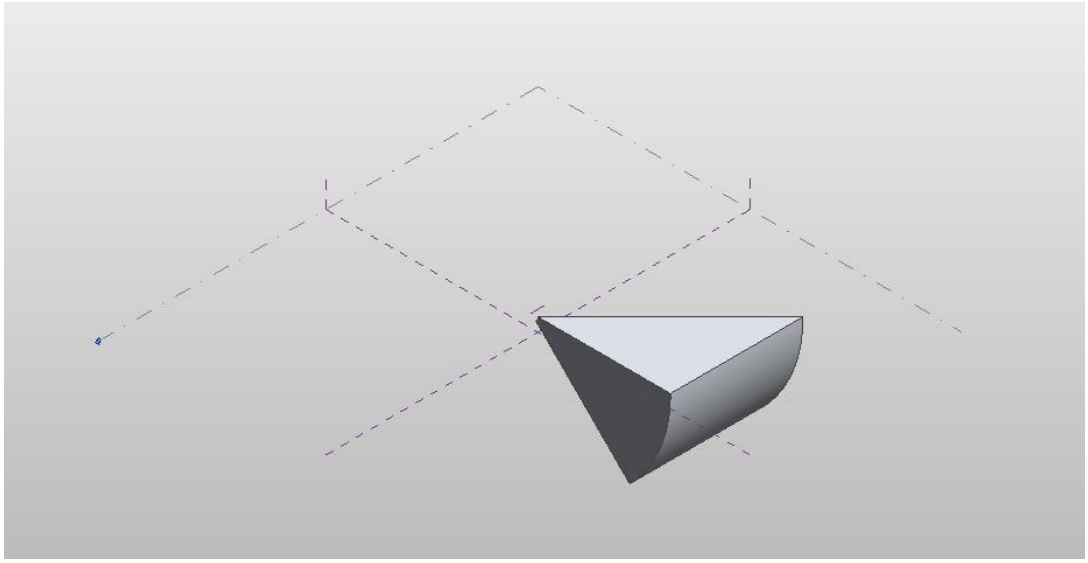


Figure 54: Resulting Revolve Form

14.2 Forms

14.2.1 Creating Forms

Similar to family creation, the conceptual design environment provides the ability to create new forms. The following types of forms can be created: extrusions, revolves, sweeps, swept blends, lofts, and surface forms. Rather than using the Blend, Extrusion, Revolution, Sweep, and SweptBlend classes used in Family creation, Mass families use the Form class for all types of forms.

An extrusion form is created from a closed curve loop that is planar. A revolve form is created from a profile and a line in the same plane as the profile which is the axis around which the shape is revolved to create a 3D form. A sweep form is created from a 2D profile that is swept along a planar path. A swept blend is created from multiple profiles, each one planar, that is swept along a single curve. A loft form is created from 2 or more profiles located on separate planes. A single surface form is created from a profile, similarly to an extrusion, but is given no height.

The following example creates a simple extruded form. Note that since the ModelCurves used to create the form are not converted to reference lines, they will be consumed by the resulting form.

Code Region 14-3: Creating an extrusion form

```
private Form CreateExtrusionForm(Autodesk.Revit.Document document)
{
    Form extrusionForm = null;

    // Create one profile
    ReferenceArray ref_ar = new ReferenceArray();

    XYZ ptA = new XYZ(10, 10, 0);
    XYZ ptB = new XYZ(90, 10, 0);
    ModelCurve modelcurve = MakeLine(document, ptA, ptB);
    ref_ar.Append(modelcurve.GeometryCurve.Reference);

    ptA = new XYZ(90, 10, 0);
```

```
    ptB = new XYZ(10, 90, 0);
    modelcurve = MakeLine(document, ptA, ptB);
    ref_ar.Append(modelcurve.GeometryCurve.Reference);

    ptA = new XYZ(10, 90, 0);
    ptB = new XYZ(10, 10, 0);
    modelcurve = MakeLine(document, ptA, ptB);
    ref_ar.Append(modelcurve.GeometryCurve.Reference);

    // The extrusion form direction
    XYZ direction = new XYZ(0, 0, 50);

    extrusionForm = document.FamilyCreate.NewExtrusionForm(true, ref_ar, direction);

    return extrusionForm;
}

public ModelCurve MakeLine(Document doc, XYZ ptA, XYZ ptB)
{
    Autodesk.Revit.ApplicationServices.Application app = doc.Application;
    // Create plane by the points
    Line line = app.Create.NewLine(ptA, ptB, true);
    XYZ norm = ptA.Cross(ptB);
    if (norm.Length == 0) norm = XYZ.BasisZ;
    Plane plane = app.Create.NewPlane(norm, ptB);
    SketchPlane skplane = doc.FamilyCreate.NewSketchPlane(plane);
    // Create line here
    ModelCurve modelcurve = doc.FamilyCreate.NewModelCurve(line, skplane);
    return modelcurve;
}
```

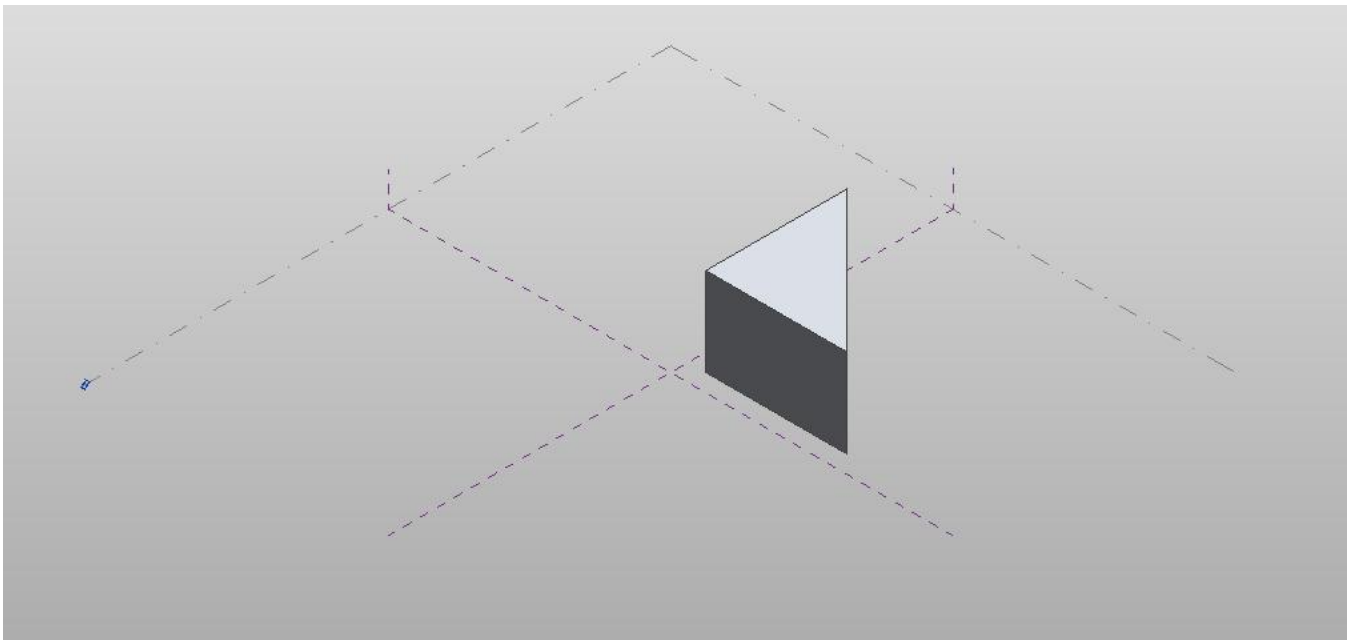


Figure 55: Resulting extrusion form

The following example shows how to create loft form using a series of CurveByPoints objects.

Code Region 14-4: Creating a loft form

```
private Form CreateLoftForm(Autodesk.Revit.Document document)
{
    Form loftForm = null;

    ReferencePointArray rpa = new ReferencePointArray();
    ReferenceArrayArray ref_ar_ar = new ReferenceArrayArray();
    ReferenceArray ref_ar = new ReferenceArray();
    ReferencePoint rp = null;
    XYZ xyz = null;

    // make first profile curve for loft
    xyz = document.Application.Create.NewXYZ(0, 0, 0);
    rp = document.FamilyCreate.NewReferencePoint(xyz);
    rpa.Append(rp);

    xyz = document.Application.Create.NewXYZ(0, 50, 10);
    rp = document.FamilyCreate.NewReferencePoint(xyz);
    rpa.Append(rp);

    xyz = document.Application.Create.NewXYZ(0, 100, 0);
    rp = document.FamilyCreate.NewReferencePoint(xyz);
    rpa.Append(rp);

    CurveByPoints cbp = document.FamilyCreate.NewCurveByPoints(rpa);
    ref_ar.Append(cbp.GeometryCurve.Reference);
    ref_ar_ar.Append(ref_ar);
    rpa.Clear();
```

```
    ref_ar = new ReferenceArray();

    // make second profile curve for loft
    xyz = document.Application.Create.NewXYZ(50, 0, 0);
    rp = document.FamilyCreate.NewReferencePoint(xyz);
    rpa.Append(rp);

    xyz = document.Application.Create.NewXYZ(50, 50, 30);
    rp = document.FamilyCreate.NewReferencePoint(xyz);
    rpa.Append(rp);

    xyz = document.Application.Create.NewXYZ(50, 100, 0);
    rp = document.FamilyCreate.NewReferencePoint(xyz);
    rpa.Append(rp);

    cbp = document.FamilyCreate.NewCurveByPoints(rpa);
    ref_ar.Append(cbp.GeometryCurve.Reference);
    ref_ar_ar.Append(ref_ar);
    rpa.Clear();
    ref_ar = new ReferenceArray();

    // make third profile curve for loft
    xyz = document.Application.Create.NewXYZ(75, 0, 0);
    rp = document.FamilyCreate.NewReferencePoint(xyz);
    rpa.Append(rp);

    xyz = document.Application.Create.NewXYZ(75, 50, 5);
    rp = document.FamilyCreate.NewReferencePoint(xyz);
    rpa.Append(rp);

    xyz = document.Application.Create.NewXYZ(75, 100, 0);
    rp = document.FamilyCreate.NewReferencePoint(xyz);
    rpa.Append(rp);

    cbp = document.FamilyCreate.NewCurveByPoints(rpa);
    ref_ar.Append(cbp.GeometryCurve.Reference);
    ref_ar_ar.Append(ref_ar);

    loftForm = document.FamilyCreate.NewLoftForm(true, ref_ar_ar);

    return loftForm;
}
```

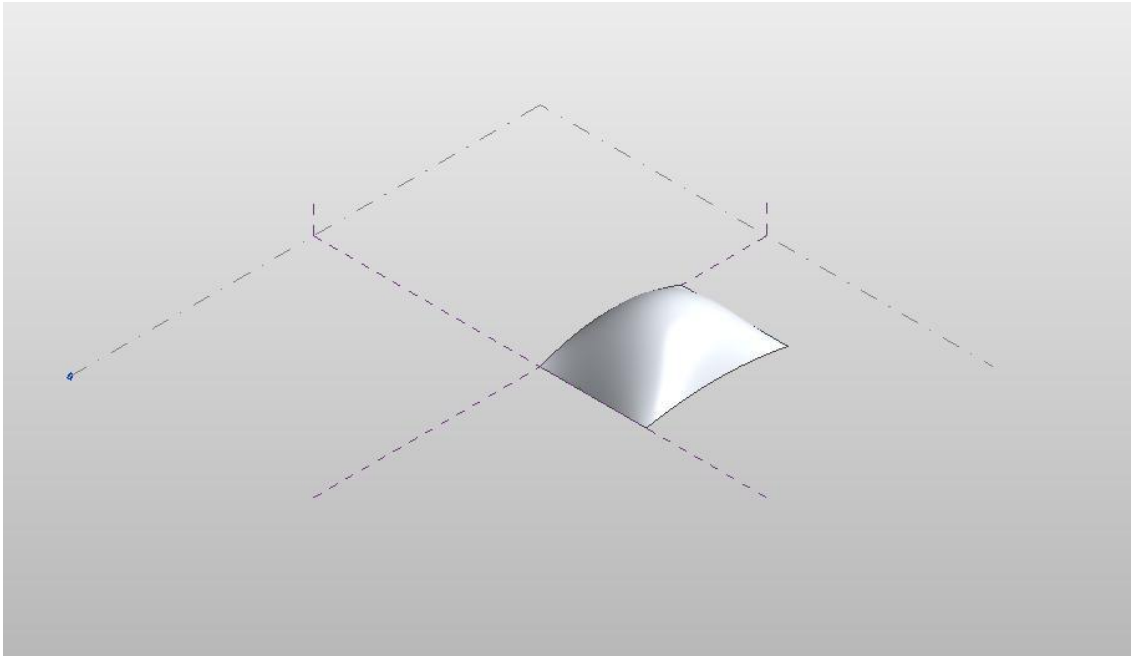


Figure 56: Resulting loft form

14.2.2 Form modification

Once created, forms can be modified by changing a sub element (i.e. a face, edge, curve or vertex) of the form, or an entire profile. The methods to modify a form include:

- AddEdge
- AddProfile
- DeleteProfile
- DeleteSubElement
- MoveProfile
- MoveSubElement
- RotateProfile
- RotateSubElement
- ScaleSubElement

Additionally, you can modify a form by adding an edge or a profile, which can then be modified using the methods listed above.

The following example moves the first profile curve of the given form by a specified offset. The corresponding figure shows the result of applying this code to the loft form from the previous example.

Code Region 14-5: Moving a profile

```
public void MoveForm(Form form)
{
    int profileCount = form.ProfileCount;
    if (form.ProfileCount > 0)
    {
```

```
        int profileIndex = 0;   // modify the first form only
        if (form.CanManipulateProfile(profileIndex))
        {
            XYZ offset = new XYZ(-25, 0, 0);
            form.MoveProfile(profileIndex, offset);
        }
    }
}
```

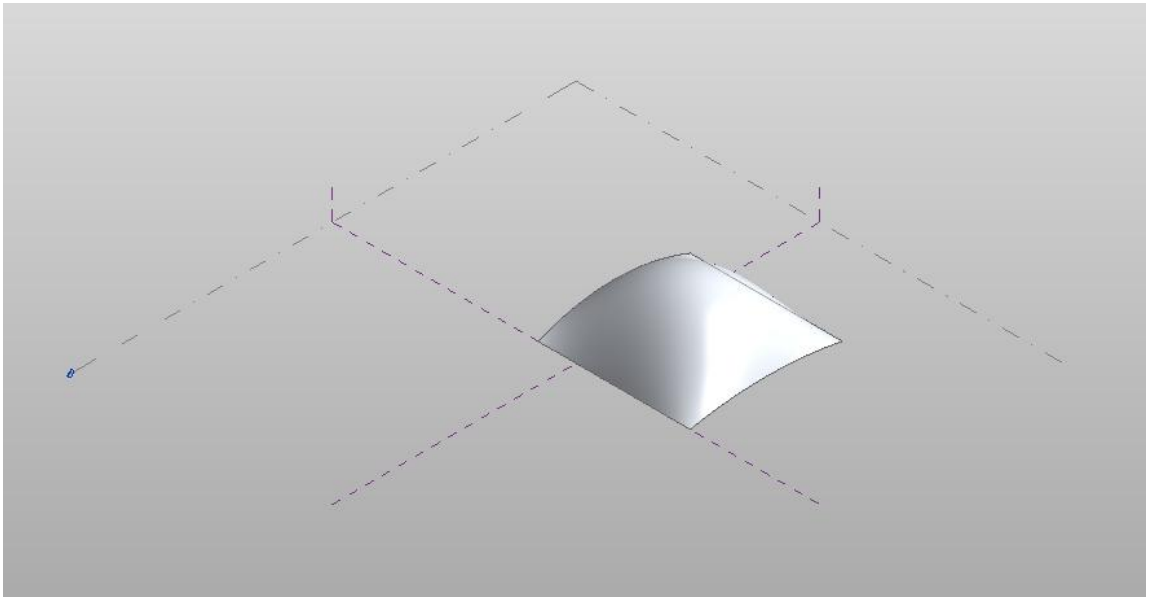


Figure 57: Modified loft form

The next sample demonstrates how to move a single vertex of a given form. The corresponding figure demonstrate the effect of this code on the previous extrusion form example

Code Region 14-6: Moving a sub element

```
public void MoveSubElement(Form form)
{
    if (form.ProfileCount > 0)
    {
        int profileIndex = 0;   // get first profile
        ReferenceArray ra = form.get_CurveLoopReferencesOnProfile(profileIndex, 0);
        foreach (Reference r in ra)
        {
            ReferenceArray ra2 = form.GetControlPoints(r);
            foreach (Reference r2 in ra2)
            {
                Point vertex = r2.GeometryObject as Point;

                XYZ offset = new XYZ(0, 15, 0);
                form.MoveSubElement(r2, offset);
                break;  // just move the first point
```

```
                }
            }
        }
}
```

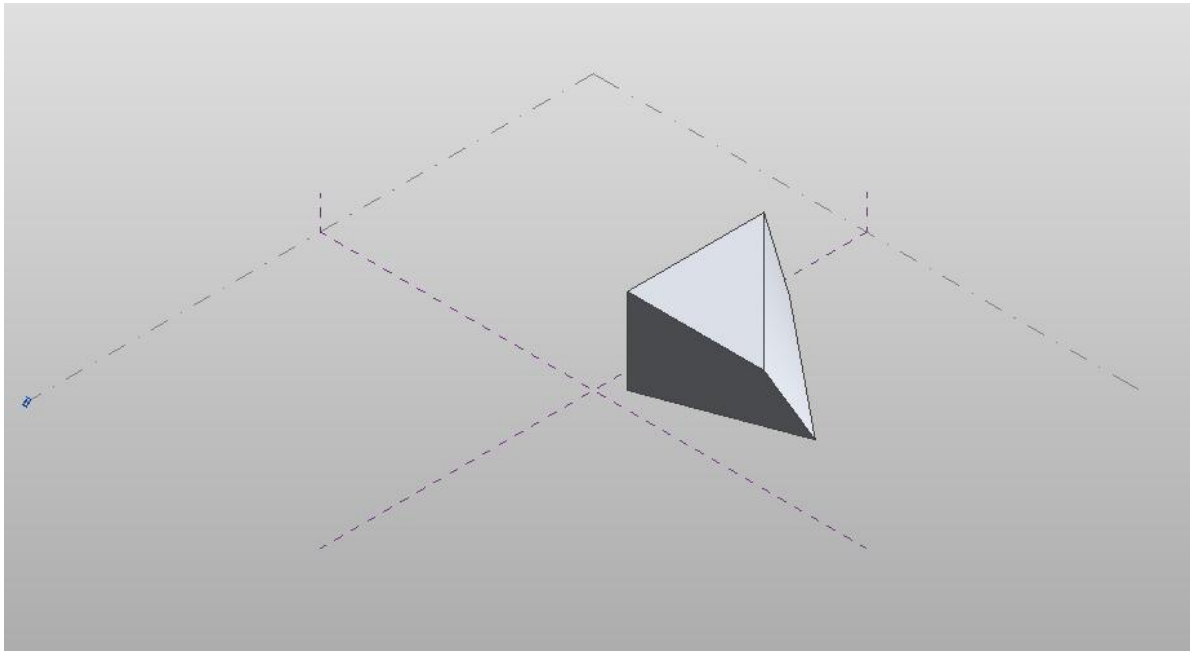


Figure 58: Modified extrusion form

14.3 Rationalizing a Surface

14.3.1 Dividing a surface

Faces of forms can be divided with UV grids. You can access the data for a divided surface using the Form.GetDividedSurfaceData() method (as is shown in a subsequent example) as well as create new divided surfaces on forms as shown below.

Code Region 14-7: Dividing a surface

```
public void DivideSurface(Document document, Form form)
{
    Application application = document.Application;
    Options opt = application.Create.NewGeometryOptions();
    opt.ComputeReferences = true;

    Autodesk.Revit.Geometry.Element geomElem = form.get_Geometry(opt);
    foreach (GeometryObject geomObj in geomElem.Objects)
    {
        Solid solid = geomObj as Solid;
        foreach (Face face in solid.Faces)
        {
            if (face.Reference != null)
            {
                DividedSurface ds = document.FamilyCreate.NewDividedSurface(face.Reference);
```

```
                    // create a divided surface with fixed number of U and V grid lines
                    SpacingRule srU = ds.USpacingRule;
                    srU.SetLayoutFixedNumber(16, SpacingRuleJustification.Center, 0, 0);

                    SpacingRule srV = ds.VSpacingRule;
                    srV.SetLayoutFixedNumber(24, SpacingRuleJustification.Center, 0, 0);

                    break;  // just divide one face of form
                }
            }
        }
    }
```

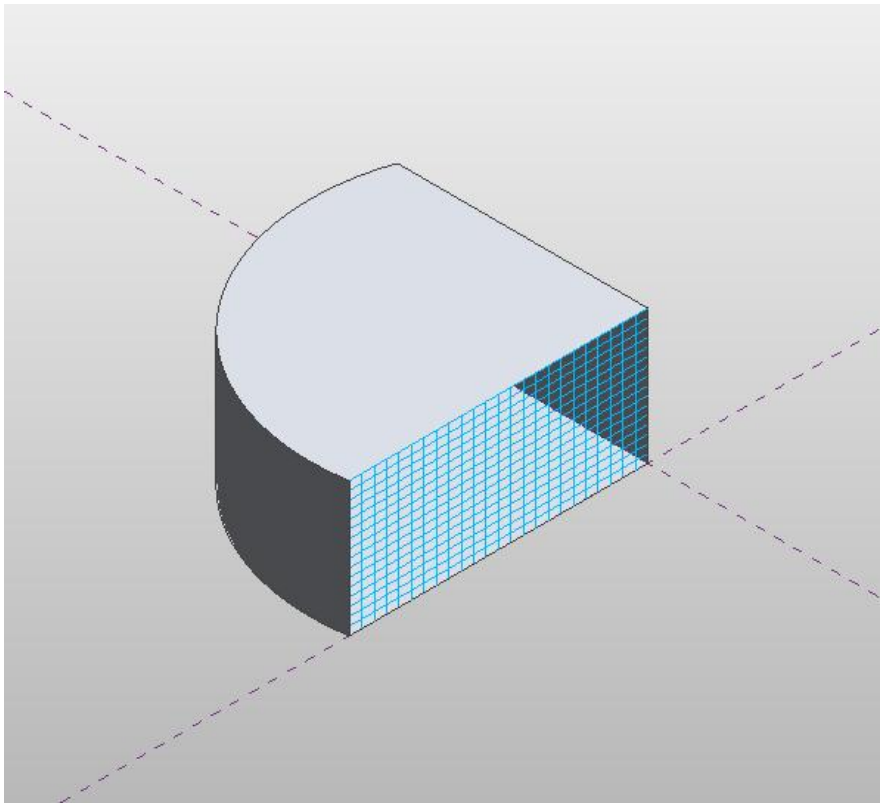


Figure 59: Face of form divided by UV grids

Accessing the USpacing and VSpacing properties of DividedSurface, you can define the SpacingRule for the U and V gridlines by specifying either a fixed number of grids (as in the example above), a fixed distance between grids, or a minimum or maximum spacing between grids. Additional information is required for each spacing rule, such as justification and grid rotation.

14.3.2 **Patterning a surface**

A divided surface can be patterned. Any of the built-in tile patterns can be applied to a divided surface. A tile pattern is an ElementType that is assigned to the DividedSurface. The tile pattern is applied to the surface according to the UV grid layout, so changing the USpacing and VSpacing properties of the DividedSurface will affect how the patterned surface appears.

The following example demonstrates how to cover a divided surface with the OctagonRotate pattern. The corresponding figure shows how this looks when applied to the divided surface in the previous example. Note this example also demonstrates how to get a DividedSurface on a form.

Code Region 14-8: Patterning a surface

```
public void TileSurface(Document document, Form form)
{
    // cover surface with OctagonRotate tile pattern
    TilePatterns patterns = document.Settings.TilePatterns;
    DividedSurfaceData dsData = form.GetDividedSurfaceData();
    if (dsData != null)
    {
        foreach (Reference r in dsData.GetReferencesWithDividedSurfaces())
        {
            DividedSurface ds = dsData.GetDividedSurfaceForReference(r);
            ds.ChangeTypeId(patterns.GetTilePattern(TilePatternsBuiltIn.OctagonRotate).Id);
        }
    }
}
```

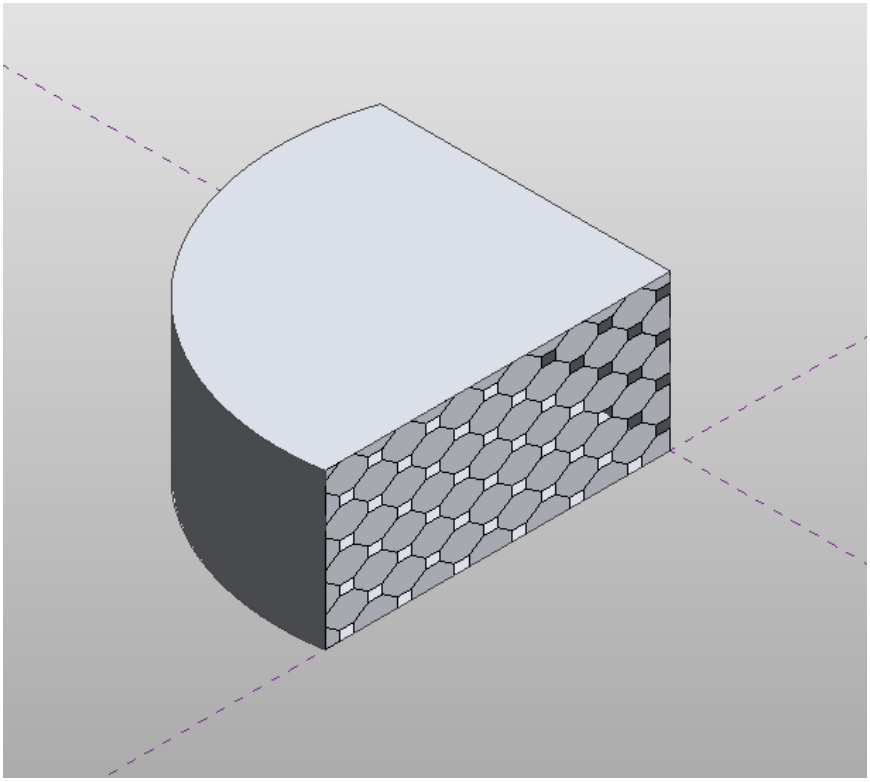


Figure 60: Tile pattern applied to divided surface

In addition to applying built-in tile patterns to a divided surface, you can create your own massing panel families using the "Curtain Panel Pattern Based.rft" template. These panel families can then be loaded into massing families and applied to divided surfaces using the DividedSurface.ChangeTypeId() method.

The following properties of Family are specific to curtain panel families:

- IsCurtainPanelFamily
- CurtainPanelHorizontalSpacing – horizontal spacing of driving mesh
- CurtainPanelVerticalSpacing – vertical spacing of driving mesh

- CurtainPanelTilePattern – choice of tile pattern

The following example demonstrates how to edit a massing panel family which can then be applied to a form in a conceptual mass document. To run this example, first create a new family document using the "Curtain Panel Pattern Based.rft" template.

Code Region 14-9: Editing a curtain panel family

```
Family family = document.OwnerFamily;
if (family.IsCurtainPanelFamily == true &&
    family.CurtainPanelTilePattern == TilePatternsBuiltIn.Rectangle)
{
    // first change spacing of grids in family document
    family.CurtainPanelHorizontalSpacing = 20;
    family.CurtainPanelVerticalSpacing = 30;

    // create new points and lines on grid
    Autodesk.Revit.ApplicationServices.Application app = document.Application;
    FilteredElementCollector collector = new FilteredElementCollector(document);
    ICollection<Element> collection = collector.OfClass(typeof(ReferencePoint)).ToElements();
    int ctr = 0;
    ReferencePoint rp0 = null, rp1 = null, rp2 = null, rp3 = null;
    foreach (Autodesk.Revit.DB.Element e in collection)
    {
        ReferencePoint rp = e as ReferencePoint;
        switch (ctr)
        {
            case 0:
                rp0 = rp;
                break;
            case 1:
                rp1 = rp;
                break;
            case 2:
                rp2 = rp;
                break;
            case 3:
                rp3 = rp;
                break;
        }
        ctr++;
    }

    ReferencePointArray rpAr = new ReferencePointArray();
    rpAr.Append(rp0);
    rpAr.Append(rp2);
    CurveByPoints curve1 = document.FamilyCreate.NewCurveByPoints(rpAr);
    PointOnEdge poeA = app.Create.NewPointOnEdge(curve1.GeometryCurve.Reference, 0.25);
    ReferencePoint rpA = document.FamilyCreate.NewReferencePoint(poeA);
    PointOnEdge poeB = app.Create.NewPointOnEdge(curve1.GeometryCurve.Reference, 0.75);
```

```
    ReferencePoint rpB = document.FamilyCreate.NewReferencePoint(poeB);

    rpAr.Clear();
    rpAr.Append(rp1);
    rpAr.Append(rp3);
    CurveByPoints curve2 = document.FamilyCreate.NewCurveByPoints(rpAr);
    PointOnEdge poeC = app.Create.NewPointOnEdge(curve2.GeometryCurve.Reference, 0.25);
    ReferencePoint rpC = document.FamilyCreate.NewReferencePoint(poeC);
    PointOnEdge poeD = app.Create.NewPointOnEdge(curve2.GeometryCurve.Reference, 0.75);
    ReferencePoint rpD = document.FamilyCreate.NewReferencePoint(poeD);
}
else
{
    throw new Exception("Please open a curtain family document before calling this
command.");
}
```

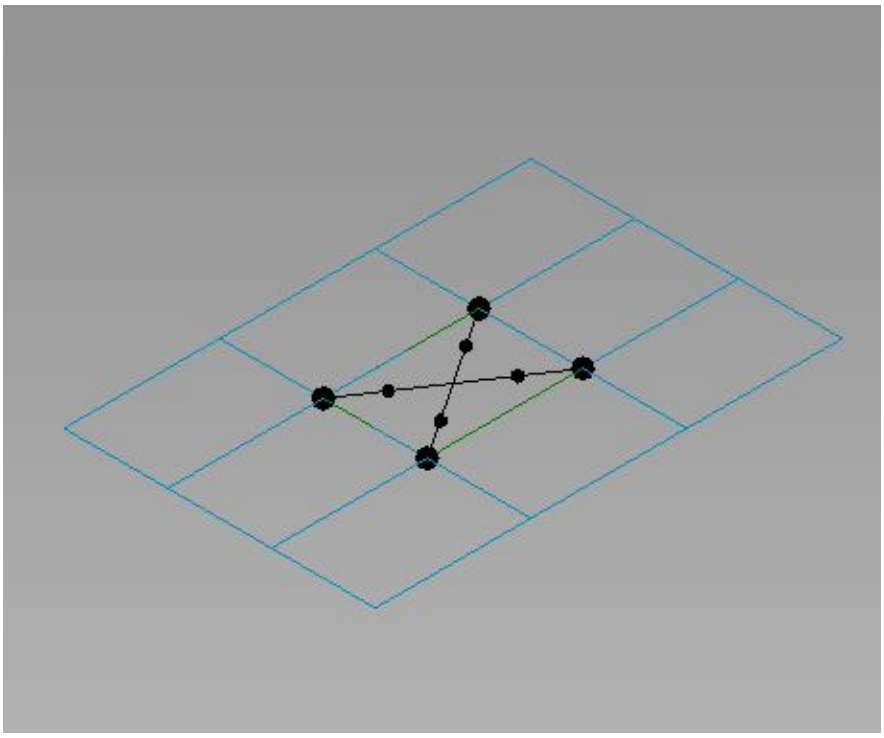


Figure 61: Curtain panel family

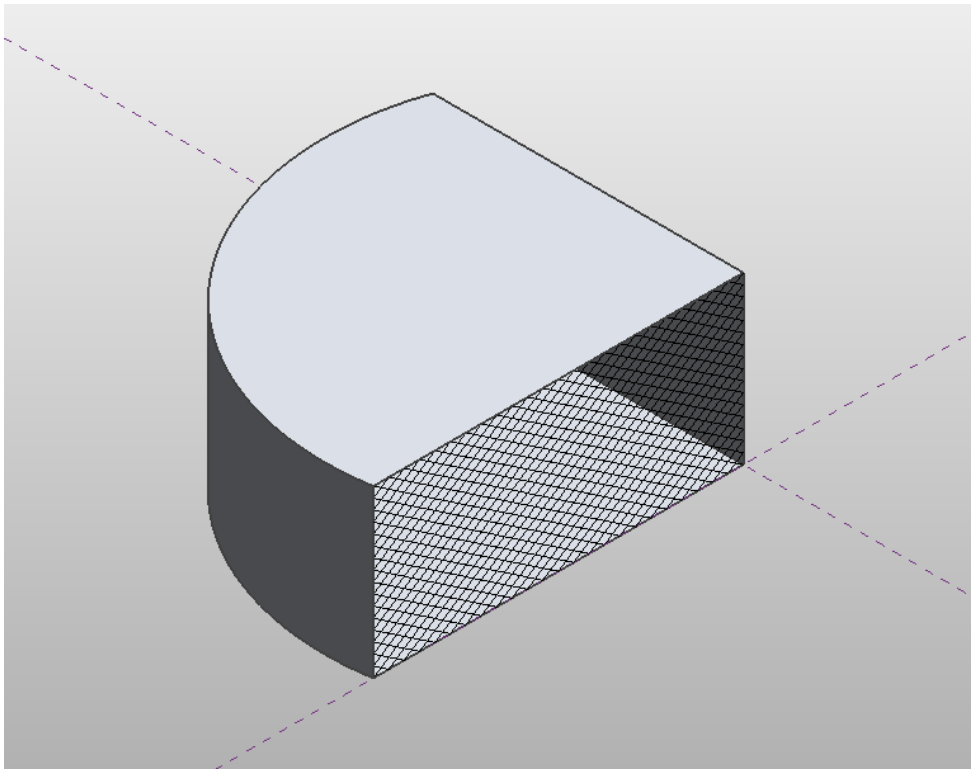


Figure 62: Curtain panel assigned to divided surface

15 Datum and Information Elements

This chapter introduces Datum Elements and Information Elements in Revit.

- Datum Elements include levels, grids, and ModelCurves.
- Information Elements include phases, design options, and EnergyDataSettings.

For more information about Revit Element classifications, refer to the Elements Essentials chapter

Note: If you need more information, refer to the related chapter:

- For LoadBase, LoadCase, LoadCombination, LoadNature and LoadUsage, refer to the Revit Structure chapter
- For ModelCurve, refer to the Sketching chapter
- For Material and FillPattern, refer to the Material chapter
- For EnergyDataSettings, refer to the Revit Architecture chapter

15.1 Levels

A level is a finite horizontal plane that acts as a reference for level-hosted elements, such as walls, roofs, floors, and ceilings. In the Revit Platform API, the Level class is derived from the Element class. The inherited Name property is used to retrieve the user-visible level name beside the level bubble in the Revit UI. To retrieve all levels in a project, use the ElementIterator iterator to search for Level objects.

15.1.1 Elevation

The Level class has the following properties:

- The Elevation property (LEVEL_ELEV) is used to retrieve or change the elevation above or below ground level.
- The ProjectElevation property is used to retrieve the elevation relative to the project origin regardless of the Elevation Base parameter value.
- Elevation Base is a Level type parameter.
  - Its BuiltInParameter is LEVEL_RELATIVE_BASE_TYPE.
  - Its StorageType is Integer
  - 0 corresponds to Project and 1 corresponds to Shared.

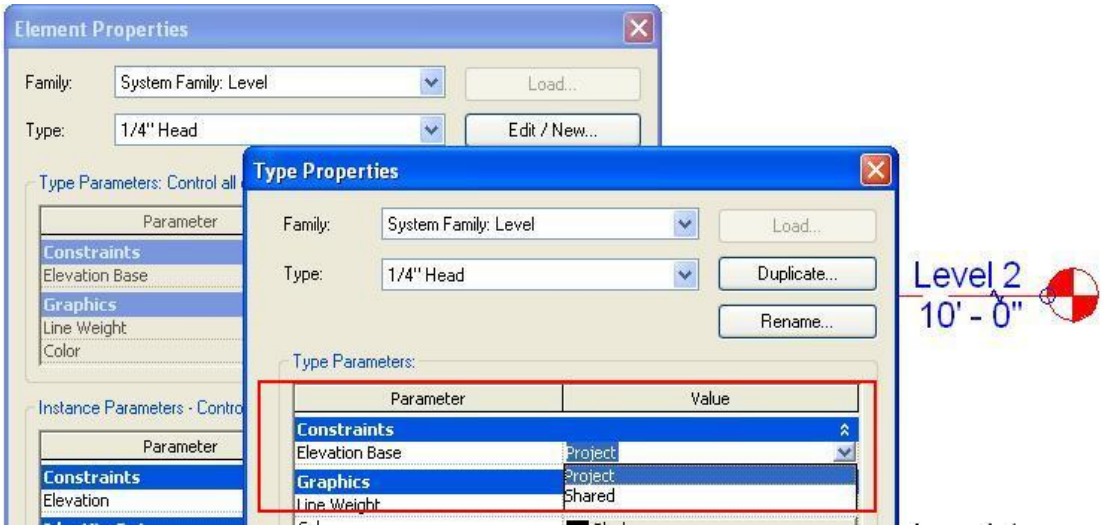


Figure 63: Level Type Elevation Base property

The following code sample illustrates how to retrieve all levels in a project using ElementIterator.

Code Region 15-1: Retrieving all Levels

```
private void Getinfo_Level(Document document)
{
    StringBuilder levelInformation = new StringBuilder();
    int levelNumber = 0;
    FilteredElementCollector collector = new FilteredElementCollector(document);
    ICollection<Element> collection = collector.OfClass(typeof(Level)).ToElements();
    foreach (Element e in collection)
    {
        Level level = e as Level;

        if (null != level)
        {
            // keep track of number of levels
            levelNumber++;

            //get the name of the level
            levelInformation.Append("\nLevel Name: " + level.Name);

            //get the elevation of the level
            levelInformation.Append("\n\tElevation: " + level.Elevation);

            // get the project elevation of the level
            levelInformation.Append("\n\tProject Elevation: " + level.ProjectElevation);
        }
    }

    //number of total levels in current document
    levelInformation.Append("\n\n There are " + levelNumber + " levels in the document!");

    //show the level information in the messagebox
```

```
    TaskDialog.Show("Revit",levelInformation.ToString());
}
```

15.1.2 Creating a Level

Using the Level command, you can define a vertical height or story within a building and you can create a level for each existing story or other building references. Levels must be added in a section or elevation view. Additionally, you can create a new level using the Revit Platform API.

The following code sample illustrates how to create a new level.

Code Region 15-2: Creating a new Level

```
Level CreateLevel(Autodesk.Revit.Document document)
{
    // The elevation to apply to the new level
    double elevation = 20.0;

    // Begin to create a level
    Level level = document.Create.NewLevel(elevation);
    if (null == level)
    {
        throw new Exception("Create a new level failed.");
    }

    // Change the level name
    level.Name = "New level";

    return level;
}
```

Note: After creating a new level, Revit does not create the associated plan view for this level. If necessary, you can create it yourself. For more information about how to create a plan view, refer to the Views chapter.

15.2 Grids

Grids are represented by the Grid class which is derived from the Element class. It contains all grid properties and methods. The inherited Name property is used to retrieve the content of the grid line's bubble.

15.2.1 Curve

The Grid class Curve property gets the object that represents the grid line geometry.

- If the IsCurved property returns true, the Curve property will be an Arc class object.
- If the IsCurved property returns false, the Curve property will be a Line class object.

For more information, refer to the Geometry chapter.

The following code is a simple example using the Grid class. The result appears in a message box after invoking the command.

Code Region 15-3: Using the Grid class

```
public void GetInfo_Grid(Grid grid)
```

```
{
    string message = "Grid : ";

    // Show IsCurved property
    message += "\nIf grid is Arc : " + grid.IsCurved;

    // Show Curve information
    Autodesk.Revit.DB.Curve curve = grid.Curve;
    if (grid.IsCurved)
    {
        // if the curve is an arc, give center and radius information
        Autodesk.Revit.DB.Arc arc = curve as Autodesk.Revit.DB.Arc;
        message += "\nArc's radius: " + arc.Radius;
        message += "\nArc's center:  (" + XYZToString(arc.Center);
    }
    else
    {
        // if the curve is a line, give length information
        Autodesk.Revit.Geometry.Line line = curve as Autodesk.Revit.DB.Line;
        message += "\nLine's Length: " + line.Length;
    }
    // Get curve start point
    message += "\nStart point: " + XYZToString(curve.get_EndPoint(0));
    // Get curve end point
    message += "; End point: " + XYZToString(curve.get_EndPoint(0));

    TaskDialog.Show("Revit", message);
}

// output the point's three coordinates
string XYZToString(XYZ point)
{
    return "(" + point.X + ", " + point.Y + ", " + point.Z + ")";
}
```

15.2.2 Creating a Grid

Two overloaded Document methods are available to create a new grid in the Revit Platform API. Using the following method with different parameters, you can create a curved or straight grid:

Code Region 15-4: NewGrid()

```
public Grid NewGrid( Arc arc );
public Grid NewGrid( Line line );
```

Note: The arc or the line used to create a grid must be in a horizontal plane.

The following code sample illustrates how to create a new grid with a line or an arc.

Code Region 15-5: Creating a grid with a line or an arc

```
void CreateGrid(Autodesk.Revit.Document document)
{
    // Create the geometry line which the grid locates
    XYZ start = new XYZ(0, 0, 0);
    XYZ end = new XYZ(30, 30, 0);
    Line geomLine = Line.get_Bound(start, end);

    // Create a grid using the geometry line
    Grid lineGrid = document.Create.NewGrid(geomLine);

    if (null == lineGrid)
    {
        throw new Exception("Create a new straight grid failed.");
    }

    // Modify the name of the created grid
    lineGrid.Name = "New Name1";

    // Create the geometry arc which the grid locates
    XYZ end0 = new XYZ(0, 0, 0);
    XYZ end1 = new XYZ(10, 40, 0);
    XYZ pointOnCurve = new XYZ(5, 7, 0);
    Arc geomArc = document.Application.Create.NewArc(end0, end1, pointOnCurve);

    // Create a grid using the geometry arc
    Grid arcGrid = document.Create.NewGrid(geomArc);

    if (null == arcGrid)
    {
        throw new Exception("Create a new curved grid failed.");
    }

    // Modify the name of the created grid
    arcGrid.Name = "New Name2";
}
```

Note: In Revit, the grids are named automatically in a numerical or alphabetical sequence when they are created.

You can also create several grids at once using the Document.NewGrids() method, which takes a CurveArray parameter.

15.3 Phase

Some architectural projects, such as renovations, proceed in phases. Phases have the following characteristics:

- Phases represent distinct time periods in a project lifecycle.
- The lifetime of an element within a building is controlled by phases.

- Each element has a construction phase but only the elements with a finite lifetime have a destruction phase.

All phases in a project can be retrieved from the Document object. A Phase object contains three pieces of useful information: Name, ID and UniqueId. The remaining properties always return null or an empty collection.

Each new modeling component added to a project has a Phase Created and a Phase Demolished property. The Phase Created property has the following characteristics:

- It identifies the phase in which the component was added.
- The default value is the same as the current view Phase value.
- Change the Phase Created parameter by selecting a new value from the drop-down list.

The Phase Demolished property has the following characteristics:

- It identifies in which phase the component is demolished.
- The default value is none.
- Demolishing a component with the demolition tool updates the property to the current Phase value in the view where you demolished the element.
- You can demolish a component by setting the Phase Demolished property to a different value.
- If you delete a phase using the Revit Platform API, all modeling components in the current phase still exist. The Phase Created parameter value for these components is changed to the next item in the drop-down list in the Properties dialog box.

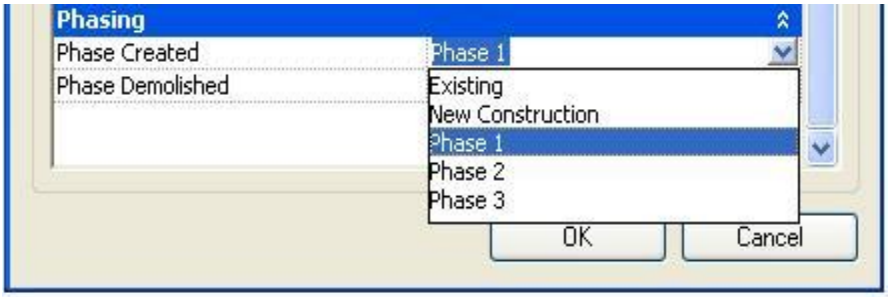


Figure 64: Phase-created component parameter value

The following code sample displays all supported phases in the current document. The phase names are displayed in a message box.

Code Region 15-6: Displaying all supported phases

```
void Getinfo_Phase(Document doc)
{
    // Get the phase array which contains all the phases.
    PhaseArray phases = doc.Phases;
    // Format the string which identifies all supported phases in the current document.
    String prompt = null;
    if (0 != phases.Size)
    {
        prompt = "All the phases in current document list as follow:";
        foreach (Phase ii in phases)
```

```
        {
            prompt += "\n\t" + ii.Name;
        }
    }
    else
    {
        prompt = "There are no phases in current document.";
    }
    // Give the user the information.
    TaskDialog.Show("Revit",prompt);
}
```

15.4 Design Options

Design options provide a way to explore alternative designs in a project. Design options provide the flexibility to adapt to changes in project scope or to develop alternative designs for review. You can begin work with the main project model and then develop variations along the way to present to a client. Most elements can be added into a design option. Elements that cannot be added into a design option are considered part of the main model and have no design alternatives.

The main use for Design options is as a property of the Element class. See the following example.

Code Region 15-7: Using design options

```
void Getinfo_DesignOption(Document document)
{
    // Get the selected Elements in the Active Document
    UIDocument uidoc = new UIDocument(document);
    ElementSet selection = uidoc.Selection.Elements;

    foreach (Autodesk.Revit.DB.Element element in selection)
    {
        // Use the DesignOption property of Element
        if (element.DesignOption != null)
        {
            TaskDialog.Show("Revit", element.DesignOption.Name.ToString());
        }
    }
}
```

The following rules apply to Design Options

- The value of the DesignOption property is null if the element is in the Main Model. Otherwise, the name you created in the Revit UI is returned.
- Only one active DesignOption Element can exist in an ActiveDocument.
    - The primary option is considered the default active DesignOption. For example, a design option set is named Wall and there are two design options in this set named “brick wall” and “glass wall”. If “brick wall” is the primary option, only this option and elements that belong to it are retrieved by the Element Iterator. “Glass wall” is inactive.

16 Annotation Elements

This chapter introduces Revit Annotation Elements, including the following:

- Dimension
- DetailCurve
- IndependentTag
- TextNote
- AnnotationSymbol

Note that:

- Dimensions are view-specific elements that display sizes and distances in a project.
- Detail curves are created for detailed drawings. They are visible only in the view in which they are drawn. Often they are drawn over the model view.
- Tags are an annotation used to identify elements in a drawing. Properties associated with a tag can appear in schedules.
- AnnotationSymbol has multiple leader options when loaded into a project.

For more information about Revit Element classification, refer to the Elements Essentials chapter.

16.1 Dimensions and Constraints

- The Dimension class represents permanent dimensions and dimension related constraint elements. Temporary dimensions created while editing an element in the UI are not accessible. Spot elevation and spot coordinate are represented by the SpotDimension class.

The following code sample illustrates, near the end, how to distinguish permanent dimensions from constraint elements.

Code Region 16-1: Distinguishing permanent dimensions from constraints

```
public void GetInfo_Dimension(Dimension dimension)
{
    string message = "Dimension : ";
    // Get Dimension name
    message += "\nDimension name is : " + dimension.Name;

    // Get Dimension Curve
    Autodesk.Revit.DB.Curve curve = dimension.Curve;
    if (curve != null && curve.IsBound)
    {
        // Get curve start point
        message += "\nCurve start point:(" + curve.get_EndPoint(0).X + ", "
             + curve.get_EndPoint(0).Y + ", " + curve.get_EndPoint(0).Z + ")";
        // Get curve end point
        message += "; Curve end point:(" + curve.get_EndPoint(1).X + ", "
             + curve.get_EndPoint(1).Y + ", " + curve.get_EndPoint(1).Z + ")";
    }

    // Get Dimension type name
```

```
    message += "\nDimension type name is : " + dimension.DimensionType.Name;

    // Get Dimension view name
    message += "\nDimension view name is : " + dimension.View.ViewName;

    // Get Dimension reference count
    message += "\nDimension references count is " + dimension.References.Size;

    if ((int)BuiltInCategory.OST_Dimensions == dimension.Category.Id.IntegerValue)
    {
        message += "\nDimension is a permanent dimension.";
    }
    else if ((int)BuiltInCategory.OST_Constraints == dimension.Category.Id.IntegerValue)
    {
        message += "\nDimension is a constraint element.";
    }

    TaskDialog.Show("Revit",message);
}
```

16.1.1 Dimensions

There are four kinds of permanent dimensions:

- Linear dimension
- Radial dimension
- Angular dimension
- Arc length dimension

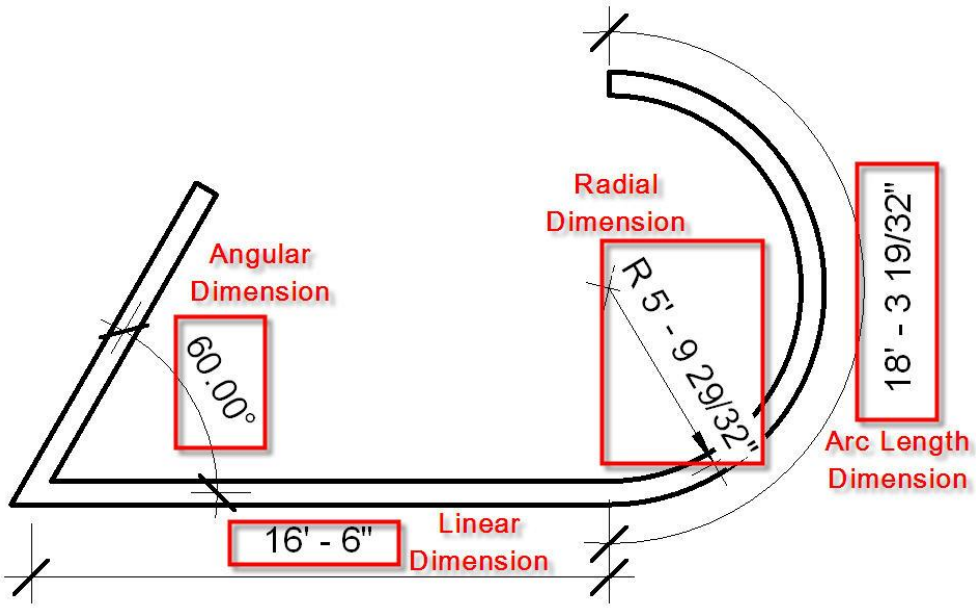


Figure 65: Permanent dimensions

The BuiltInCategory for all permanent dimensions is OST_Dimensions. There is not an easy way to distinguish the four dimensions using the API.

Except for radial dimension, every dimension has one dimension line. Dimension lines are available from the Dimension.Curve property which is always unbound. In other words, the dimension line does not have a start-point or end-point. Based on the previous picture:

- A Line object is returned for a linear dimension.
- An arc object is returned for a radial dimension or angular dimension.
- A radial dimension returns null.

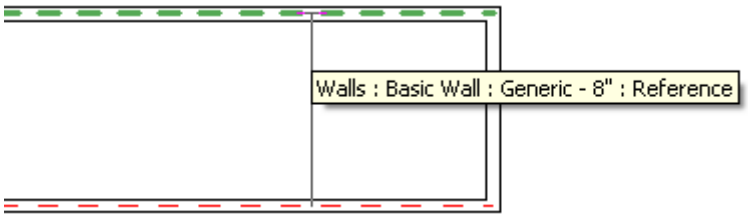


Figure 66: Dimension references

A dimension is created by selecting geometric references as the previous picture shows. Geometric references are represented as a Reference class in the API. The following dimension references are available from the References property. For more information about Reference, please see the Geometry.Reference section in the Geometry chapter.

- Radial dimension - One Reference object for the curve is returned
- Angular and arc length dimensions - Two Reference objects are returned.
- Linear dimensions - Two or more Reference objects are returned. In the following picture, the linear dimension has five Reference objects.

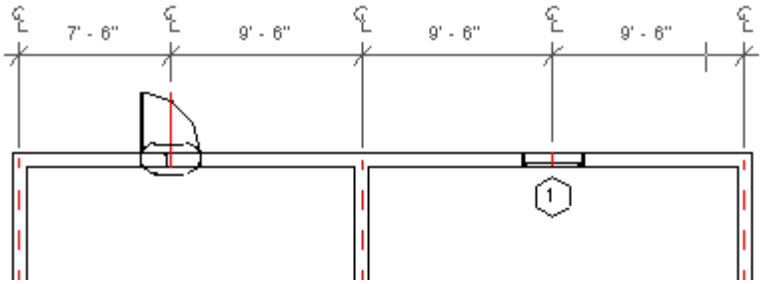


Figure 67: Linear dimension references

Dimensions, like other Annotation Elements, are view-specific. They display only in the view where they are added. The Dimension.View property returns the specific view.

16.1.2 Constraint Elements

Dimension objects with Category Constraints (BuitInCategory.OST_Constraints) represent two kinds of dimension-related constraints:

- Linear and radial dimension constraints
- Equality constraints

In the following picture, two kinds of locked constraints correspond to linear and radial dimension. In the application, they appear as padlocks with green dashed lines. (The green dashed line is available from the Dimension.Curve property.) Both linear and radial dimension constraints return two Reference objects from the Dimension.References property.

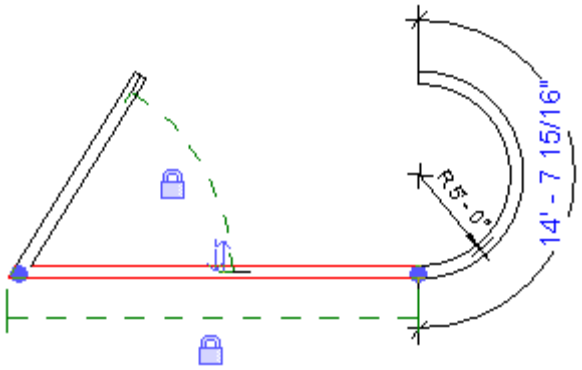


Figure 68: Linear and Radial dimension constraints

Constraint elements are not view-specific and can display in different views. Therefore, the View property always returns null. In the following picture, the constraint elements in the previous picture are also visible in the 3D view.

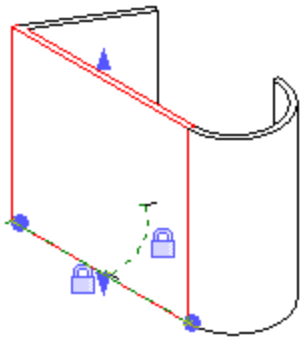


Figure 69: Linear and Radial dimension constraints in 3D view

Although equality constraints are based on dimensions, they are also represented by the Dimension class. There is no direct way to distinguish linear dimension constraints from equality constraints in the API using a category or DimensionType. Equality constraints return three or more References while linear dimension constraints return two or more References.

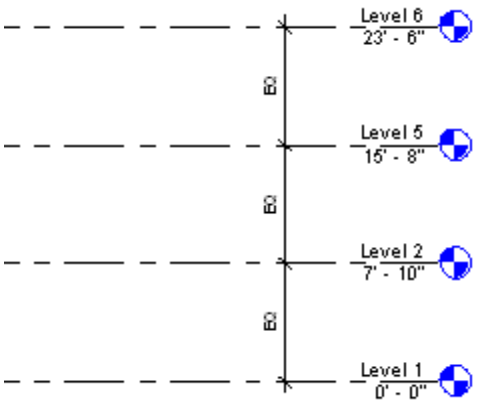


Figure 70: Equality constraints

Note: Not all constraint elements are represented by the Dimension class but all belong to a Constraints (OST_Constraints) category such as alignment constraint.

16.1.3 Spot Dimensions

Spot coordinates and spot elevations are represented by the SpotDimension class and are distinguished by category. Like the permanent dimension, spot dimensions are view-specific. The type and category for each spot dimension are listed in the following table:

Type	Category
Spot Coordinates	OST_SpotCoordinates
Spot Elevations	OST_SpotElevations

Table 34: Spot dimension Type and Category

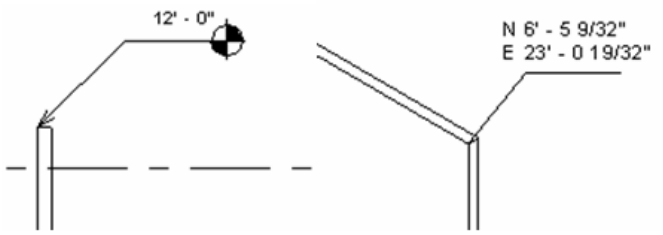


Figure 71: SpotCoordinates and SpotElevations

The SpotDimension Location can be downcast to LocationPoint so that the point coordinate that the spot dimension points to is available from the LocationPoint.Point property.

- SpotDimensions have no dimension curve so their Curve property always returns null.
- The SpotDimension References property returns one Reference representing the point or the edge referenced by the spot dimension.
- To control the text and tag display style, modify the SpotDimension and SpotDimensionType Parameters.

16.1.4 Comparison

The following table compares different kinds of dimensions and constraints in the API:

Dimension or Constraint		API Class	BuiltInCategory	Curve	Reference	View	Location
Permanent Dimension	linear dimension	Dimension	OST_Dimensions	A Line	>=2	Specific view	null
				Null	1		
	radial dimension			An Arc	2		
	angular dimension			An Arc	2		
	arc length dimension		OST_Constraints	A Line	2		
				An Arc	2		
Dimension Constraint	linear dimension constraint			A Line	>=3		
	angular dimension						
Equality Constraint							
Spot Dimension	Spot Coordinates	SpotDimension	OST_SpotCoordinates	Null	1	Specific view	LocationPoint
			OST_SpotElevations				

Dimension or Constraint		API Class	BuiltInCategory	Curve	Reference	View	Location
	Spot Elevations						

Table 35: Dimension Category Comparison

16.1.5 Create and Delete

The NewDimension() method is available in the Creation.Document class. This method can create a linear dimension only.

Code Region 16-2: NewDimension()

```
public Dimension NewDimension (View view, Line line, ReferenceArray references)
```

```
public Dimension NewDimension (View view, Line line, ReferenceArray references,
DimensionType dimensionType)
```

Using the NewDimension() method input parameters, you can define the visible View, dimension line, and References (two or more). However, there is no easy way to distinguish a linear dimension DimensionType from other types. The overloaded NewDimension() method with the DimensionType parameter is rarely used.

The following code illustrates how to use the NewDimension() method to duplicate a dimension.

Code Region 16-3: Duplicating a dimension with NewDimension()

```
public void DuplicateDimension(Document document, Dimension dimension)
{
    Line line = dimension.Curve as Line;
    if (null != line)
    {
        Autodesk.Revit.DB.View view = dimension.View;
        ReferenceArray references = dimension.References;
        Dimension newDimension = document.Create.NewDimension(view, line, references);
    }
}
```

Though only linear dimensions are created, you can delete all dimensions and constraints represented by Dimension and SpotDimension using the Document.Delete() method.

16.2 Detail Curve

Detail curve is an important Detail component usually used in the detail or drafting view. Detail curves are accessible in the DetailCurve class and its derived classes.

DetailCurve is view-specific as are other annotation elements. However, there is no DetailCurve.View property. When creating a detail curve, you must compare the detail curve to the model curve view.

Code Region 16-4: NewDetailCurve() and NewModelCurve()

```
public DetailCurve NewDetailCurve (View, Curve, SketchPlane)
```

```
public ModelCurve NewModelCurve (Curve, SketchPlane)
```

Generally only 2D views such as level view and elevation view are acceptable, otherwise an exception is thrown.

Except for view-related features, DetailCurve is very similar to ModelCurve. For more information about ModelCurve properties and usage, see the ModelCurve section in the Sketching chapter.

16.3Tags

A tag is an annotation used to identify drawing elements. The API exposes the IndependentTag and RoomTag classes to cover most tags used in the Revit application. For more details about RoomTag, see the Room section in the Revit Architecture chapter.

Note: The IndependentTag class represents the tag element in Revit and other specific tags such as keynote, beam system tag, electronic circuit symbol (Revit MEP), and so on. In Revit internal code, the specific tags have corresponding classes derived from IndependentTag. As a result, specific features are not exposed by the API and cannot be created using the NewTag() method. They can be distinguished by the following categories:

Tag Name	BuiltInCategory
Keynote Tag	OST_KeynoteTags
Beam System Tag	OST_BeamSystemTags
Electronic Circuit Tag	OST_ElectricalCircuitTags
Span Direction Tag	OST_SpanDirectionSymbol
Path Reinforcement Span Tag	OST_PathReinSpanSymbol
Rebar System Span Tag	OST_IOSRebarSystemSpanSymbolCtrl

Table 36: Tag Name and Category

In this section, the main focus is on the tag type represented in the following picture.

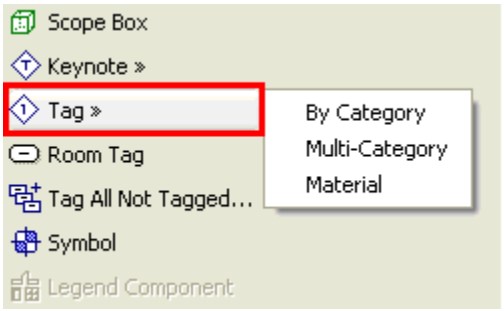


Figure 72: IndependentTag

Every category in the family library has a pre-made tag. Some tags are automatically loaded with the default Revit application template, while others are loaded manually. The IndependentTag objects return different categories based on the host element if it is created using the By Category option. For example, the Wall and Floor IndependentTag are respectively OST_WallTags and OST_FloorTags.

If the tag is created using the Multi-Category or Material style, their categories are respectively OST_MultiCategoryTags and OST_MaterialTags.

Similar to DetailCurve, NewTag() only works in the 2D view, otherwise an exception is thrown. The following code is an example of IndependentTag creation. Run it when the level view is the active view.

Note: You can't change the text displayed in the IndependentTag directly. You need to change the parameter that is used to populate tag text in the Family Type for the Element that's being tagged. In the example below, that parameter is "Type Mark", although this setting can be changed in the Family Editor in the Revit UI.

Code Region 16-5: Creating an IndependentTag

```
private IndependentTag CreateIndependentTag(Autodesk.Revit.Document document, Wall wall)
{
    // make sure active view is not a 3D view
    Autodesk.Revit.DB.Vie view = document.ActiveView;

    // define tag mode and tag orientation for new tag
    TagMode tagMode = TagMode.TM_ADDBY_CATEGORY;
    TagOrientation tagorn = TagOrientation.TAG_HORIZONTAL;

    // Add the tag to the middle of the wall
    LocationCurve wallLoc = wall.Location as LocationCurve;
    XYZ wallStart = wallLoc.Curve.get_EndPoint(0);
    XYZ wallEnd = wallLoc.Curve.get_EndPoint(1);
    XYZ wallMid = wallLoc.Curve.Evaluate(0.5, true);

    IndependentTag newTag = document.Create.NewTag(view, wall, true, tagMode, tagorn,
                                                   wallMid);
    if (null == newTag)
    {
        throw new Exception("Create IndependentTag Failed.");
    }

    // newTag.TagText is read-only, so we change the Type Mark type parameter to
    // set the tag text.  The label parameter for the tag family determines
    // what type parameter is used for the tag text.

    WallType type = wall.WallType;

    Parameter foundParameter = type.get_Parameter("Type Mark");
    bool result = foundParameter.Set("Hello");

    // set leader mode free
    // otherwise leader end point move with elbow point

    newTag.LeaderMode = LeaderEndCondition.LEC_FREE;
    XYZ elbowPnt = wallMid + new XYZ(5.0, 5.0, 0.0);
    newTag.LeaderElbow = elbowPnt;
    XYZ headerPnt = wallMid + new XYZ(10.0, 10.0, 0.0);
```

```
    newTag.TagHeadPosition = headerPnt;

    return newTag;
}
```

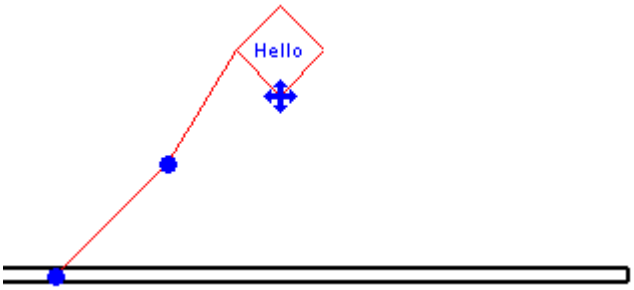


Figure 73: Create IndependentTag using sample code

16.4Text

In the API, the TextNote class represents Text. Its general features are as follows:

Function	API Method or Property
Add text to the application	Document.Create.NewTextNote() or Document.Create.NewTextNotes() methods
Get and set string from the text component	TextNote.Text property
Get and set text component position	TextNote.Coord Property
Get and set text component width	TextNote.Width Property
Get all text component leaders	TextNote.Leaders Property
Add a leader to the text component	TextNote.AddLeader() Method
Remove all leaders from the text component	TextNote.RemoveLeaders() Method

Table 37: General Features of TextNote

Revit supports two kinds of Leaders: straight leaders and arc leaders. Control the TextNote leader type using the TextNoteLeaderType enumerated type:

Function	Member Name
 —Add a right arc leader	TNLT_ARC_R
 —Add a left arc leader	TNLT_ARC_L
 —Add a right leader.	TNLT_STRAIGHT_R
 —Add a left leader.	TNLT_STRAIGHT_L

Table 38: Leader Types

Note: Straight leaders and arc leaders cannot be added to a Text type at the same time.

16.5 Annotation Symbol

An annotation symbol is a symbol applied to a family to uniquely identify that family in a project.

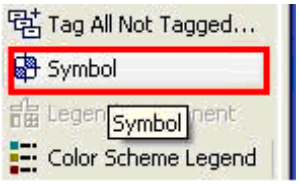


Figure 74: Add annotation symbol

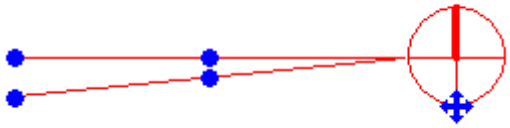


Figure 75: Annotation Symbol with two leaders

16.5.1 Create and Delete

The annotation symbol creation method is available in the Creation.Document class.

Code Region 16-6: NewAnnotationSymbol()

```
public AnnotationSymbol NewAnnotationSymbol(XYZ location, AnnotationSymbolType
annotationSymbolType, View specView)
```

The annotation symbol can be deleted using the Document.Delete() method.

16.5.2 Add and Remove Leader

Add and remove leaders using the addLeader() and removeLeader() methods.

Code Region 16-7: Using addLeader() and removeLeader()

```
public void AddAndRemoveLeaders(AnnotationSymbol symbol)
{
    int leaderSize = symbol.Leaders.Size;
    string message = "Number of leaders in Annotation Symbol before add: " + leaderSize;
    symbol.addLeader();
    leaderSize = symbol.Leaders.Size;
    message += "\nNumber of leaders after add: " + leaderSize;
    symbol.removeLeader();
    leaderSize = symbol.Leaders.Size;
    message += "\nNumber of leaders after remove: " + leaderSize;

    TaskDialog.Show("Revit", message);
}
```

17 Sketching

To create elements or edit their profiles in Revit, you must first create sketch objects. Examples of elements that require sketches include:

- Roofs
- Floors
- Stairs
- Railings.

Sketches are also required to define other types of geometry, such as:

- Extrusions
- Openings
- Regions

In the Revit Platform API, sketch functions are represented by 2D and 3D sketch classes such as the following:

- 2D sketch:
  - SketchPlane
  - Sketch
  - ModelCurve
  - and more
- 3D sketch:
  - GenericForm
  - Path3D

In addition to Sketch Elements, ModelCurve is also described in this chapter. For more details about Element Classification, see the Elements Classification section in the Elements Essentials chapter.

17.1 The 2D Sketch Class

The Sketch class represents enclosed curves in a plane used to create a 3D model. The key features are represented by the SketchPlane and CurveLoop properties.

When editing a Revit file, you cannot retrieve a Sketch object by iterating Document.Elements enumeration because all Sketch objects are transient Elements. When accessing the Family's 3D modeling information, Sketch objects are important to forming the geometry. For more details, refer to the 3D Sketch section in this chapter.

SketchPlane is the basis for all 2D sketch classes such as ModelCurve and Sketch. SketchPlane is also the basis for 2D Annotation Elements such as DetailCurve. Both ModelCurve and DetailCurve have the SketchPlane property and need a SketchPlane in the corresponding creation method. SketchPlane is always invisible in the Revit UI.

Every ModelCurve must lie in one SketchPlane. In other words, wherever you draw a ModelCurve either in the UI or by using the API, a SketchPlane must exist. Therefore, at least one SketchPlane exists in a 2D view where a ModelCurve is drawn.

The 2D view contains the CeilingPlan, FloorPlan, and Elevation ViewTypes. By default, a SketchPlane is automatically created for all of these views. The 2D view-related SketchPlane Name returns the view name such as Level 1 or North.

Figure 76: Pick a Plane to identify a new Work Plane

When you specify a new work plane, you can select Pick a plane as illustrated in the previous picture. After you pick a plane, select a plane on a particular element such as a wall as the following picture shows. In this case, the SketchPlane.Name property returns a string related to that element. For example, in the following picture, the SketchPlane.Name property returns 'Generic - 8' the same as the Wall.Name property.

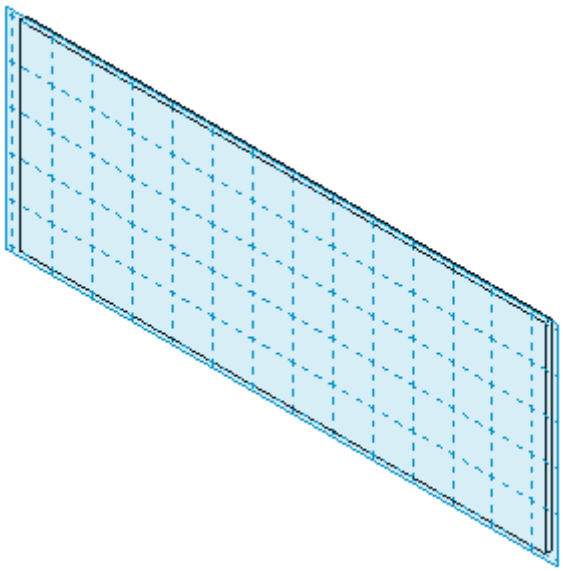


Figure 77: Pick a Plane on a wall as Work Plane

Note: A SketchPlane is different from a work plane because a work plane is visible and can be selected. It does not have a specific class in the current API, but is represented by the Element class. A work plane must be defined based on a specific SketchPlane. Both the work plane and SketchPlane Category property return null. Although SketchPlane is always

invisible, there is always a SketchPlane that corresponds to a work plane. A work plane is used to express a SketchPlane in text and pictures.

The following information applies to SketchPlane members:

- ID, UniqueId, Name, and Plane properties return a value;
- Parameters property is empty
- Location property returns a Location object
- Other properties return null.

Plane contains the SketchPlane geometric information. SketchPlane sets up a plane coordinate system with Plane as the following picture illustrates:

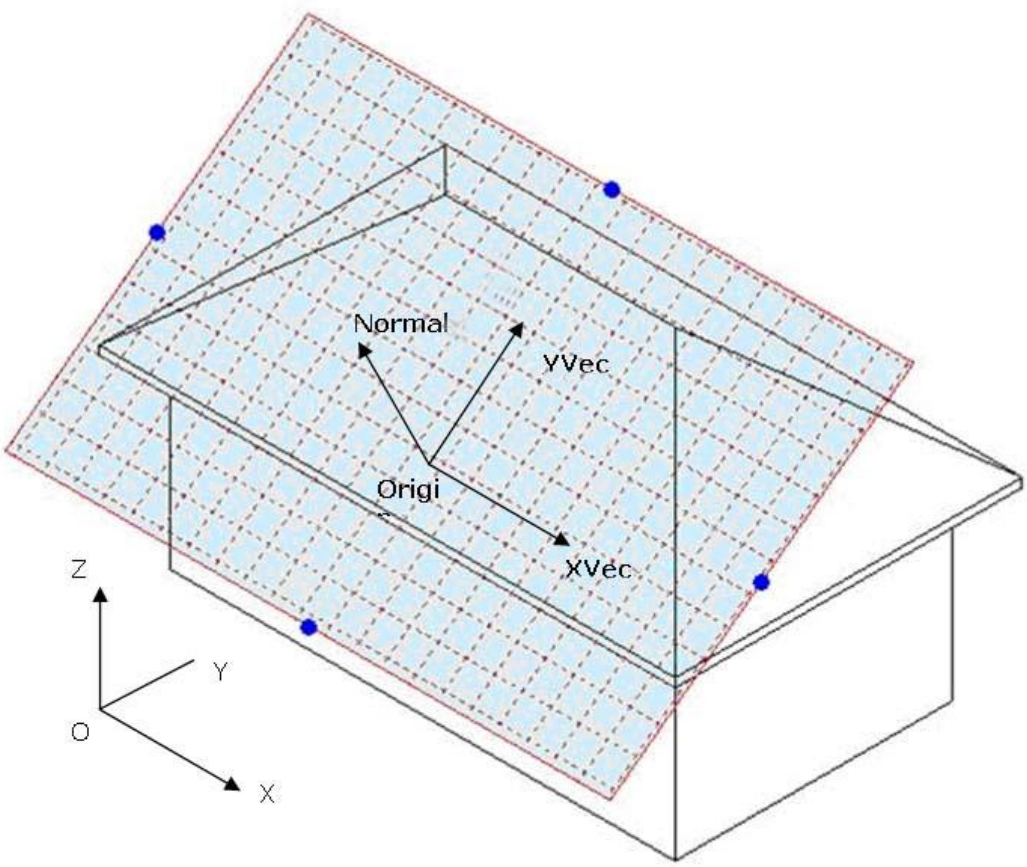


Figure 78: SketchPlane and Plane coordinate system

The following code sample illustrates how to create a new SketchPlane:

Code Region 17-1: Creating a new SketchPlane

```
private SketchPlane CreateSketchPlane(UIApplication application)
{
    //try to create a new sketch plane
    XYZ newNormal = new XYZ(1, 1, 0);  // the normal vector
    XYZ newOrigin = new XYZ(0, 0, 0);  // the origin point
    // create geometry plane
    Plane geometryPlane = application.Application.Create.NewPlane(newNormal, newOrigin);

    // create sketch plane
```

```
    SketchPlane sketchPlane =
        application.ActiveUIDocument.Document.Create.NewSketchPlane(geometryPlane);

    return sketchPlane;
}
```

17.23D Sketch

3D Sketch is used to edit a family or create a 3D object. In the Revit Platform API, you can complete the 3D Sketch using the following classes.

- Extrusion
- Revolution
- Blend
- Sweep

In other words, there are four operations through which a 2D model turns into a 3D model. For more details about sketching in 2D, refer to the 2D Sketch section in this chapter.

17.2.1**Extrusion**

Revit uses extrusions to define 3D geometry for families. You create an extrusion by defining a 2D sketch on a plane; Revit then extrudes the sketch between a start and an end point.

Query the Extrusion Form object for a generic form to use in family modeling and massing. The Extrusion class has the following properties:

Property	Description
ExtrusionStart	Returns the Extrusion Start point. It is a Double type.
ExtrusionEnd	Returns the Extrusion End point. It is a Double type.
Sketch	Returns the Extrusion Sketch. It contains a sketch plane and some curves.

Table 39: Extrusion Properties

The value of the ExtrusionStart and ExtrusionEnd properties is consistent with the parameters in the Revit UI. The following figure illustrates the corresponding parameters and the Extrusion result.

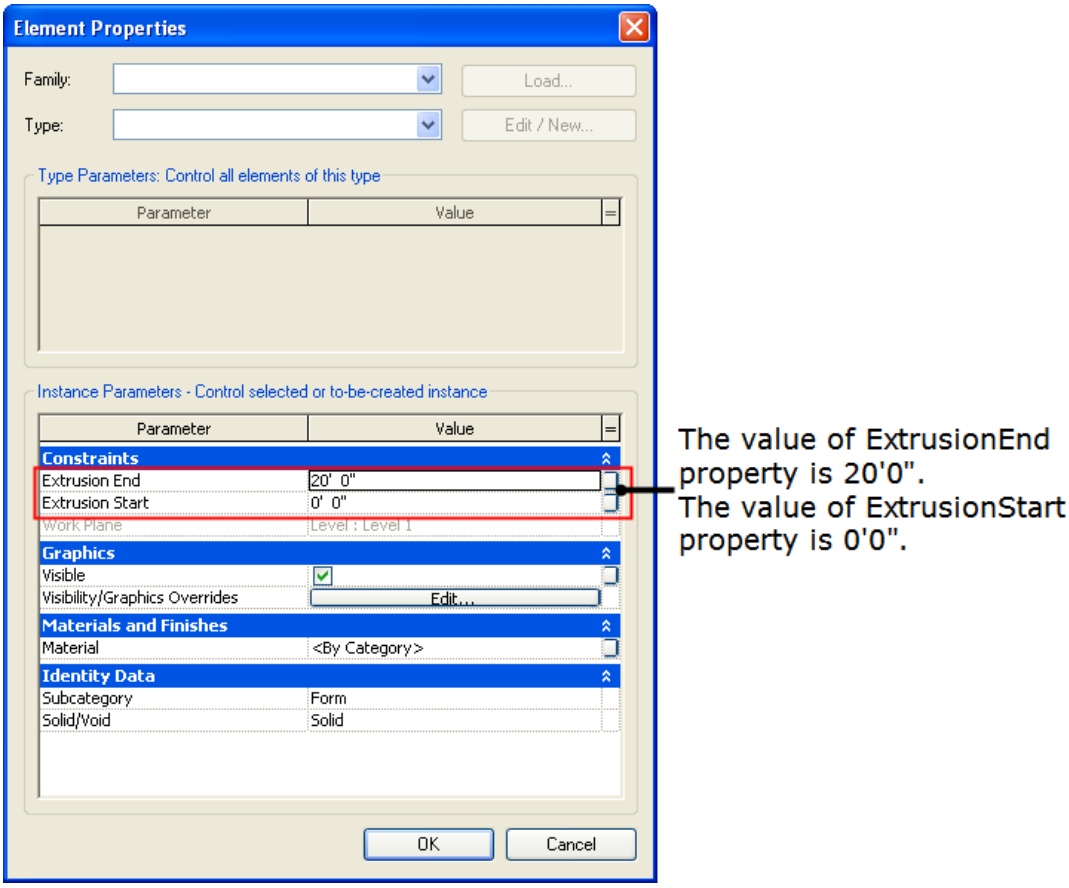


Figure 79: Extrusion parameters in the UI

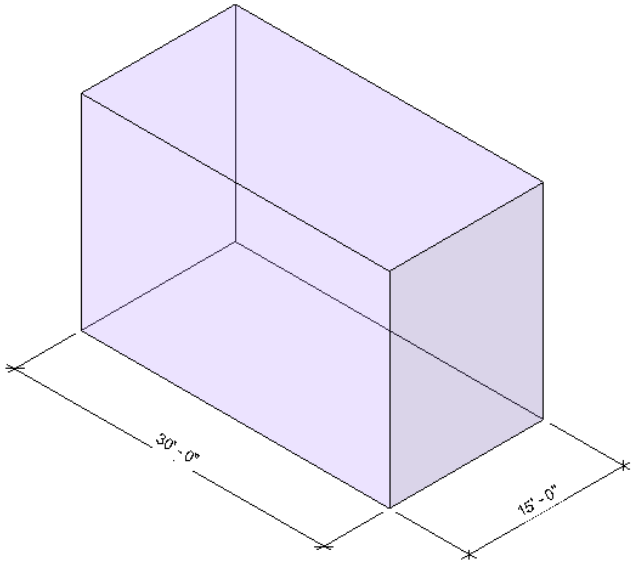


Figure 80: Extrusion result

17.2.2**Revolution**

The Revolve command creates geometry that revolves around an axis. You can use the revolve command to create door knobs or other knobs on furniture, a dome roof, or columns.

Query the Revolution Form object for a generic form to use in family modeling and massing. The Revolution class has the following properties:

Property	Description
Axis	Returns the Axis. It is a ModelLine object.
EndAngle	Returns the End Angle. It is a Double type.
Sketch	Returns the Extrusion Sketch. It contains a SketchPlane and some curves.

Table 40: Revolution Properties

EndAngle is consistent with the same parameter in the Revit UI. The following pictures illustrate the Revolution corresponding parameter, the sketch, and the result.

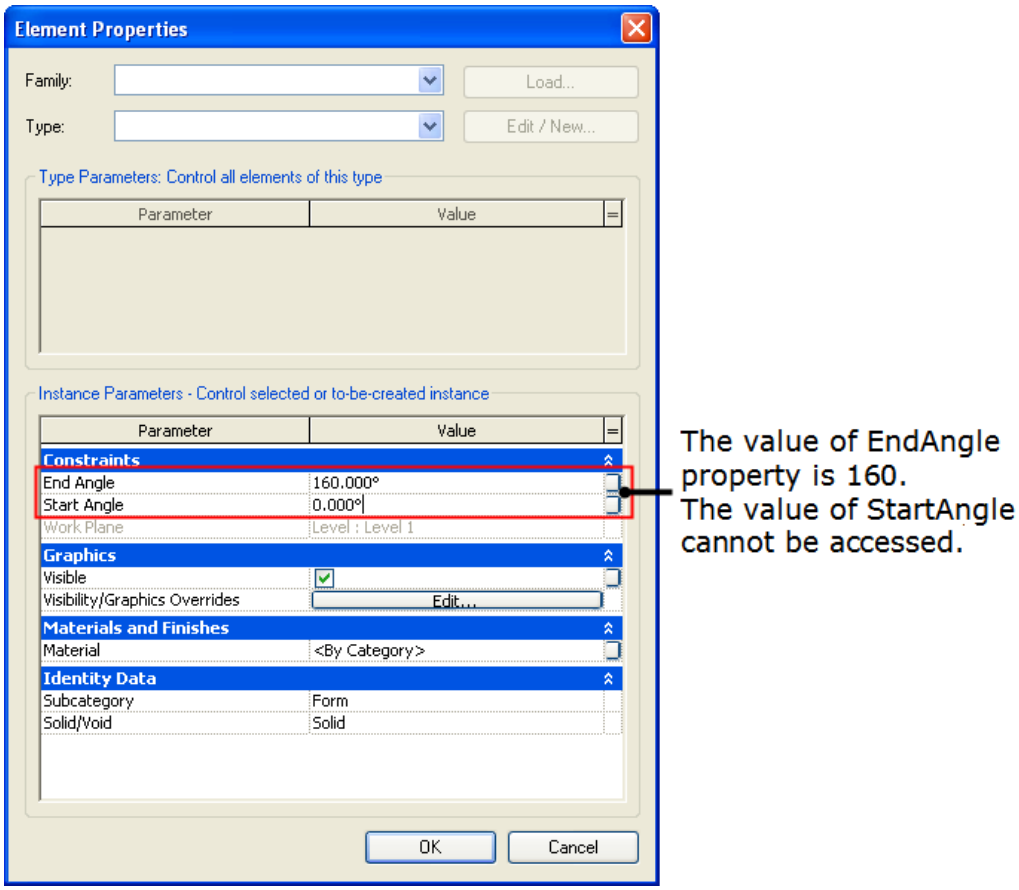


Figure 81: Corresponding parameter

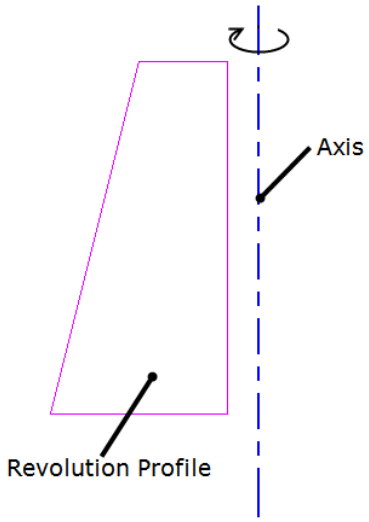


Figure 82: Revolution sketch

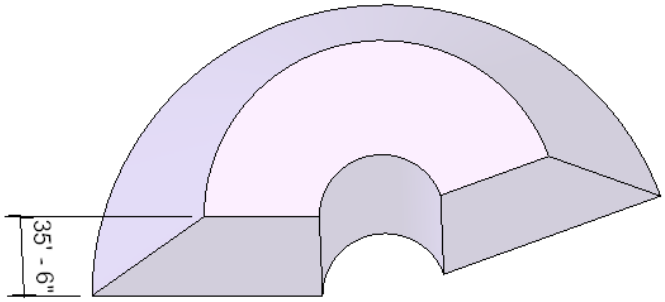


Figure 83: Revolution result

Note:

- The Start Angle is not accessible using the Revit Platform API.
- If the End Angle is positive, the Rotation direction is clockwise. If it is negative, the Rotation direction is counterclockwise

17.2.3**Blend**

The Blend command blends two profiles together. For example, if you sketch a large rectangle and a smaller rectangle on top of it, Revit Architecture blends the two shapes together.

Query the Blend Form object for a generic form to use in family modeling and massing. The Blend class has the following properties:

Property	Description
BottomSketch	Returns the Bottom Sketch. It is a Sketch object.
TopSketch	Returns the Top Sketch Blend. It is a Sketch object.
FirstEnd	Returns the First End. It is a Double type.
SecondEnd	Returns the Second End. It is a Double type.

Table 41: Blend Properties

The FirstEnd and SecondEnd property values are consistent with the same parameters in the Revit UI. The following pictures illustrate the Blend corresponding parameters, the sketches, and the result.

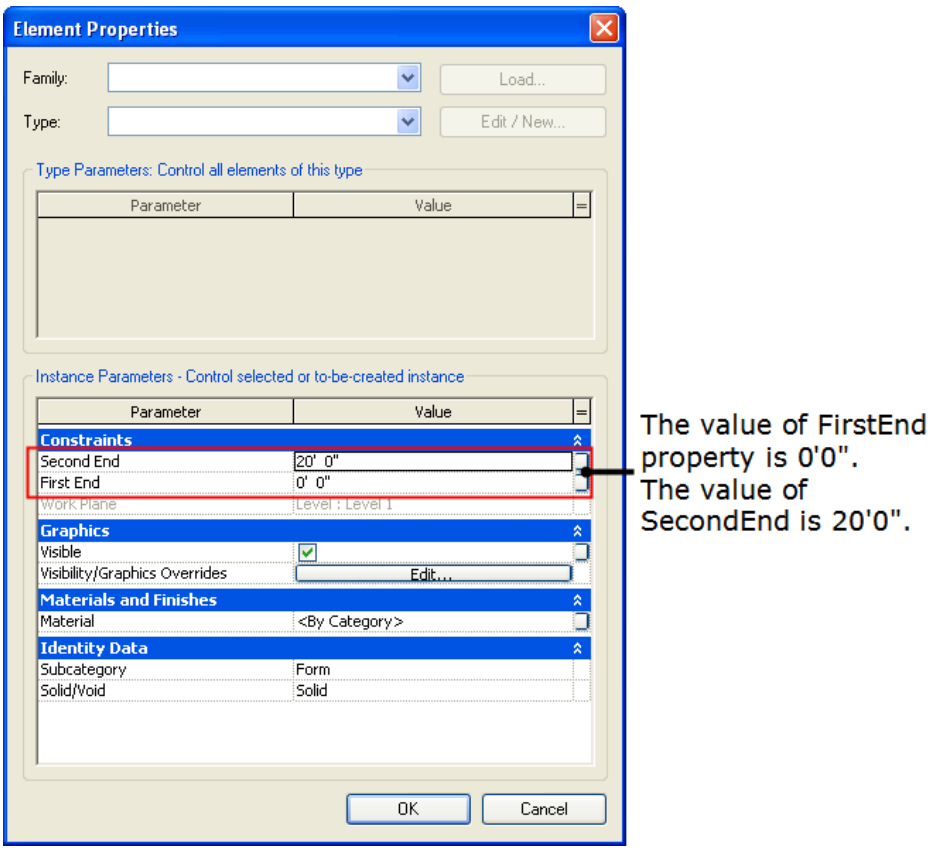


Figure 84: Blend parameters in the UI

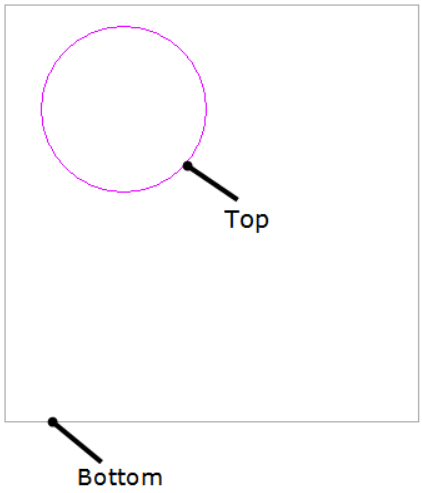


Figure 85: Blend top sketch and bottom sketch

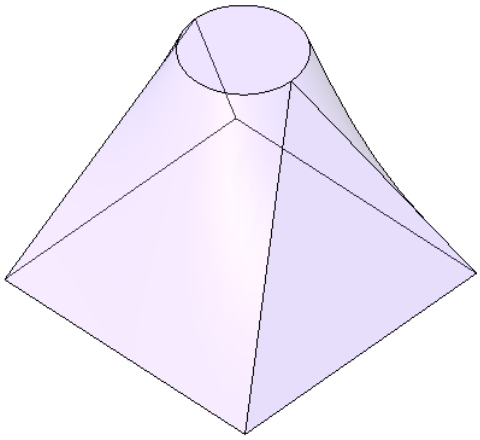


Figure 86: Blend result

17.2.4**Sweep**

The Sweep command sweeps one profile along a created 2D path or selected 3D path. The path may be an open or closed loop, but must pierce the profile plane.

Query the Sweep Form object for a generic form for use in family modeling and massing. The Sweep class has the following properties:

Property	Description
Path3d	Returns the 3D Path Sketch. It is a Path3D object.
PathSketch	Returns the Plan Path Sketch. It is a Sketch object.
ProfileSketch	Returns the profile Sketch. It is a Sketch object.
EnableTrajSegmentation	Returns the Trajectory Segmentation state. It is a Boolean.
MaxSegmentAngle	Returns the Maximum Segment Angle. It is a Double type.

Table 42: Sweep Properties

Creating a 2D Path is similar to other forms. The 3D Path is fetched by picking the created 3D curves.

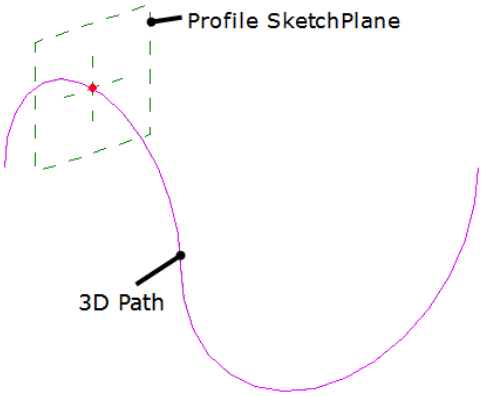


Figure 87: Pick the Sweep 3D path

Note: The following information applies to Sweep:

- The Path3d property is available only when you use Pick Path to get the 3D path.
- PathSketch is available whether the path is 3D or 2D.

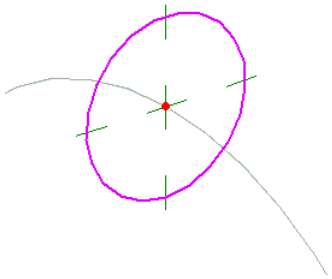


Figure 88: Sweep profile sketch

Note: The ProfileSketch is perpendicular to the path.

Segmented sweeps are useful for creating mechanical duct work elbows. Create a segmented sweep by setting two sweep parameters and sketching a path with arcs.

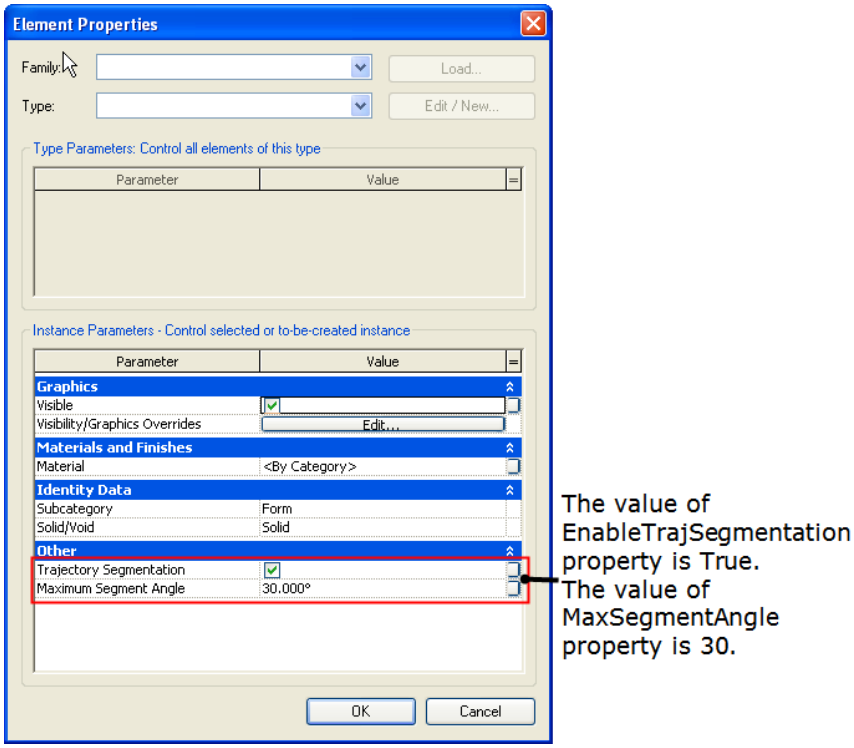


Figure 89: Corresponding segment settings in the UI

Note: The following information applies to segmented Sweeps:

- The parameters affect only arcs in the path.
- The minimum number of segments for a sweep is two.
- Change a segmented sweep to a non-segmented sweep by clearing the Trajectory Segmentation check box. The EnableTrajSegmentation property returns false.
- If the EnableTrajSegmentation property is false, the value of MaxSegmentAngle is the default 360°.

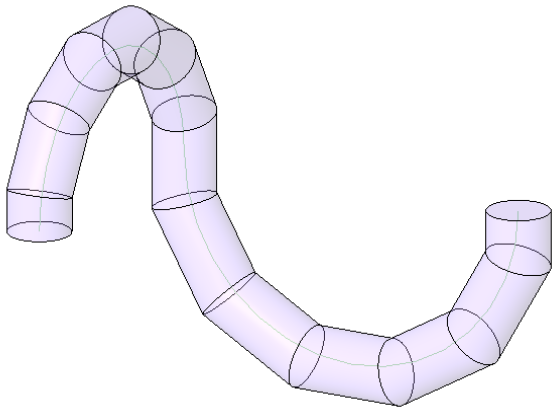


Figure 90: Sweep result

17.3 ModelCurve

ModelCurve represents model lines in the project. It exists in 3D space and is visible in all views.

The following pictures illustrate the four ModelCurve derived classes:

Figure 91:ModelLine and ModelArc

Figure 92: ModelEllipse and ModelNurbSpline

17.3.1 Creating a ModelCurve

The key to creating a ModelCurve is to create the Geometry.Curve and SketchPlane where the Curve is located. Based on the Geometry.Curve type you input, the corresponding ModelCurve returned can be downcast to its correct type.

The following sample illustrates how to create a new model curve (ModelLine and ModelArc):

Code Region 17-2: Creating a new model curve

```
// get handle to application from document
Autodesk.Revit.ApplicationServices.Application application = document.Application;

// Create a geometry line in Revit application
```

```
XYZ startPoint = new XYZ(0, 0, 0);
XYZ endPoint = new XYZ(10, 10, 0);
Line geomLine = application.Create.NewLine(startPoint, endPoint, true);

// Create a geometry arc in Revit application
XYZ end0 = new XYZ(1, 0, 0);
XYZ end1 = new XYZ(10, 10, 10);
XYZ pointOnCurve = new XYZ(10, 0, 0);
Arc geomArc = application.Create.NewArc(end0, end1, pointOnCurve);

// Create a geometry plane in Revit application
XYZ origin = new XYZ(0, 0, 0);
XYZ normal = new XYZ(1, 1, 0);
Plane geomPlane = application.Create.NewPlane(normal, origin);

// Create a sketch plane in current document
SketchPlane sketch = document.Create.NewSketchPlane(geomPlane);

// Create a ModelLine element using the created geometry line and sketch plane
ModelLine line = document.Create.NewModelCurve(geomLine, sketch) as ModelLine;

// Create a ModelArc element using the created geometry arc and sketch plane
ModelArc arc = document.Create.NewModelCurve(geomArc, sketch) as ModelArc;
```

17.3.2 **Properties**

ModelCurve has properties that help you set specific GeometryCurves. In this section, the GeometryCurve and LineStyle properties are introduced.

17.3.2.1 **GeometryCurve**

The GeometryCurve property is used to get or set the model curve's geometry curve. Except for ModelHermiteSpline, you can get different Geometry.Curves from the four ModelCurves;

- Line
- Arc
- Ellipse
- Nurbspline.

The following code sample illustrates how to get a specific Curve from a ModelCurve.

Code Region 17-3: Getting a specific Curve from a ModelCurve

```
//get the geometry modelCurve of the model modelCurve
Autodesk.Revit.DB.Curve geoCurve = modelCurve.GeometryCurve;

if (geoCurve is Autodesk.Revit.DB.Line)
{
    Line geoLine = geoCurve as Line;
}
```

The GeometryCurve property return value is a general Geometry.Curve object, therefore, you must use an As operator to convert the object type.

Note: The following information applies to GeometryCurve:

- In Revit you cannot create a Hermite curve but you can import it from other software such as AutoCAD. Geometry.Curve is the only geometry class that represents the Hermite curve.
- The SetPlaneAndCurve() method and the Curve and SketchPlane property setters are used in different situations.
    - When the new Curve lies in the same SketchPlane, or the new SketchPlane lies on the same planar face with the old SketchPlane, use the Curve or SketchPlane property setters.
    - If new Curve does not lay in the same SketchPlane, or the new SketchPlane does not lay on the same planar face with the old SketchPlane, you must simultaneously change the Curve value and the SketchPlane value using SetPlaneAndCurve() to avoid internal data inconsistency.

17.3.2.2 **LineStyle**

Line style does not have a specific class but is represented by the Document.Element class.

- All line styles for a ModelCurve are available from the LineStyles property in the Element Properties dialog box.
- The Element object that represents line style cannot provide information for all properties. Most properties will return null or an empty collection.
- The only information returned is the following:
    - Id
    - UniqueId
    - Name.
- Use the getLineStyle() method to retrieve the current line style or use the setLineStyle() method to set the current line style to one returned from the LineStyles property.

18 Views

Views are images produced from a Revit model with privileged access to the data stored in the documents. They can be graphics, such as plans, or text, such as schedules. Each project document has one or more different views. The last focused window is the active view.

In this chapter, you learn how views are generated, the types of views supported by Revit, and the features for each view.

18.1 Overview

This section is a high-level overview discussing the following:

- How views are generated
- View types
- Element visibility
- Creating and deleting views.

18.1.1 View Process

The following figure illustrates how a view is generated.

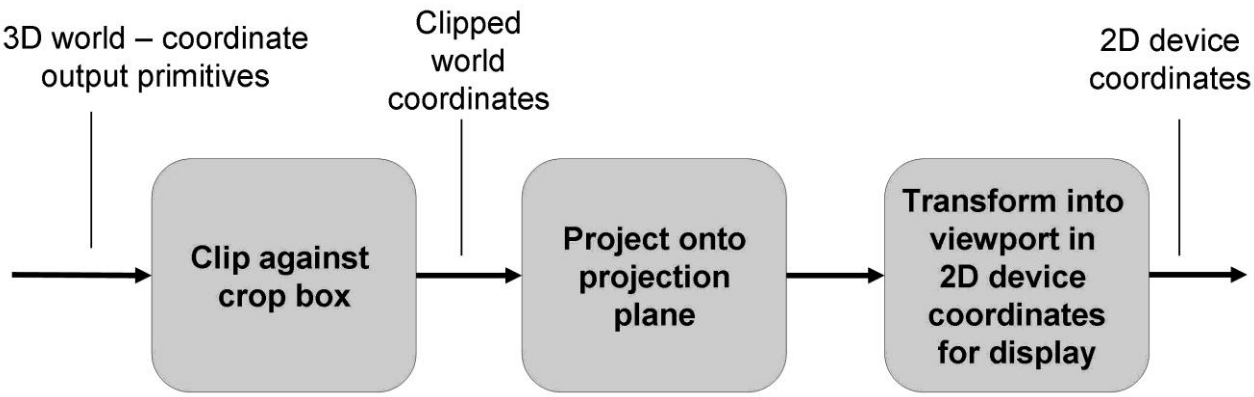


Figure 93: Create view process

Each view is generated by projecting a three-dimensional object onto a two-dimensional projection plane. Projections are divided into two basic classes:

- Perspective
- Parallel

After the projection type is determined, you must specify the conditions under which the 3D model is needed and the scene is to be rendered. For more information about projection, refer to the View3D section.

World coordinates include the following:

- The viewer's eye position
- The viewing plane location where the projection is displayed.

Revit uses two coordinate systems:

- The global or model space coordinates where the building exists
- The viewing coordinate system.

The viewing coordinate system represents how the model is presented in the observer's view. Its origin is the viewer's eye position whose coordinates in the model space are retrieved by the View.Origin property. The X, Y, and Z axes are represented by the View.RightDirection, View.UpDirection, and View.ViewDirection properties respectively.

- View.RightDirection is towards the right side of the screen.
- View.UpDirection towards the up side of the screen.
- View.ViewDirection from the screen to the viewer.

The viewing coordinate system is right-handed. For more information, see the Perspective Projection picture and the Parallel Projection picture in the View3D section in this chapter.

Some portions of a 3D model space that do not display, such as those that are behind the viewer or too far away to display clearly, are excluded before being projected onto the projection plane. This action requires cropping the view. The following rules apply to cropping:

- Elements outside of the crop box are no longer in the view.
- The View.CropBox property provides the geometry information for the box. It returns an instance of BoundingBoxXYZ indicating a rectangular parallelepiped in the viewing coordinate system. The coordinate system and the crop box shape are illustrated in the View3D section.
- The View.CropBoxVisible property determines whether the crop box is visible in the view.
- The View.CropBoxActive property determines whether the crop box is actually being used to crop the view.

After cropping, the model is projected onto the projection plane. The following rules apply to the projection:

- The projection contents are mapped to the screen view port for display.
- During the mapping process, the projection contents are scaled so that they are shown properly on the screen.
- The View.Scale property is the ratio of the actual model size to the view size.
- The view boundary on paper is the crop region, which is a projection of the crop box on the projection plane.
- The size and position of the crop region is determined by the View.OutLine property.

18.1.2**View Types**

A project model can have several view types. The following picture demonstrates the different types of views in the Project browser.

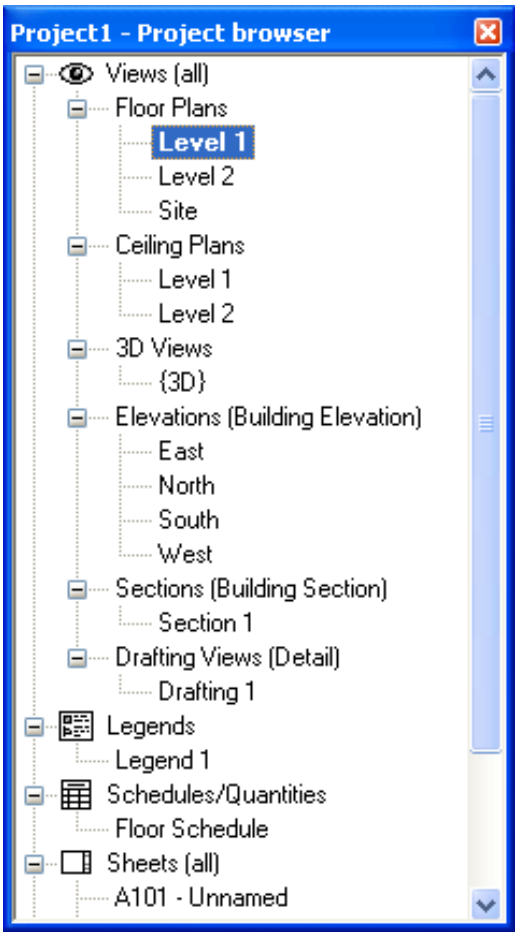


Figure 94: Different views in the Project browser

In the API, there are two ways to classify views. The first way is by using the view element View.ViewType property. It returns an enumerated value indicating the view type. The following table lists all available view types.

Member Name	Description
AreaPlan	Area view.
CeilingPlan	Reflected ceiling plan view.
ColumnSchedule	Coulmn schedule view.
CostReport	Cost report view.
Detail	Detail view.
DraftingView	Drafting view.
DrawingSheet	Drawing sheet view.
Elevation	Elevation view.
EngineeringPlan	Engineering view.
FloorPlan	Floor plan view.
Internal	Revit's internal view.
Legend	Legend view.
LoadsReport	Loads report view.
PanelSchedule	Panel schedule view.

Member Name	Description
PressureLossReport	Pressure Loss Report view.
Rendering	Rendering view.
Report	Report view.
Schedule	Schedule view.
Section	Cross section view.
ThreeD	3-D view.
Undefined	Undefined/unspecified view.
Walkthrough	Walkthrough view.

Table 43: Autodesk.Revit.DB.ViewType

The second way to classify views is by the class type.

The following table lists the view types and the corresponding views in the Project browser.

Project Browser Views	View Type	Class Type
Area Plans	ViewType.AreaPlan	Elements.ViewPlan
Ceiling Plans	ViewType.CeilingPlan	Elements.ViewPlan
Graphic Column Schedule	ViewType.ColumnSchedule	Elements.View
Detail Views	ViewType.Detail	Elements.ViewSection
Drafting Views	ViewType.DraftingView	Elements.ViewDrafting
Sheets	ViewType.DrawingSheet	Elements.ViewSheet
Elevations	ViewType.Elevation	Elements.View
Structural Plans (Revit Structure)	ViewType.EngineeringPlan	Elements.ViewPlan
Floor Plans	ViewType.FloorPlan	Elements.ViewPlan
Legends	ViewType.Legend	Elements.View
Reports (Revit MEP)	ViewType.LoadsReport	Elements.View
Reports (Revit MEP)	ViewType.PanelSchedule	Elements.View
Reports (Revit MEP)	ViewType.PresureLossReport	Elements.View
Renderings	ViewType.Rendering	Elements.ViewDrafting
Reports	ViewType.Report	Elements.View
Schedules/Quantities	ViewType.Schedule	Elements.View
Sections	ViewType.Section	Elements.View
3D Views	ViewType.ThreeD	Elements.View3D
Walkthroughs	ViewType.Walkthrough	Elements.View3D

Table 44: Project Browser Views

The following example shows how to get the class type of a view, in this case, the ActiveView of the Document.

Code Region 18-1: Determining the View class type

```
// Get the active view of the current document.
Autodesk.Revit.DB.View view = document.ActiveView;

// Get the class type of the active view, and format the prompt string
String prompt = "Revit is currently in ";
if (view is Autodesk.Revit.DB.View3D)
{
    prompt += "3D view.";
}
else if (view is Autodesk.Revit.DB.ViewSection)
{
    prompt += "section view.";
}
else if (view is Autodesk.Revit.DB.ViewSheet)
{
    prompt += "sheet view.";
}
else if (view is Autodesk.Revit.DB.ViewDrafting)
{
    prompt += "drafting view.";
}
else
{
    prompt += "normal view, the view name is " + view.Name;
}

// Give the user some information
TaskDialog.Show("Revit", prompt);
```

This example shows how to use the ViewType property of a view to determine the view's type.

Code Region 18-2: Determining the View type

```
public void GetViewType(Autodesk.Revit.DB.View view)
{
    // Get the view type of the given view, and format the prompt string
    String prompt = "The view is ";

    switch (view.ViewType)
    {
        case ViewType.AreaPlan:
            prompt += "an area view.";
            break;
        case ViewType.CeilingPlan:
            prompt += "a reflected ceiling plan view.";
            break;
        case ViewType.ColumnSchedule:
```

```
                prompt += "a column schedule view.";
                break;
            case ViewType.CostReport:
                prompt += "a cost report view.";
                break;
            case ViewType.Detail:
                prompt += "a detail view.";
                break;
            case ViewType.DraftingView:
                prompt += "a drafting view.";
                break;
            case ViewType.DrawingSheet:
                prompt += "a drawing sheet view.";
                break;
            case ViewType.Elevation:
                prompt += "an elevation view.";
                break;
            case ViewType.EngineeringPlan:
                prompt += "an engineering view.";
                break;
            case ViewType.FloorPlan:
                prompt += "afloor plan view.";
                break;
            // ...
            default:
                break;
        }

        // Give the user some information
        MessageBox.Show(prompt, "Revit", MessageBoxButtons.OK);
    }
```

18.1.3 Element Visibility in a View

Views keep track of visible elements. All elements that are graphical and visible in the view can be retrieved using a FilteredElementCollector constructed with a document and the id of the view. However, some elements in the set may be hidden or covered by other elements. You can see them by rotating the view or removing the elements that cover them. Accessing these visible elements may require Revit to rebuild the geometry of the view. The first time your code uses this constructor for a given view, or the first time your code uses this constructor for a view whose display settings have just been changed, you may experience a significant performance degradation.

Elements are shown or hidden in a view by category.

- The View.getVisiblility() method queries a category to determine if it is visible or invisible in the view.
- The View.setVisibility() method sets all elements in a specific category to visible or invisible.

A FilteredElementCollector based on a view will only contain elements visible in the current view. You cannot retrieve elements that are not graphical or elements that are invisible. A FilteredElementCollector based on a document retrieves all elements in the document including

invisible elements and non-graphical elements. For example, when creating a default 3D view in an empty project, there are no elements in the view but there are many elements in the document, all of which are invisible.

The following code sample counts the number of wall category elements in the active document and active view. The number of elements in the active view differs from the number of elements in the document since the document contains non-graphical wall category elements.

Code Region 18-3: Counting elements in the active view

```
private void CountElements(Autodesk.Revit.DB.Document document)
{
    StringBuilder message = new StringBuilder();
    FilteredElementCollector viewCollector =
        new FilteredElementCollector(document, document.ActiveView.Id);
    viewCollector.OfCategory(BuiltInCategory.OST_Walls);
    message.AppendLine("Wall category elements within active View: "
       + viewCollector.ToElementIds().Count);

    FilteredElementCollector docCollector = new FilteredElementCollector(document);
    docCollector.OfCategory(BuiltInCategory.OST_Walls);
    message.AppendLine("Wall category elements within document: "
       + docCollector.ToElementIds().Count);

    TaskDialog.Show("Revit", message.ToString());
}
```

18.1.4 Creating and Deleting Views

The Revit Platform API provides five methods to create the corresponding view elements derived from Autodesk.Revit.Elements.View class.

Method	Parameters
View3D NewView3D(XYZ viewDirection)	viewDirection: Vector pointing towards the viewer's eye.
ViewPlan NewViewPlan(string pViewName, Level pLevel, ViewType viewType)	pViewName: Name for the new plan view. It must be unique or a null pointer. pLevel: Level associated with the plan view. viewType: Type of plan view created. It must be Floor Plan or Ceiling Plan (Structural Plan in Structure).
ViewSection NewViewSection(BoundingBoxXYZ box)	box: View orientation and bounds. The X axis points towards the right of screen; Y - towards up; Z - towards the user.
ViewSheet NewViewSheet(FamilySymbol titleBlock)	titleBlock: The titleblock family symbol applied to the sheet.

Method	Parameters
ViewDrafting NewViewDrafting( )	

Table 45: Create View Element Methods

If a view is created successfully, these methods return a reference to the view, otherwise it returns null. The methods are described in the following sections.

Delete a view by using the Document.Delete method with the view ID. You can also delete elements associated with a view. For example, deleting the level element causes Revit to delete the corresponding plan view or deleting the camera element causes Revit to delete the corresponding 3D view.

Note: You cannot gain access to Schedule views in the Revit Platform API as there is no NewScheduleView() method.

18.2 The View3D Class

View3D is a freely-oriented three-dimensional view. There are two kinds of 3D views, perspective and orthographic. The difference is based on the projection ray relationship. The View3D.IsPerspective property indicates whether a 3D view is perspective or orthographic.

18.2.1 Perspective View

The following picture illustrates how a perspective view is created.

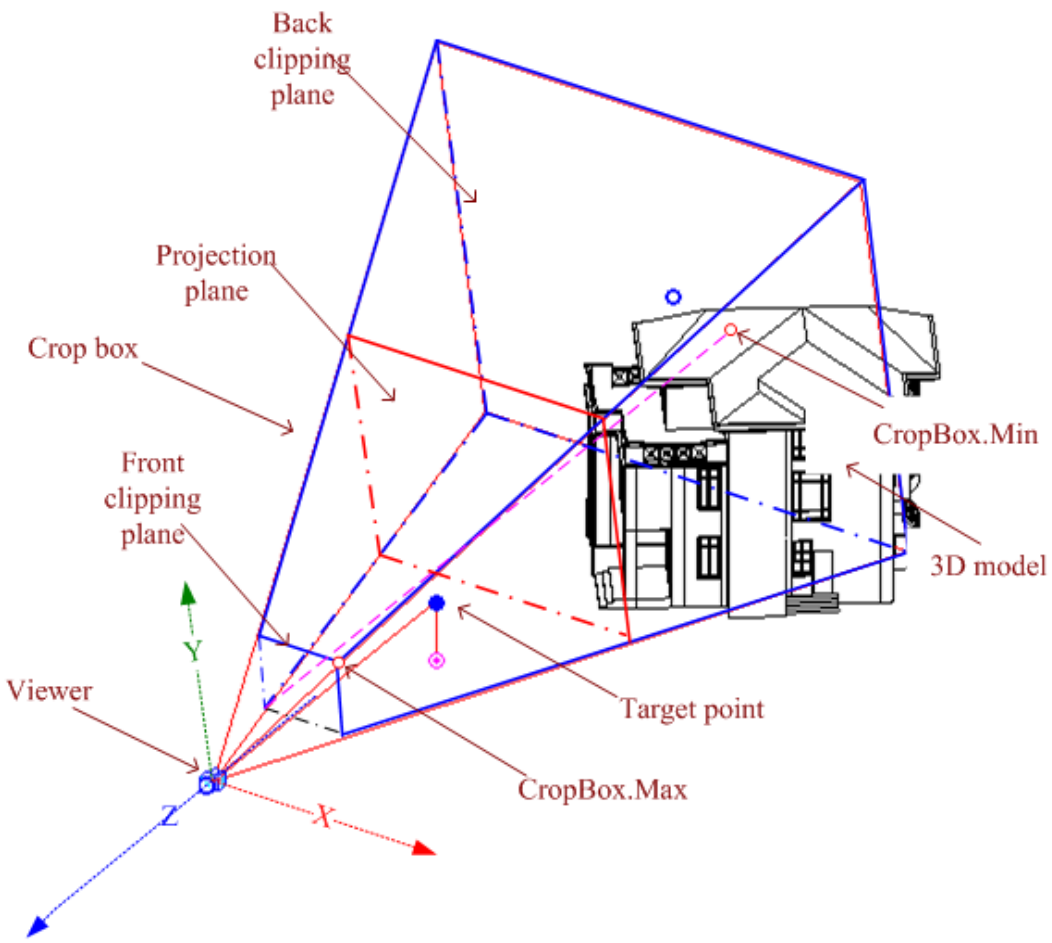


Figure 95: Perspective projection

- The straight projection rays pass through each point in the model and intersect the projection plane to form the projection contents.
- To facilitate the transformation from the world coordinate onto the view plane, the viewing coordinate system is based on the viewer.
- Its origin, represented by the View.Origin property, is the viewer position.
- The viewer's world coordinates are retrieved using the View3D.EyePosition property. Therefore, in 3D views, View.Origin is always equal to View3D.EyePosition.
- As described, the viewing coordinate system is determined as follows:
  - The X-axis is determined by View.RightDirection.
  - The Y-axis is determined by View.UpDirection.
  - The Z-axis is determined by View.ViewDirection.
- The view direction is from the target point to the viewer in the 3D space, and from the screen to the viewer in the screen space.

The perspective view crop box is part of a pyramid with the apex at the viewer position. It is the geometry between the two parallel clip planes. The crop box bounds the portion of the model that is clipped out and projected onto the view plane.

- The crop box is represented by the View.CropBox property, which returns a BoundingBoxXYZ object.

- The CropBox.Min and CropBox.Max points are marked in the previous picture. Note that the CropBox.Min point in a perspective view is generated by projecting the crop box front clip plane onto the back clip plane.

Crop box coordinates are based on the viewing coordinate system. Use Transform.OfPoint() to transform CropBox.Min and CropBox.Max to the world coordinate system. For more detail about Transform, refer to the Geometry.Transform section in the Geometry chapter.

The project plane plus the front and back clip plane are all plumb to the view direction. The line between CropBox.Max and CropBox.Min is parallel to the view direction. With these factors, the crop box geometry can be calculated.

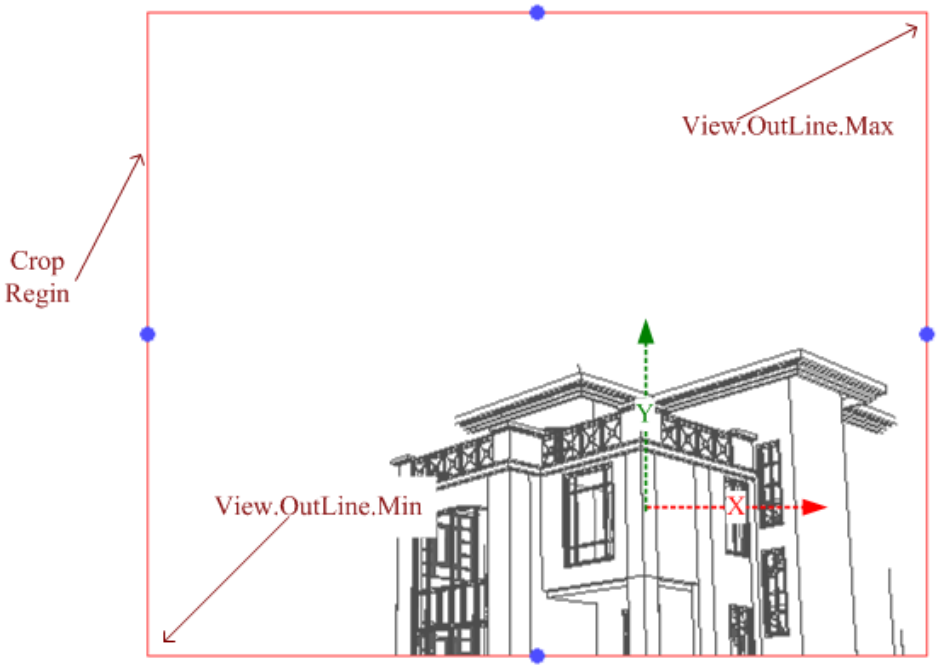


Figure 96: Perspective 3D view

The previous picture shows the projection plane on screen after cropping. The crop region is the rectangle intersection of the projection plane and crop box.

- Geometry information is retrieved using the View.CropRegion property. This property returns a BoundingBoxUV instance.
- The View.OutLine.Max property points to the upper right corner.
- The View.OutLine.Min property points to the lower left corner.
- Like the crop box, the crop region coordinates are based on the viewing coordinate system. The following expressions are equal.

View.CropBox.Max.X(Y) / View.OutLine.Max.X(Y)

== View.CropBox.Min.X(Y) / View.OutLine.Min.X(Y)

Since the size of an object's perspective projection varies inversely with the distance from that object to the center of the projection, scale is meaningless for perspective views. The perspective 3D view Scale property always returns zero. Currently, the API cannot create Perspective views.

18.2.2 Orthographic View

Autodesk.Revit.Creation.Document provides the NewView3D method to create an orthographic 3D view.

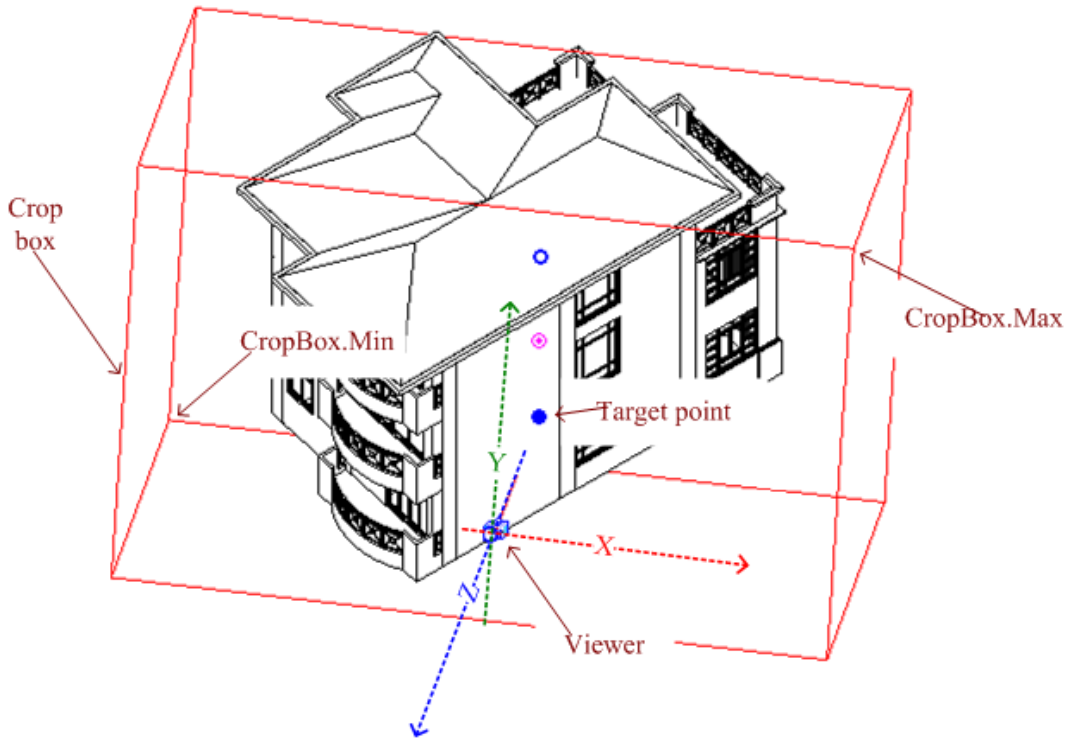


Figure 97: Parallel projection

Orthographic views are generated using parallel projection rays by projecting the model onto a plane that is normal to the rays. The viewing coordinate system is similar to the perspective view, but the crop box is a parallelepiped with faces that are parallel or normal to the projection rays. The View.CropBox property points to two diagonal corners whose coordinates are based on the viewing coordinate system.

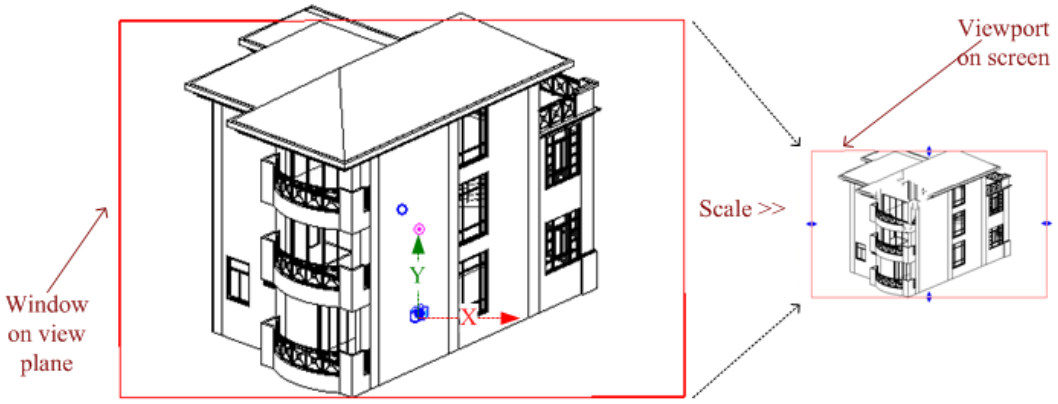


Figure 98: Scale the window on view plane to screen viewport

The model is projected onto a view plane and then scaled onto the screen. The View.Scale property represents the ratio of actual model size to the view size. The related expressions are as follows:

View.CropBox.Max.X(Y) / View.OutLine.Max.X(Y)

== View.CropBox.Min.X(Y) / View.OutLine.Min.X(Y)

== View.Scale

Code Region 18-4: NewView3D()

```
public View3D NewView3D(XYZ viewDirection);
```

Create an orthographic 3D view by passing the view vector to the Autodesk.Revit.Creation.Document.NewView3D method. This vector can be a unit vector or not. Revit determines the following:

- Position of the viewer.
- How to create the viewing coordinate system using the view direction.
- How to create the crop box to crop the model.

Once the view is created, you can resize the crop box to view different portions of the model. The API does not support modifying the viewing coordinate system.

The following code sample illustrates how to create a 3D view. The created view is orthographic.

Code Region 18-5: Creating a 3D view

```
// Create a new View3D
XYZ direction = new XYZ(1, 1, 1);
View3D view3D = document.Create.NewView3D(direction);
if (null == view3D)
{
    throw new Exception("Failed to create new View3D");
}
```

18.2.3 3D Views SectionBox

Each view has a crop box. The crop box focuses on a portion of the model to project and show in the view. For 3D views, there is another box named section box.

- The section box determines which model portion appears in a 3D view.
- The section box is used to clip the 3D model's visible portion.
- The part outside the box is invisible even if it is in the crop box.
- The section box is different from the crop box in that it can be rotated and moved with the model.

The section box is particularly useful for large models. For example, if you want to render a large building, use a section box. The section box limits the model portion used for calculation. To display the section box, in the 3D view Element Properties dialog box, select Section Box in the Extents section. You can also set it using the API:

Code Region 18-6: Showing the Section Box

```
private void ShowHideSectionBox(Autodesk.Revit.DB.View3D view3D)
{
    foreach (Parameter p in view3D.Parameters)
```

```
    {
        // Get Section Box parameter
        if (p.Definition.Name.Equals("Section Box"))
        {
            // Show Section Box
            p.Set(1);
            // Hide Section Box
            // p.Set(0);
            break;
        }
    }
}
```

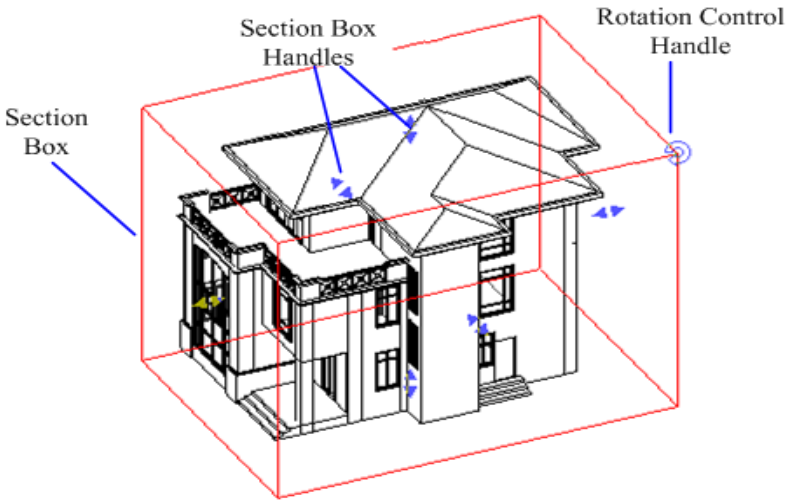


Figure 99: Section box

The View3D.SectionBox property is used to get and change the box extents. In some cases, setting the View3D.SectionBox can have a side effect. Setting the property to certain values can change the box capacity and display it in the view. However, you can assign a null value to the SectionBox to keep the modified value and make the section box invisible in the view. To avoid displaying the section box, change the section box value, then set the section box to null. The following code sample illustrates this process. Notice it only works when the Section Box check box is selected in the View property dialog box.

Code Region 18-7: Hiding the Section Box

```
private void ExpandSectionBox(View3D view)
{
   // The orignial section box
   BoundingBoxXYZ sectionBox = view.SectionBox;

   // Expand the section box
   XYZ deltaXYZ = sectionBox.Max - sectionBox.Min;
   sectionBox.Max += deltaXYZ / 2;
   sectionBox.Min -= deltaXYZ / 2;

   //After reseting the section box, it will be shown in the view.
```

```
   //It only works when the Section Box check box is
   //checked in View property dialog.
    view.SectionBox = sectionBox;

   //Setting the section box to null will make it hidden.
    view.SectionBox = null;       // line x
}
```

Note: If you set view.SectionBox to null, it has the same effect as hiding the section box using the Section Box parameter. The current section box is stored by view and is restored when you show the section box using the SectionBox parameter.

18.3 ViewPlan

Plan views are level-based. There are two types of plan views, floor plan view and ceiling plan view.

- Generally the floor plan view is the default view opened in a new project.
- Most projects include at least one floor plan view and one ceiling plan view.
- Plan views are usually created after adding new levels to the project.

Adding new levels using the API does not add plan views automatically. Autodesk.Revit.Creation.Document provides a NewViewPlan() method to create a plan view.

Code Region 18-8: NewViewPlan()

```
public ViewPlan NewViewPlan(string pViewName, Level pLevel, ViewType viewType);
```

The viewType parameter must be FloorPlan or CeilingPlan. The level parameter represents a level element in the project to which the plan view is associated.

The following code creates a floor plan and a ceiling plan based on a certain level.

Code Region 18-9: Creating a floor plan and ceiling plan

```
// Create a Level and a Floor Plan based on it
double elevation = 10.0;
Level level1 = document.Create.NewLevel(elevation);
ViewPlan floorView = document.Create.NewViewPlan(null, level1, ViewType.FloorPlan);

// Create another Level and a Ceiling Plan based on it
elevation += 10.0;
Level level2 = document.Create.NewLevel(elevation);
ViewPlan ceilingView = document.Create.NewViewPlan(null, level2, ViewType.CeilingPlan);
```

18.4 ViewDrafting

The drafting view is not associated with the model. It allows the user to create detail drawings that are not included in the model.

- In the drafting view, the user can create details in different view scales (coarse, fine, or medium).
- You can use 2D detailing tools, including:

- Detail lines
- Detail regions
- Detail components
- Insulation
- Reference planes
- Dimensions
- Symbols
- Text

These tools are the same tools used to create a detail view.

- Drafting views do not display model elements.

Use the Autodesk.Revit.Creation.NewViewDrafting() method to create a drafting view. Model elements are not displayed in the drafting view.

18.5 ViewSection

Section views cut through the model to expose the interior structure. The Auotdesk.Revit.Creation.NewViewSection() method creates the section view.

Code Region 18-10: NewViewSection()

```
public ViewSection NewViewSection(BoundingBoxXYZ box);
```

The box parameter is the section view crop box. It provides the orientation and bounds which are required for the section view. Usually, another view's crop box is used as the parameter. You can also build a custom BoundingBoxXYZ instance to represent the orientation and bounds.

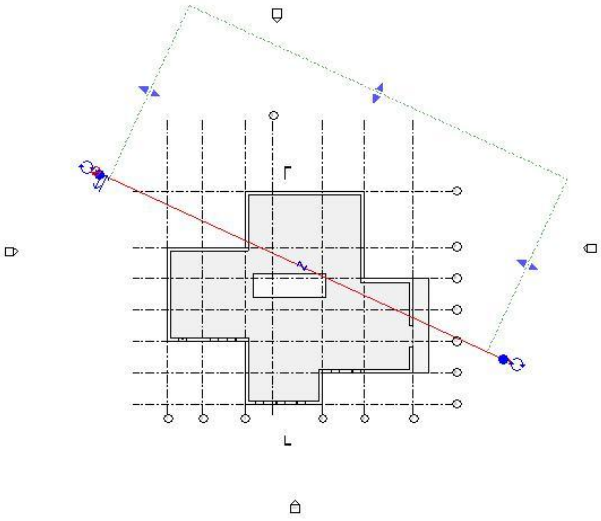


Figure 100: Plan view section

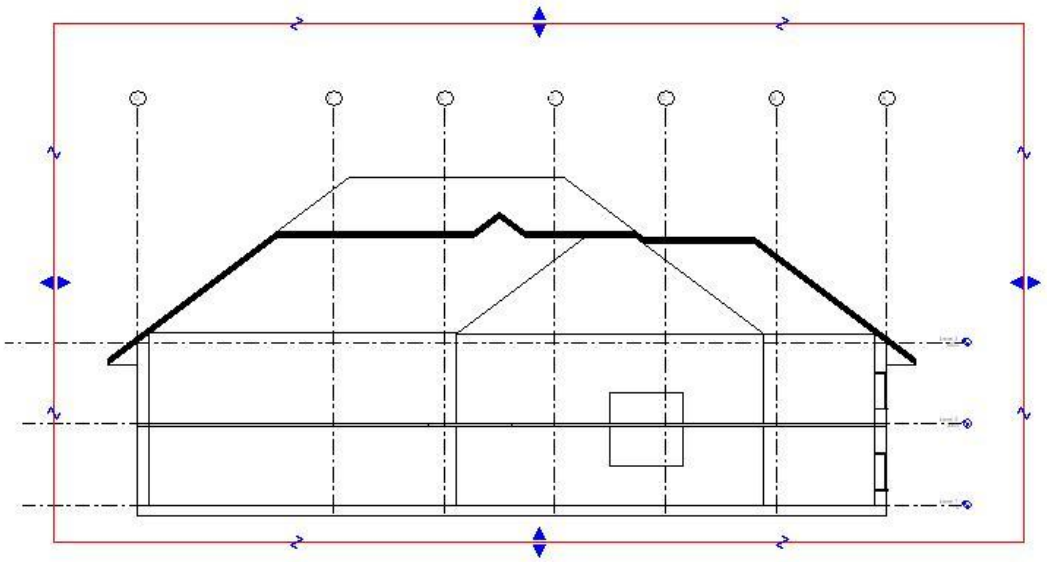


Figure 101: Section view section

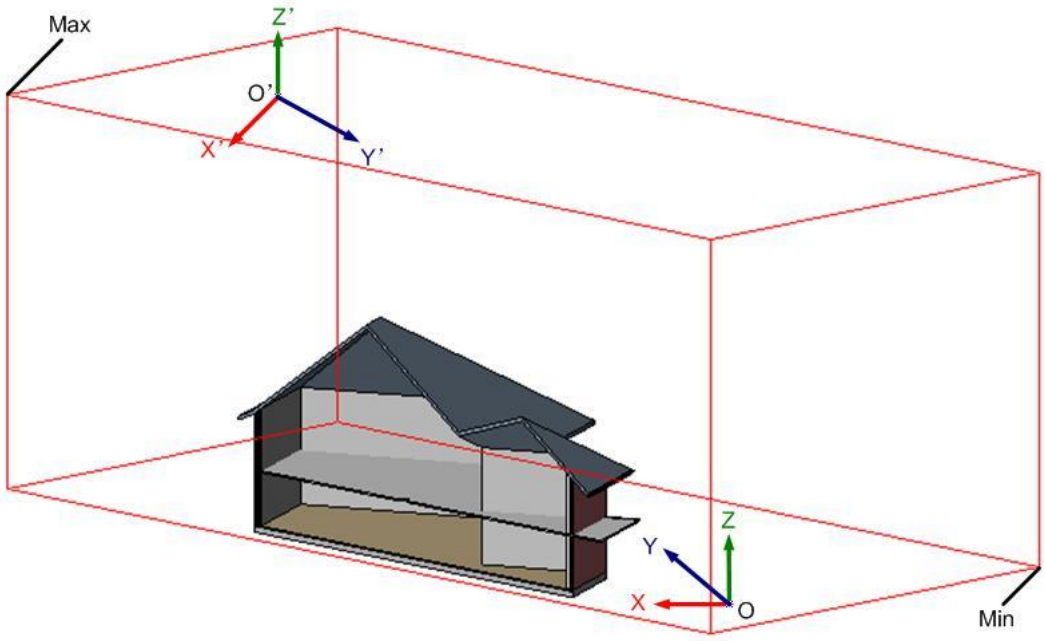


Figure 102: Section view box

For example, apply the box to the section view create method. The coordinate system O-XYZ is the model coordinate system and the box Max and Min points are based on this system. The created section view introduces a new system, O`-X`Y`Z`, as the viewing coordinate system.

- Point O` is the section view origin.
- Its coordinates are retrieved using the View.Origin property.
- The section view crop box is calculated using the box.

Note: When you create a section using the command View > New > Section, the created view is located in the Sections (Building Section) node in the Project Browser. The view class type is Elements.View, and the view type is ViewType.Section. However, when you create a section using the Autodesk.Revit.Creation.NewViewSection method(), the created view is in the

Detail Views (Detail) node. The class type is Elements.ViewSection and the view type is ViewType.Detail.

18.6 ViewSheet

A sheet contains views and a title block. When creating a sheet view with the Autodesk.Revit.Creation.NewViewSheet() method, a title block family symbol is a required parameter for the method. The Autodesk.Revit.Document TitleBlocks property contains all title blocks in the document. Choose one title block to create the sheet.

Code Region 18-11: AddView()

```
public void AddView(View newView, UV location);
```

The newly created sheet has no views. The ViewSheet.AddView() method is used to add views.

- The Geometry.UV location parameter identifies where the added views are located. It points to the added view's center coordinate (measured in inches).
- The coordinates, [0, 0], are relative to the sheet's lower left corner.

Each sheet has a unique sheet number in the complete drawing set. The number is displayed before the sheet name in the Project Browser. It is convenient to use the sheet number in a view title to cross-reference the sheets in your drawing set. You can retrieve or modify the number using the SheetNumber property. The number must be unique; otherwise an exception is thrown when you set the number to a duplicate value.

The following example illustrates how to create and print a sheet view. Begin by finding an available title block in the document and use it to create the sheet view. Next, add the document active view. The active view is placed in the center of the sheet. Finally, print the sheet by calling the View.Print() method.

Code Region 18-12: Creating a sheet view

```
private void CreateSheetView(Autodesk.Revit.DB.Document document, View3D view3D)
{

    // Get an available title block from document
    FamilySymbolSet fsSet = document.TitleBlocks;
    if (fsSet.Size == 0)
    {
        throw new Exception("No title blocks");
    }

    FamilySymbol fs = null;
    foreach (FamilySymbol f in fsSet)
    {
        if (null != f)
        {
            fs = f;
            break;
        }
    }

    // Create a sheet view
```

```
    ViewSheet viewSheet = document.Create.NewViewSheet(fs);
    if (null == viewSheet)
    {
        throw new Exception("Failed to create new ViewSheet.");
    }

    // Add passed in view onto the center of the sheet
    UV location = new UV((viewSheet.Outline.Max.U - viewSheet.Outline.Min.U) / 2,
                         (viewSheet.Outline.Max.V - viewSheet.Outline.Min.V) / 2);

    viewSheet.AddView(view3D, location);

    // Print the sheet out
    if (viewSheet.CanBePrinted)
    {
        TaskDialog taskDialog = new TaskDialog("Revit");
        taskDialog.MainContent = "Print the sheet?";
        TaskDialogCommonButtons buttons = TaskDialogCommonButtons.Yes |
                                          TaskDialogCommonButtons.No;
        taskDialog.CommonButtons = buttons;
        TaskDialogResult result = taskDialog.Show();

        if (result == TaskDialogResult.Yes)
        {
            viewSheet.Print();
        }
    }
}
```

Note: You cannot add a sheet view to another sheet and you cannot add a view to more than one sheet; otherwise an argument exception occurs.

18.6.1 Printer Setup

You may want to change the settings of the printer before printing a sheet. The API exposes the settings for the printer with the PrintManager class, and related Autodesk.Revit.DB classes:

Class	Functionality
Autodesk.Revit.DB.PrintManager	Represents the Print information in Print Dialog (File->Print) within the Revit UI.
Autodesk.Revit.DB.PrintParameters	An object that contains settings used for printing the document.
Autodesk.Revit.DB.PrintSetup	Represents the Print Setup (File->Print Setup...) within the Revit UI.
Autodesk.Revit.DB.PaperSize	An object that represents a Paper Size of Print Setup within the Autodesk Revit project.
Autodesk.Revit.DB.PaperSizeSet	A set that can contain any number of paper size

	objects.
Autodesk.Revit.DB.PaperSource	An object that represents a Paper Source of Print Setup within the Autodesk Revit project.
Autodesk.Revit.DB.PaperSourceSet	A set that can contain any number of paper source objects.
Autodesk.Revit.DB.ViewSheetSetting	Represents the View/Sheet Set (File->Print) within the Revit UI.
Autodesk.Revit.DB.PrintSetting	Represents the Print Setup (File->Print Setup...) within the Revit UI.

For an example of code that uses these objects, see the ViewPrinter sample application that is included with the Revit Platform SDK.

19 Material

In the Revit Platform API, material data is stored and managed as an Element. The Material element includes Elements.Material and its subclasses:

- MaterialGeneric
- MaterialConcrete
- MaterialWood
- MaterialSteel
- MaterialOther.

Material features are represented by properties, such as FillPattern, Color, Render and so on.

In this chapter, you learn how to access material elements and how to manage the Material objects in the document. The section, Walkthrough: Get Window Materials, provides a walkthrough showing how to get a window material.

19.1 General Material Information

Before you begin the walkthrough, read through the following section for a better understanding of the Material class.

19.1.1 Classification

All Elements.Material objects are available in the Settings class Materials property. For more details, refer to the Management in Document section in this chapter. Material objects are also available in Document, Category, Element, Face, and so on, and are discussed in the pertinent sections in this chapter. Wherever you get a material object, it is represented as the Elements.Material class requiring you to downcast the object to its derived type.

There are two ways to downcast the Elements.Material object to its derived type. Use Runtime Type Information (RTTI) or BuiltInParameter.

19.1.1.1 Use Runtime Type Information (RTTI)

The basic way to downcast the object is to use RTTI, as illustrated below.

Code Region 19-1: Downcasting using RTTI

```
FilteredElementCollector collector = new FilteredElementCollector(document);

// OfClass (and ElementClassFilter) does not support subclasses of Material
collector.OfClass(typeof(Material));

FilteredElementIterator materialItr = collector.GetElementIterator();
materialItr.Reset();
while (materialItr.MoveNext())
{
    Material material = materialItr.Current as Material;
    if (material is MaterialSteel)
    {
        // downcast to MaterialSteel
        MaterialSteel steelMa = material as MaterialSteel;
    }
```

```
}
```

19.1.1.2 **Use BuiltInParameter**

The second option is to use the BuiltInParameter PHY_MATERIAL_PARAM_TYPE. It is an Elements.Material class integer parameter. The parameter value identifies the Material type, as shown in the following table.

Value	Type in API	Type in Revit
0	MaterialOther	Unassigned
1	MaterialConcrete	Concrete
2	MaterialSteel	Steel
3	MaterialGeneric	Generic
4	MaterialWood	Wood

Table 46: PHY_MATERIAL_PARAM_TYPE Parameter Values

Note: Do not convert the integer parameter to Autodesk.Revit.DB.Structure.StructuralMaterialType. StructuralMaterialType represents a structural FamilyInstance family parameter in the Revit Structure product. The FamilyInstance or FamilySymbol Material property is determined by the Family. It is not related to the Material object. In Revit, when editing a Structural Family (Settings > Family Category and Parameter), note that the Family Parameter, Structural Material Type, corresponds to the StructuralMaterialType enumerated type.

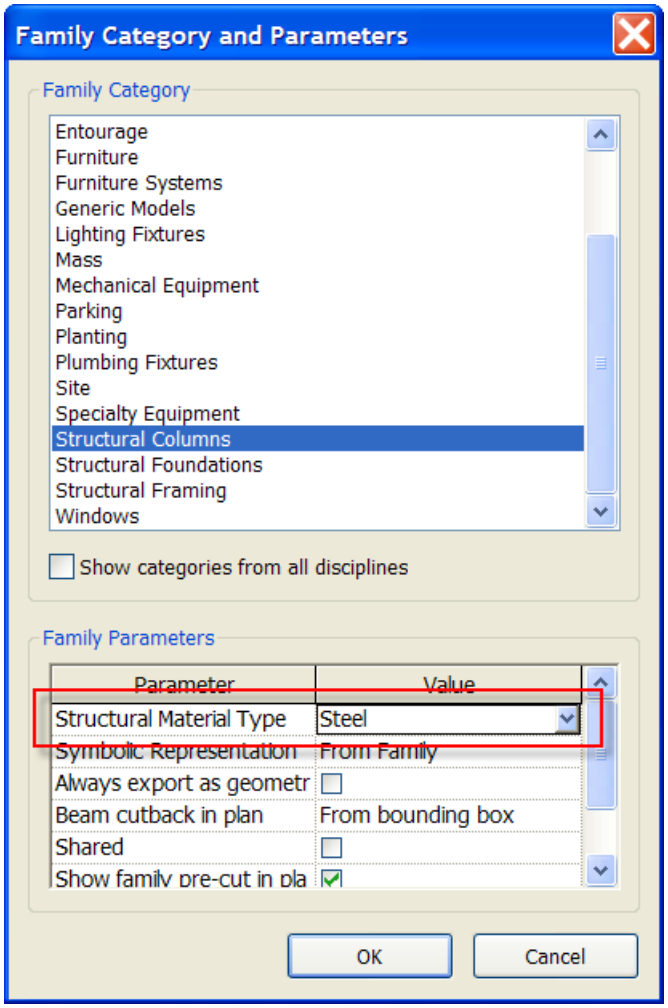


Figure 103: Family parameter and Enum material

Although there is no enumerated type for the Integer you can define one, as illustrated in the following code:

Code Region 19-2: Determining material type

```
private void GetMaterialType(Material material)
{
    Parameter parameter = material.get_Parameter(BuiltInParameter.PHY_MATERIAL_PARAM_TYPE);
    int materialType = parameter.AsInteger();
    switch (materialType)
    {
        case 0:    // Other
             MaterialOther matOther = material as MaterialOther;
             if (null != matOther)
             {
                 // use MaterialOther object
             }
             break;
        case 1:  // Concrete
            MaterialConcrete matConcrete = material as MaterialConcrete;
            if (null != matConcrete)
```

```
            {
                // use MaterialConcrete object
            }
            break;
        case 2:   // Steel
            MaterialSteel matSteel = material as MaterialSteel;
            if (null != matSteel)
            {
                // use MaterialSteel object
            }
            break;
        case 3:    // Generic
            MaterialGeneric matGeneric = material as MaterialGeneric;
            if (null != matGeneric)
            {
                // use MaterialGeneric object
            }
            break;
        case 4:    // Wood
            MaterialWood matWood = material as MaterialWood;
            if (null != matWood)
            {
                // use MaterialWood object
            }
            break;
    }
}
```

There is no way to change the basic Material type. For example, you cannot change a MaterialSteel object to a MaterialWood object.

Note: The API does not provide access to the values of Concrete Type for Concrete material.

19.1.2 Properties

The material object properties identify a specific type of material including color, fill pattern, and more.

19.1.2.1 Properties and Parameter

Elements.Material and its subclasses provide properties that have a counterpart Parameter. For example, the Color property corresponds to the BuiltInParameter MATERIAL_PARAM_COLOR.

In some cases, it is better to use a parameter rather than a property. For example, the AsValueString() method (refer to the Parameter chapter sections AsValueString() and SetValueString()) is used to get a Parameter value in the unit. The result is converted to the same value as the one you see in Revit.

Code Region 19-3: Getting a material parameter

```
public void GetYoungModulus(MaterialSteel materialSteel)
{
    //get young mod x as a parameter
```

```
    Parameter parameter =
            materialSteel.get_Parameter(BuiltInParameter.PHY_MATERIAL_PARAM_YOUNG_MOD1);
    double dYOUNG_MODX = parameter.AsDouble();
    string strYOUNG_MOD = parameter.AsValueString();
    string message = "Young's Modulus X from parameter: " + dYOUNG_MODX;

    // get young mod x as property
    double dYOUNG_MODX2 = materialSteel.YoungModulusX;
    message += "\nYoung's Modulus X from property: " + dYOUNG_MODX2;

    message += "\nYoung's Modulus X AsValueString; " + strYOUNG_MOD;

    TaskDialog.Show("Revit", message);
}
```

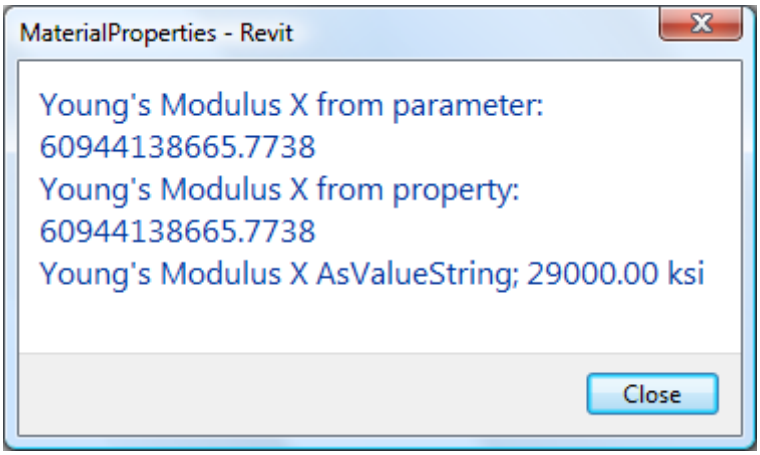


Figure 104: Result from getting a material parameter code

19.1.2.2 **Rendering Information**

Collections of rendering data are organized into objects called Assets, which are read-only. You can obtain all available Appearance-related assets from the Application.Assets property. An appearance asset can be accessed from a material via the Material.RenderAppearance property.

The following figure shows the Render Appearance Library dialog box, which shows how some rendering assets are displayed in the UI.

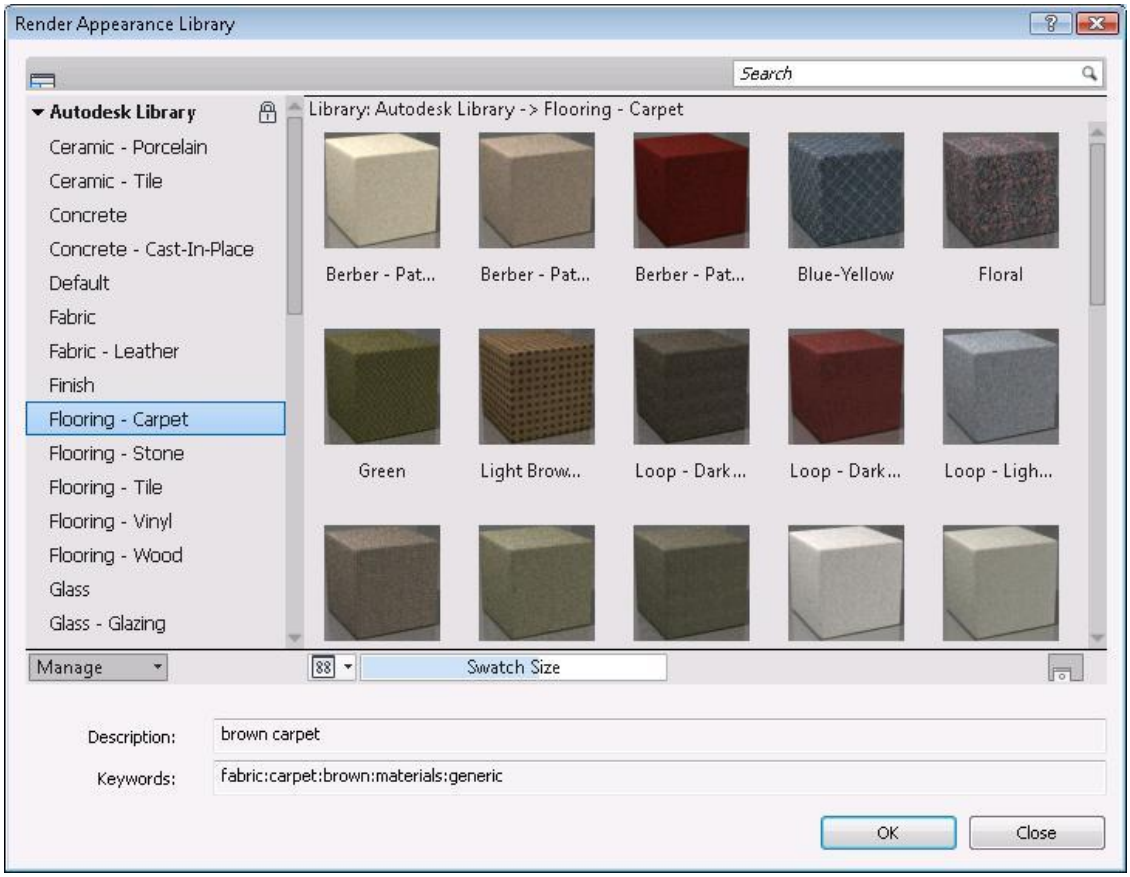


Figure 105: Render Appearance Library

The Materials sample application included with the SDK shows how to set the RenderApperance property to a material selected in a dialog. The dialog is populated with all the Asset objects in Application.Assets.

19.1.2.3**FillPattern**

All FillPatterns in a document are available in the Settings class FillPatterns property. Currently the FillPattern object only contains the Name and ID information. There are two kinds of FillPatterns: Drafting and Model. In the UI, you can only set Drafting fill patterns to Material.CutPattern. However, the classification is not exposed in the API. The following example shows how to change the material FillPattern.

Code Region 19-4: Setting the fill pattern

```
public void SetFillPattern(Document document, Material material)
{
    foreach (FillPattern fillPattern in document.Settings.FillPatterns)
    {
        // always set successfuly
        material.CutPattern = fillPattern;
        material.SurfacePattern = fillPattern;
    }
}
```

19.2 Material Management

The MaterialSet object retrieved from the Settings class manages all Materials in the Document. Every Elements.Material object in the Document is identified by a unique name.

The following example illustrates how to use the material name to get material and how to get the Document material set.

Code Region 19-5: Getting a material by name

```
Materials materials = document.Settings.Materials;
Material floorMaterial = null;
string floorMaterialName = "Default Floor";
floorMaterial = materials.get_Item(floorMaterialName);
if (null != floorMaterial)
{
    TaskDialog.Show("Revit", "Material found.");
}

//or travel all the materials
foreach (Material material in materials)
{
    if (floorMaterialName == material.Name)
    {
        floorMaterial = material;
        break;
    }
}
if (null != floorMaterial)
{
    TaskDialog.Show("Revit", "Material found.");
}
```

Note: To run the sample code, make sure the material name exists in your document. All material names for the current document are located under the Manage tab (Project Settings panel > Materials).

19.2.1 Creating Materials

There are two ways to create a new Elements.Material object in the API.

- Duplicate an existing Material
- Add a new Material with a specific type.

When using the Duplicate() method, the returned Material object has the same type as the original and it is added to the Materials collection automatically.

Code Region 19-6: Duplicating a material

```
private bool DuplicateMaterial(Material material)
{
    bool duplicated = false;
    //try to duplicate a new instance of Material class using duplicate method
    //make sure the name of new material is unique in MaterailSet
```

```
    string newName = "new" + material.Name;
    Material myMaterial = material.Duplicate(newName);
    if (null == myMaterial)
    {
        TaskDialog.Show("Revit", "Failed to duplicate a material!");
    }
    else
    {
        duplicated = true;
    }

    return duplicated;
}
```

Use the Materials class to add a new Material directly. No matter how it is applied, it is necessary to specify a unique name. The unique name is the Elements.Material object key.

Code Region 19-7: Adding a new Material

```
private MaterialConcrete AddConcrete(Document document)
{
    string newName = "myConcreteMaterial";
    MaterialConcrete myConcrete = document.Settings.Materials.AddConcrete(newName);

    if (document.Settings.Materials.Contains(newName) == true)
    {
        TaskDialog.Show("Revit", "Concrete material added successfully!");
    }

    return myConcrete;
}
```

19.2.2 Deleting Materials

There are two ways to delete a material.

- Materials.Remove()
- Document.Delete()

The Materials.Remove() method is specific to the Elements.Material class.

Code Region 19-8: Removing a material

```
private bool CreateOtherMaterial(Autodesk.Revit.DB.Document document)
{
    string oldName = "myOtherMaterial";
    MaterialOther myOther = document.Settings.Materials.AddOther(oldName);
    if (null == myOther)
    {
        return false;
    }
```

```
    //try to duplicate a new instance using duplicate method
    //make sure the name of new material is unique in MaterailSet
    string newName = "new" + myOther.Name;
    MaterialOther newOther = myOther.Duplicate(newName) as MaterialOther;
    if (null == newOther)
    {
        TaskDialog.Show("Revit", "Failed to duplicate an other material!");
        return false;
    }
    else
    {
        TaskDialog.Show("Revit", "Duplicated other material.");
    }

    // Remove the original created material
    document.Settings.Materials.Remove(oldName);

    if (document.Settings.Materials.Contains(oldName) == false)
    {
        TaskDialog.Show("Revit", "Material (myOtherMaterial) removed successfully!");
    }

    return true;
}
```

Document.Delete() is a more generic method. See the Editing Elements chapter for details.

Note: Though you can delete material using Document.Delete(), the Materials collection is not updated immediately after it is called. As a result, it is easy to cause Material management inconsistencies in your Add-in application. The best approach is to only use Materials.Remove().

19.3 Element Material

One element can have several elements and components. For example, FamilyInstance has SubComponents and Wall has CompoundStructure which contain several CompoundStructureLayers. (For more details about SubComponents refer to the Family Instances chapter and refer to the Walls, Floors, Roofs and Openings chapter for more information about CompoundStructure.)

In the Revit Platform API, get an element's materials using the following guidelines:

- If the element contains elements, get the materials separately.
- If the element contains components, get the material for each component from the parameters or in specific way (see Material section in the Walls, Floors, Roofs and Openings chapter).
- If the component's material returns null, get the material from the corresponding Element.Category sub Category.

19.3.1 Material in a Parameter

If the Element object has a Parameter where ParameterType is ParameterType.Material, you can get the element material from the Parameter. For example, a structural column FamilySymbol (a

FamilyInstance whose Category is BuiltInCategory.OST_StructuralColumns) has the Column Material parameter. Get the Material using the ElementId. The following code example illustrates how to get the structural column Material that has one component.

Code Region 19-9: Getting an element material from a parameter

```
public void GetMaterial(Document document, FamilyInstance familyInstance)
{
    foreach (Parameter parameter in familyInstance.Parameters)
    {
        Definition definition = parameter.Definition;
        // material is stored as element id
        if (parameter.StorageType == StorageType.ElementId)
        {
            if (definition.ParameterGroup == BuiltInParameterGroup.PG_MATERIALS &&
                     definition.ParameterType == ParameterType.Material)
            {
                Autodesk.Revit.Elements.Material material = null;
                ElementId materialId = parameter.AsElementId();
                if (-1 == materialId.Value)
                {
                    //Invalid ElementId, assume the material is "By Category"
                    if (null != familyInstance.Category)
                    {
                        material = familyInstance.Category.Material;
                    }
                }
                else
                {
                    material = document.Settings.Materials.get_Item(materialId);
                }

                TaskDialog.Show("Revit","Element material: " + material.Name);

                break;
            }
        }
    }
}
```

Note: If the material property is set to By Category in the UI, the ElementId for the material is -1 and cannot be used to retrieve the Material object as shown in the sample code. Try retrieving the Material from Category as described in the next section.

Some material properties contained in other compound parameters are not accessible from the API. As an example, in the following figure, for System Family: Railing, the Rail Structure parameter's StorageType is StorageType.None. As a result, you cannot get material information in this situation.

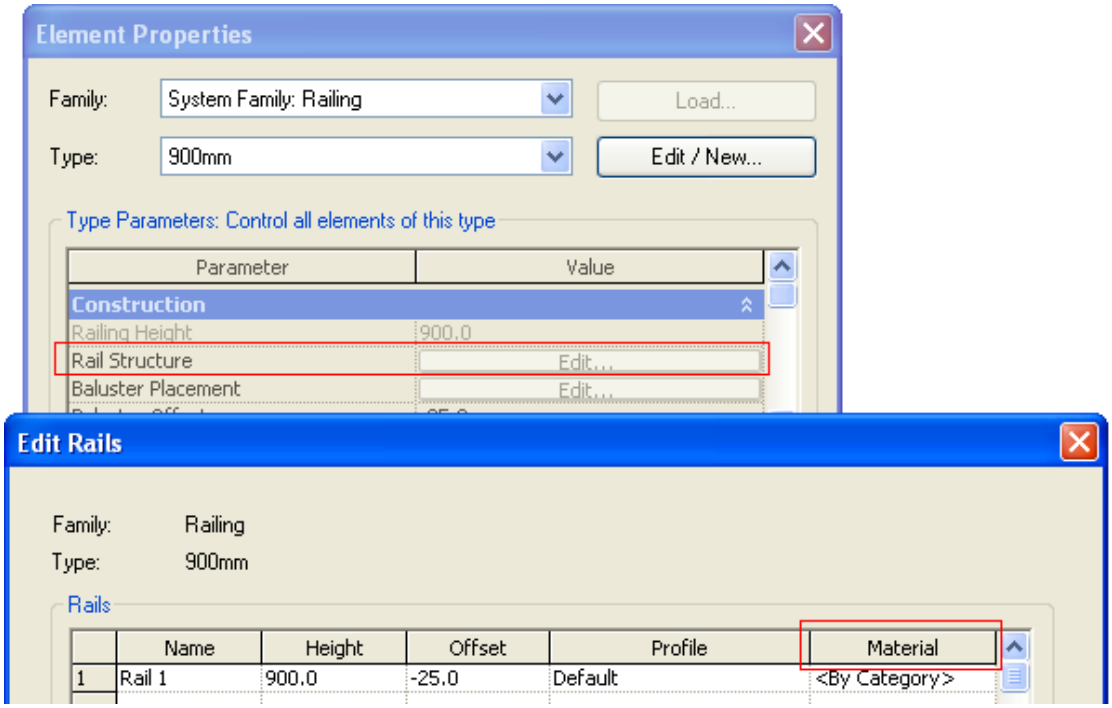


Figure 106: Rail structure property

19.3.2**Material and Category**

Only model elements can have material.

From the Revit Manage tab, click Settings > Object Styles to display the Object Styles dialog box. Elements whose category is listed in the Model Objects tab have material information.

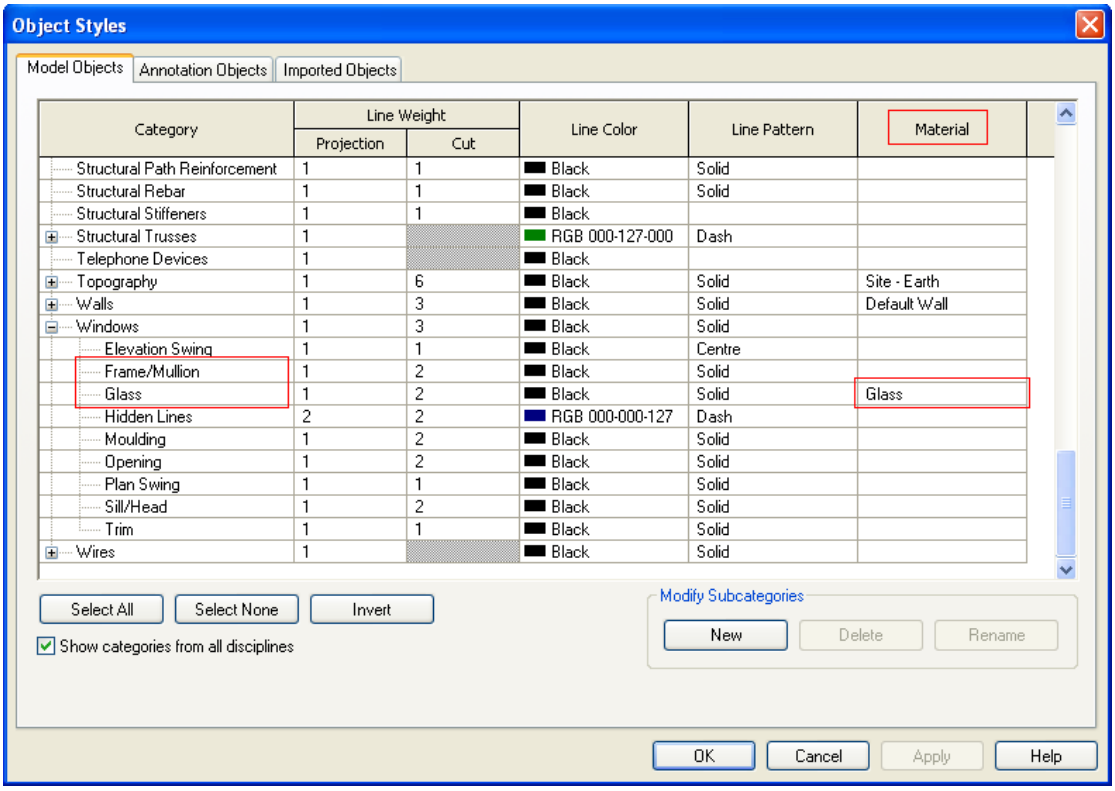


Figure 107: Category material

Only Model elements can have the Material property assigned. Querying Material for a category that corresponds to other than Model elements (e.g. Annotations or Imported) will therefore always result in a null. For more details about the Element and Category classifications, refer to the Elements Essentials chapter.

If an element has more than one component, some of the Category.Subcategories correspond to the components.

In the previous Object Styles dialog box, the Windows Category and the Frame/Mullion and Glass subcategories are mapped to components in the windows element. In the following picture, it seems the window symbol Glass Pane Material parameter is the only way to get the window pane material. However, the value is By Category and the corresponding Parameter returns -1 as an invalid ElementId.

In this case, the pane's Material is not null and it depends on the Category OST_WindowsFrameMullionProjection's Material property which is a subcategory of the window's category, OST_Windows. If it returns null as well, the pane's Material is determined by the parent category OST_Windows. For more details, refer to the Walkthrough: Get Materials of a Window section in this chapter.

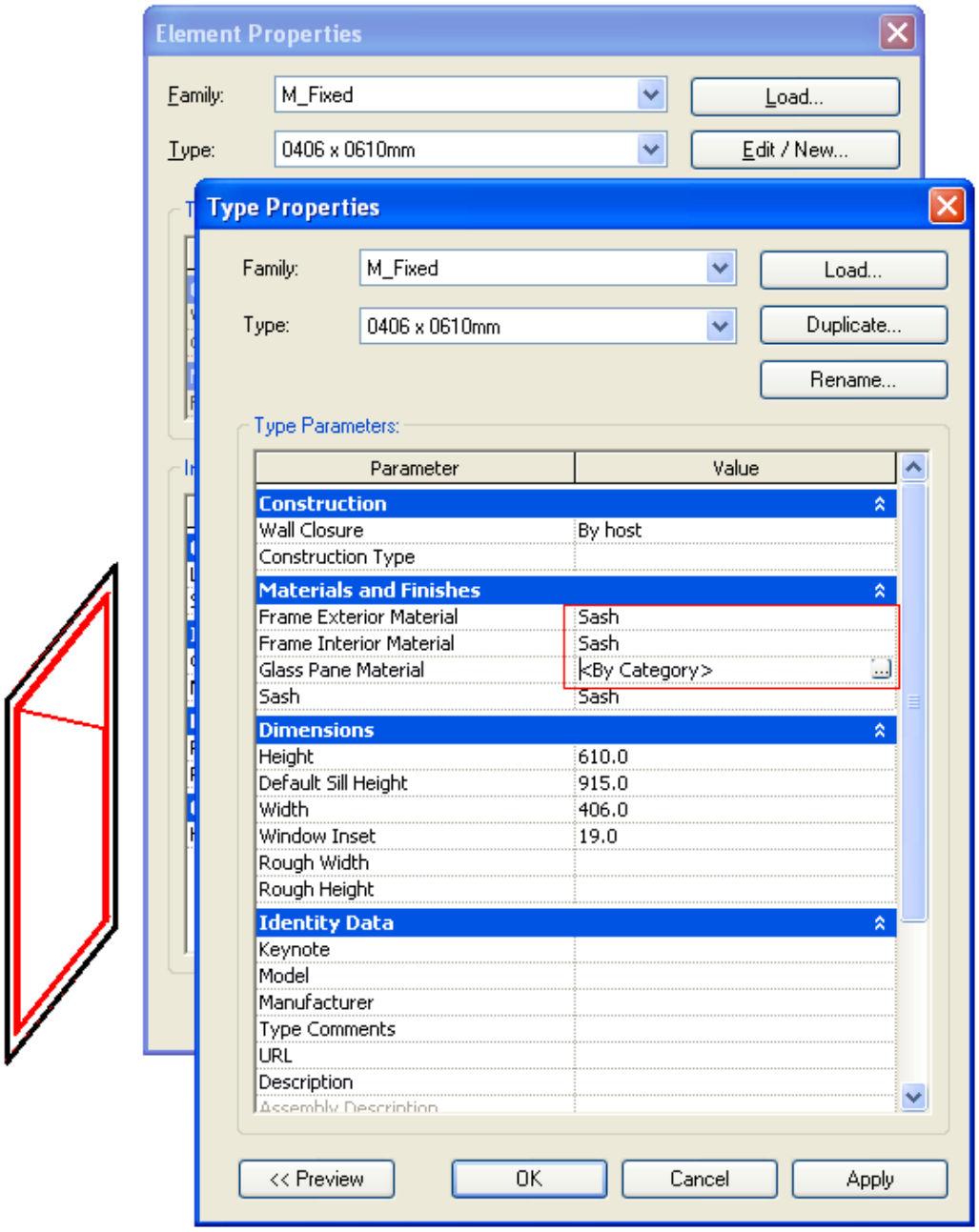


Figure 108: Window material

19.3.3**CompoundStructureLayer Material**

You can get the CompoundStructureLayer object from HostObjAttributes. For more details, refer to the Walls, Floors, Roofs and Openings chapter.

19.3.4**Retrieve Element Materials**

The following diagram shows the workflow to retrieve Element Materials:

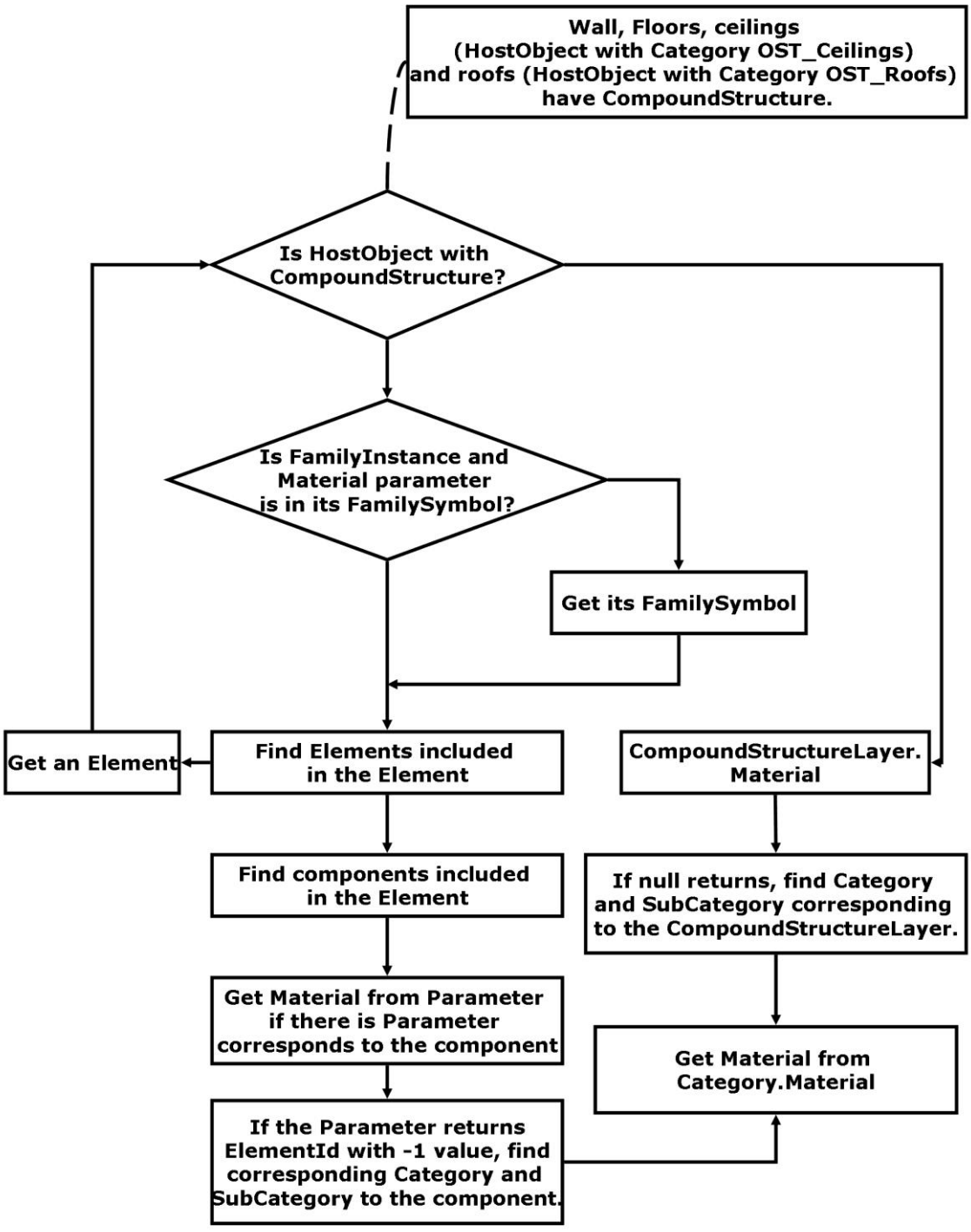


Figure 109: Getting Element Material workflow

This workflow illustrates the following process:

- The workflow shows how to get the Elements.Material object (not Autodesk.Revit.DB.Structure.StructuralMaterialType enumerated type) that belongs to the element.
- There are two element classifications when retrieving the Material:
  - HostObject with CompoundStructure – Get the Elements.Material object from the CompoundStructureLayer class Material property.

    - Others - Get the Material from the Parameters.
- When you get a null Material object or an invalid ElementId with a value of -1, try the Material from the corresponding category. Note that a FamilyInstance and its FamilySymbol usually have the same category.
- The more you know about the Element object, the easier it is to get the material. For example:
    - If you know the Element is a beam, you can get the instance parameter Beam Material
    - If you know the element is a window, you can cast it to a FamilyInstance and get the FamilySymbol.
- After that you can get the Parameters such as Frame Exterior Material or Frame Interior Material to get the Elements.Material object. If you get null try to get the Material object from the FamilySymbol Category.
- Not all Element Materials are available in the API.

19.3.5 Walkthrough: Get Window Materials

The following code illustrates how to get the Window Materials.

Code Region 19-10: Getting window materials walkthrough

```
public void GetMaterial(Document document, FamilyInstance window)
{
    Materials materials = document.Settings.Materials;
    FamilySymbol windowSymbol = window.Symbol;
    Category category = windowSymbol.Category;
    Autodesk.Revit.DB.Material frameExteriorMaterial = null;
    Autodesk.Revit.DB.Material frameInteriorMaterial = null;
    Autodesk.Revit.DB.Material sashMaterial = null;
    // Check the paramters first
    foreach (Parameter parameter in windowSymbol.Parameters)
    {
        switch (parameter.Definition.Name)
        {
            case "Frame Exterior Material":
                frameExteriorMaterial = materials.get_Item(parameter.AsElementId());
                break;
            case "Frame Interior Material":
                frameInteriorMaterial = materials.get_Item(parameter.AsElementId());
                break;
            case "Sash":
                sashMaterial = materials.get_Item(parameter.AsElementId());
                break;
            default:
                break;
        }
    }
    // Try category if the material is set by category
    if (null == frameExteriorMaterial)
        frameExteriorMaterial = category.Material;
```

```
    if (null == frameInteriorMaterial)
        frameInteriorMaterial = category.Material;
    if (null == sashMaterial)
        sashMaterial = category.Material;
    // Show the result because the category may have a null Material,
    // the Material objects need to be checked.
    string materialsInfo = "Frame Exterior Material: " + (null != frameExteriorMaterial ?
frameExteriorMaterial.Name : "null") + "\n";
    materialsInfo += "Frame Interior Material: " + (null != frameInteriorMaterial ?
frameInteriorMaterial.Name : "null") + "\n";
    materialsInfo += "Sash: " + (null != sashMaterial ? sashMaterial.Name : "null") + "\n";
    TaskDialog.Show("Revit",materialsInfo);
}
```

20 Geometry

The Geometry namespace contains graphic-related types used to describe the graphical representation in the API. The API provides three classes used to describe and store the geometry information data according to their base classes:

- Geometry Node class – Includes classes derived from the GeometryObject class.
- Geometry Helper class – Includes classes derived from the APIObject class and value types
- Collection class – Includes classes derived from the IEnumerable or IEnumerator interface.

In this chapter, you learn how to use various graphic-related types, how to retrieve geometry data from an element, how to transform an element, and more.

20.1 Example: Retrieve Geometry Data from a Wall

This walkthrough illustrates how to get geometry data from a wall. The following information is covered:

- Getting the wall geometry edges.
- Getting the wall geometry faces.

Note: Retrieving the geometry data from Element in this example is limited because Instance is not considered. For example, sweeps included in the wall are not available in the sample code. The goal for this walkthrough is to give you a basic idea of how to retrieve geometry data but not cover all conditions. For more information about retrieving geometry data from Element, refer to Example: Retrieving Geometry Data from a Beam.

20.1.1 Create Geometry Options

In order to get the wall's geometry information, you must create a Geometry.Options object which provides detailed customized options. The code is as follows:

Code Region 20-1: Creating Geometry.Options

```
Autodesk.Revit.DB.Options geomOption = application.Create.NewGeometryOptions();
if (null != geomOption)
{
    geomOption.ComputeReferences = true;
    geomOption.DetailLevel = Autodesk Autodesk.Revit.DB.DetailLevels.Fine;

    // Either the DetailLevel or the View can be set, but not both
    //geomOption.View = commandData.Application.ActiveUIDocument.Document.ActiveView;

    TaskDialog.Show("Revit", "Geometry Option created successfully.");
}
```

Note: For more information, refer to the Geometry.Options section in this chapter.

20.1.2 Retrieve Faces and Edges

Wall geometry is a solid made up of faces and edges. Complete the following steps to get the faces and edges:

1. Retrieve a Geometry.Element instance using the Wall class Geometry property. This instance contains all geometry objects in the Object property, such as a solid, a line, and so on.
2. Iterate the Object property to get a geometry solid instance containing all geometry faces and edges in the Faces and Edges properties.
3. Iterate the Faces property to get all geometry faces.
4. Iterate the Edges property to get all geometry edges.

The sample code follows:

Code Region 20-2: Retrieving faces and edges

```
private void GetFacesAndEdges(Wall wall)
{
    String faceInfo = "";

    Autodesk.Revit.DB.Options opt = new Options();
    Autodesk.Revit.DB.GeometryElement geomElem = wall.get_Geometry(opt);
    foreach (GeometryObject geomObj in geomElem.Objects)
    {
        Solid geomSolid = geomObj as Solid;
        if (null != geomSolid)
        {
            int faces = 0;
            double totalArea = 0;
            foreach (Face geomFace in geomSolid.Faces)
            {
                faces++;
                faceInfo += "Face " + faces + " area: " + geomFace.Area.ToString() + "\n";
                totalArea += geomFace.Area;
            }
            faceInfo += "Number of faces: " + faces + "\n";
            faceInfo += "Total area: " + totalArea.ToString() + "\n";
            foreach (Edge geomEdge in geomSolid.Edges)
            {
                // get wall's geometry edges
            }
        }
    }
    TaskDialog.Show("Revit", faceInfo);
}
```

20.2 Geometry Node Class

The Geometry Node class describes the graphical representation in the API. Eight Geometry Node classes are in the API. The classes are:

- Profile - A geometric profile consisting of a loop of curves.
- Face - A 3D solid face. Some derived classes give detailed faces.
- Edge - A 3D solid edge.

- Curve - A parametric curve. Some derived classes provide detailed curves.
- Element - Geometric representation of an element containing all geometry information in its property. This element is not an Element type.
- Mesh - A triangular mesh used to describe the shape of a 3D face.
- Instance - An instance of another element (symbol). Specifically, you can retrieve a symbol's geometry information from the instance referring to it.
- Solid - 3D solid.

20.2.1 Geometry.Instance

The Instance class represents an instance of another element (symbol), specially positioned by this element. It is used to store geometry information for the symbol. The Instance class can pack any geometry information data, including another instance, making it a good way to build a geometry tree using the instance in a complicated geometry representation.

The Instance class stores geometry data in the SymbolGeometry property using a local coordinate system. It also provides a Transform instance to convert the local coordinate system to a world coordinate space. To get the same geometry data as the Revit application from the Instance class, use the transform property to convert each geometry object retrieved from the SymbolGeometry property.

Generally, three geometry objects can be retrieved from an instance. They are:

- Curve – Curve or its derived class.
- Solid – Contains faces and edges.
- Instance – Another instance which forms this instance.

Users need to transform the retrieved geometry objects using the Transform property. For more details, refer to the Geometry.Transform section in this chapter.

Two samples are presented to explain the main usage.

20.2.1.1 Sample 1

Get curves from an instance, and perform the coordinate transformation.

Code Region 20-3: Getting curves from an instance

```
public void GetAndTransformCurve(Autodesk.Revit.ApplicationServices.Application app,
           Autodesk.Revit.DB.Element element, Options geoOptions)
{
   // Get geometry element of the selected element
   Autodesk.Revit.DB.GeometryElement geoElement = element.get_Geometry(geoOptions);

   // Get geometry object
   foreach (GeometryObject geoObject in geoElement.Objects)
   {
       // Get the geometry instance which contains the geometry information
       Autodesk.Revit.DB.GeometryInstance instance =
                  geoObject as Autodesk.Revit.DB.GeometryInstance;
       if (null != instance)
       {
           foreach (GeometryObject o in instance.SymbolGeometry.Objects)
           {
               // Get curve
```

```
                Curve curve = o as Curve;
                if (curve != null)
                {
                    // transfrom the curve to make it in the instance's coordinate space
                    curve = curve.get_Transformed(instance.Transform);
                }
            }
        }
    }
}
```

20.2.1.2 **Sample 2**

Get the solid information from an instance, and perform the coordinate transformation.

Code Region 20-4: Getting solid information from an instance

```
private void GetAndTransformSolidInfo(Application application, Element element,
                                      Options geoOptions)
{
    // Get geometry element of the selected element
    Autodesk.Revit.DB.GeometryElement geoElement = element.get_Geometry(geoOptions);

    // Get geometry object
    foreach (GeometryObject geoObject in geoElement.Objects)
    {
        // Get the geometry instance which contains the geometry information
        Autodesk.Revit.DB.GeometryInstance instance =
            geoObject as Autodesk.Revit.DB.GeometryInstance;
        if (null != instance)
        {
            foreach (GeometryObject instObj in instance.SymbolGeometry.Objects)
            {
                Solid solid = instObj as Solid;
                if (null == solid || 0 == solid.Faces.Size || 0 == solid.Edges.Size)
                {
                    continue;
                }

                Transform instTransform = instance.Transform;
                // Get the faces and edges from solid, and transform the formed points
                foreach (Face face in solid.Faces)
                {
                    Mesh mesh = face.Triangulate();
                    foreach (XYZ ii in mesh.Vertices)
                    {
                        XYZ point = ii;
                        XYZ transformedPoint = instTransform.OfPoint(point);
                    }
```

```
                }
                foreach (Edge edge in solid.Edges)
                {
                    foreach (XYZ ii in edge.Tessellate())
                    {
                        XYZ point = ii;
                        XYZ transformedPoint = instTransform.OfPoint(point);
                    }
                }
            }
        }
    }
}
```

Note: You find similar situations when you get geometry information for doors, windows, and other elements. For more details about the retrieved geometry, refer to Example: Retrieving Geometry Data from a Beam in this chapter.

20.2.2 Geometry.Mesh

Geometry mesh is a triangular mesh used to describe the shape of a 3D face. The face object can be a curved surface which can be described by its endpoints just like a polygon. As a result, the mesh object provides a way to split the face into many small triangles where the endpoints locate the split face.

The following code sample illustrates how to get the geometry of a mass. The mass geometry faces are described by the mesh.

Code Region 20-5: Drawing the geometry of a mass

```
private void TriangulateFace(Face face)
{
    // Get mesh
    Mesh mesh = face.Triangulate();
    for (int i = 0; i < mesh.NumTriangles; i++)
    {
        MeshTriangle triangle = mesh.get_Triangle(i);
        XYZ vertex1 = triangle.get_Vertex(0);
        XYZ vertex2 = triangle.get_Vertex(1);
        XYZ vertex3 = triangle.get_Vertex(2);
    }
}
```

The following pictures illustrate how a mass looks in the Revit application (left picture) and the same mass drawn with the API (right picture).

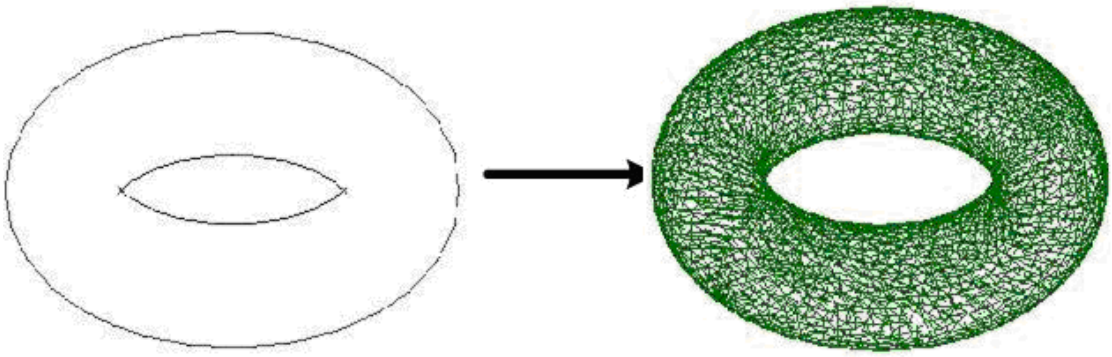


Figure 110: Draw a mass using the API

Note: The approximation tolerance used for display purposes is defined internally by Revit including distance tolerance and angle tolerance.

20.2.3 Solid

The Solid class defines a 3D solid from which users can get faces and edges. The Solid class defines a geometry solid, such as a cube or a cylinder. You can get the following using the solid's properties:

- Faces
- Edges
- Surface area
- Volume

Note: Sometimes the API can have unused solids containing zero edges and faces. Check the Edges and Faces properties before completing further work.

Code Region 20-6: Getting a solid from a GeometryObject

```
// instObj is a GeometryObject
Solid solid = instObj as Solid;
if (null == solid || 0 == solid.Faces.Size || 0 == solid.Edges.Size)
{
    continue;
}
```

20.3 Geometry Helper Class

Several Geometry Helper classes are in the API. The Helper classes are used to describe geometry information for certain elements, such as defining a CropBox for a view using the BoundingBoxXYZ class.

- BoundingBoxXYZ - A 3D rectangular box used in cases such as defining a 3D view section area.
- Transform - Transforming the affine 3D space.
- Reference - A stable reference to a geometric object in a Revit model, which is used when creating elements like dimensions.
- Plane – A flat surface in geometry.
- Options - User preferences for parsing geometry.

- XYZ - Object representing coordinates in 3D space.
- UV - Object representing coordinates in 2D space.
- BoundingBoxUV - A 2D rectangle parallel to the coordinate axes.

20.3.1 Geometry.Transform

Transforms are limited to 3x4 transformations (Matrix) in the Revit application, transforming an object's place in the model space relative to the rest of the model space and other objects. The transforms are built from the position and orientation in the model space. Three direction Vectors (BasisX, BasisY and BasisZ properties) and Origin point provide all of the transform information. The matrix formed by the four values is as follows:

$$\begin{pmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} & \text{Origin.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} & \text{Origin.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} & \text{Origin.Z} \end{pmatrix}$$

Applying the transformation to the point is as follows:

$$\begin{pmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} & \text{Origin.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} & \text{Origin.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} & \text{Origin.Z} \end{pmatrix} \times \begin{pmatrix} XYZ.X \\ XYZ.Y \\ XYZ.Z \\ 1 \end{pmatrix}$$

The Transform OfPoint method implements the previous function. The following code is a sample of the same transformation process.

Code Region 20-7: Transformation example

```
public static XYZ TransformPoint(XYZ point, Transform transform)
{
    double x = point.X;
    double y = point.Y;
    double z = point.Z;

    //transform basis of the old coordinate system in the new coordinate // system
    XYZ b0 = transform.get_Basis(0);
    XYZ b1 = transform.get_Basis(1);
    XYZ b2 = transform.get_Basis(2);
    XYZ origin = transform.Origin;

    //transform the origin of the old coordinate system in the new
    //coordinate system
    double xTemp = x * b0.X + y * b1.X + z * b2.X + origin.X;
    double yTemp = x * b0.Y + y * b1.Y + z * b2.Y + origin.Y;
    double zTemp = x * b0.Z + y * b1.Z + z * b2.Z + origin.Z;

    return new XYZ(xTemp, yTemp, zTemp);
}
```

The Geometry.Transform class properties and methods are identified in the following sections.

20.3.1.1Identity
Transform the Identity.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

20.3.1.2Reflection
Reflect a specified plane.

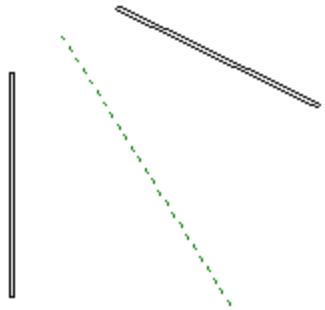


Figure 111: Wall Reflection relationship

As the previous picture shows, one wall is mirrored by a reference plane. The Reflection property needs the geometry plane information for the reference plane.

Code Region 20-8: Using the Reflection property

```
private Transform Reflect(ReferencePlane refPlane)
{
    Transform mirTrans = Transform.get_Reflection(refPlane.Plane);

    return mirTrans;
}
```

20.3.1.3**Rotation**
Rotate by a specified angle around a specified axis and point.

20.3.1.4**Translation**
Translate by a specified vector. Given a vector XYZ data, a transformation is created as follow:

$$\begin{pmatrix} x & y & z \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \end{pmatrix}$$

20.3.1.5 **Determinant**

Transformation determinant.

$$\begin{vmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} \end{vmatrix}$$

20.3.1.6 **HasReflection**

This is a Boolean value that indicates whether the transformation produces a reflection.

20.3.1.7 **Scale**

A value that represents the transformation scale.

20.3.1.8 **Inverse**

An inverse transformation. Transformation matrix A is invertible if a transformation matrix B exists such that A*B = B*A = I (identity).

20.3.1.9 **IsIdentity**

Boolean value that indicates whether this transformation is an identity.

20.3.1.10 **IsTranslation**

Boolean value that indicates whether this transformation is a translation.

Geometry.Transform provides methods to perform basal matrix operations.

20.3.1.11 **Multiply**

Multiplies a transformation by a specified transformation and returns the result.

Operator* - Multiplies two specified transforms.

20.3.1.12 **ScaleBasis**

Scales the basis vectors and returns the result.

20.3.1.13 **ScaleBasisAndOrigin**

Scales the basis vectors and the transformation origin returns the result.

20.3.1.14 **OfPoint**

Applies the transformation to the point. The Origin property is used.

20.3.1.15 **OfVector**

Applies the transform to the vector. The Origin property is not used.

20.3.1.16 **AlmostEqual**

Compares two transformations. AlmostEqual is consistent with the computation mechanism and accuracy in the Revit core code. Additionally, Equal and the == operator are not implemented in the Transform class.

The API provides several shortcuts to complete geometry transformation. The Transformed property in several geometry classes is used to do the work, as shown in the following table.

Class Name	Function Description

Class Name	Function Description
Curve.get_Transformed(Transform transform)	Applies the specified transformation to a curve
Instance.get_Transformed(Transform transform)	Transforms the instance.
Profile.get_Transformed(Transform transform)	Transforms the profile and returns the result.
Mesh.get_Transformed(Transform transform)	Transforms the mesh and returns the result.

Table 47: Transformed Methods

Note: The transformed method clones itself then returns the transformed cloned result.

20.3.2 Geometry.Reference

The Reference class does not contain properties or methods. However, it is very useful in element creation.

- Dimension creation requires references.
- The reference identifies a path within a geometric representation tree in a flexible manner.
- The tree is used to view specific geometric representation creation.

The API exposes four types of references based on different Pick pointer types. They are retrieved from the API in different ways:

1. For Point – Curve.EndPointReference property
2. For Curve (Line, Arc, and etc.) - Curve.Reference property
3. For Face – Face.Reference property
4. For Cut Edge – Edge.Reference property

Different reference types cannot be used arbitrarily. For example:

- The NewLineBoundaryConditions() method requires a reference for Line
- The NewAreaBoundaryConditions() method requires a reference for Face
- The NewPointBoundaryConditions() method requires a reference for Point.

20.3.3 Geometry.Options

Geometry is typically extracted from the indexed property Element.Geometry. This property accepts an options class which you must supply. The options class customizes the type of output you receive based on its properties:

- ComputeReferences – Indicates whether to compute the geometry reference when retrieving geometry information. The default value is false, so if this property is not set to true, the reference will not be accessible.
- IncludeNonVisibleObjects – Indicates to also return geometry objects which are not visible in a default view.
- View - Gets geometry information from a specific view. Note that if a view is assigned, the detail level for this view will be used in place of "DetailLevel".
- DetailLevel – Indicates the preferred detail level. The default is Medium.

20.3.3.1 ComputeReferences

If you set this property to false, the API does not compute a geometry reference. All Reference properties retrieved from the geometry tree return nothing. For more details about references, refer to the Reference section.

20.3.3.2**IncludeNonVisibleObjects**

Most of the non-visible geometry is construction and conditional geometry that the user sees when editing the element (i.e., the center plane of a window family instance). The default for this property is false. However, some of this conditionally visible geometry represents real-world objects, such as insulation surrounding ducts in Revit MEP, and it should be extracted.

20.3.3.3**View**

If users set the View property to a different view, the retrieved geometry information can be different. Review the following examples for more information:

1. In Revit, draw a stair in 3D view then select the Crop Region, Crop Region Visible, and Section Box properties in the 3D view. In the Crop Region, modify the section box in the 3D view to display a portion of the stair. If you get the geometry information for the stair using the API and set the 3D view as the Options.View property, only a part of the stair geometry can be retrieved. The following pictures show the stair in the Revit application (left) and one drawn with the API (right).

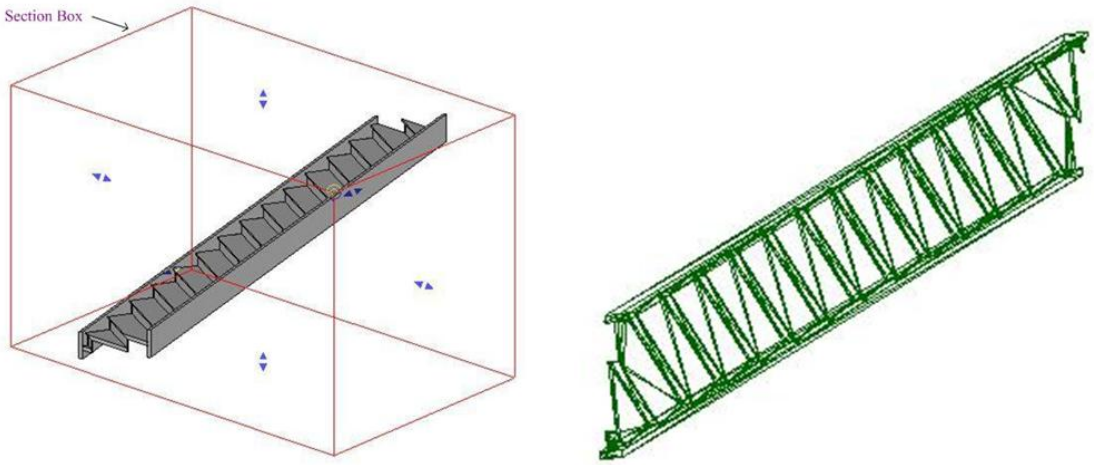


Figure 112: Different section boxes display different geometry

Draw a stair in Revit then draw a section as shown in the left picture. If you get the information for this stair using the API and set this section view as the Options.View property, only a part of the stair geometry can be retrieved. The stair drawn with the API is shown in the right picture.

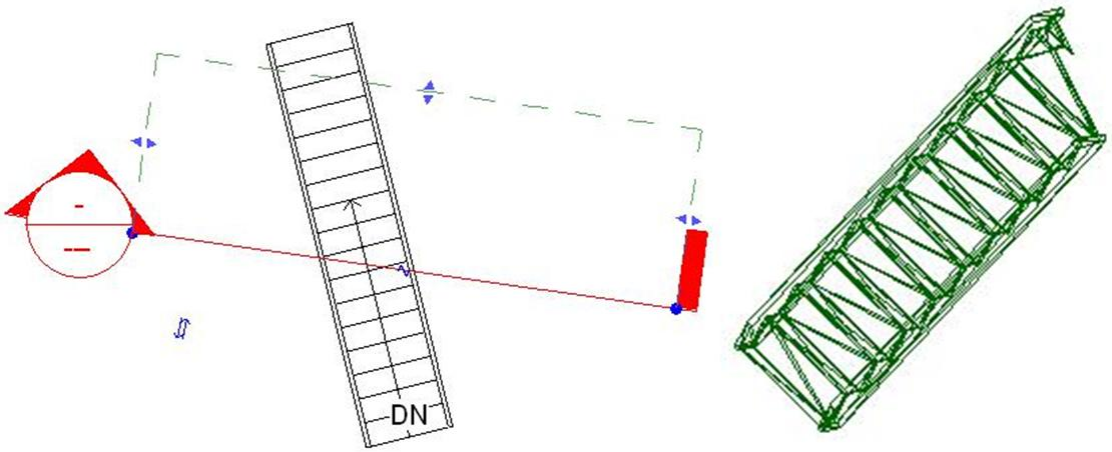


Figure 113: Retrieve Geometry section view

20.3.3.4 DetailLevel

The API defines three enumerations in Geometry.Options.DetailLevels. The three enumerations correspond to the three Detail Levels in the Revit application, shown as follows.

Figure 114: Three detail levels

Different geometry information is retrieved based on different settings in the DetailLevel property. For example, draw a beam in the Revit application then get the geometry from the beam using the API to draw it. The following pictures show the drawing results:

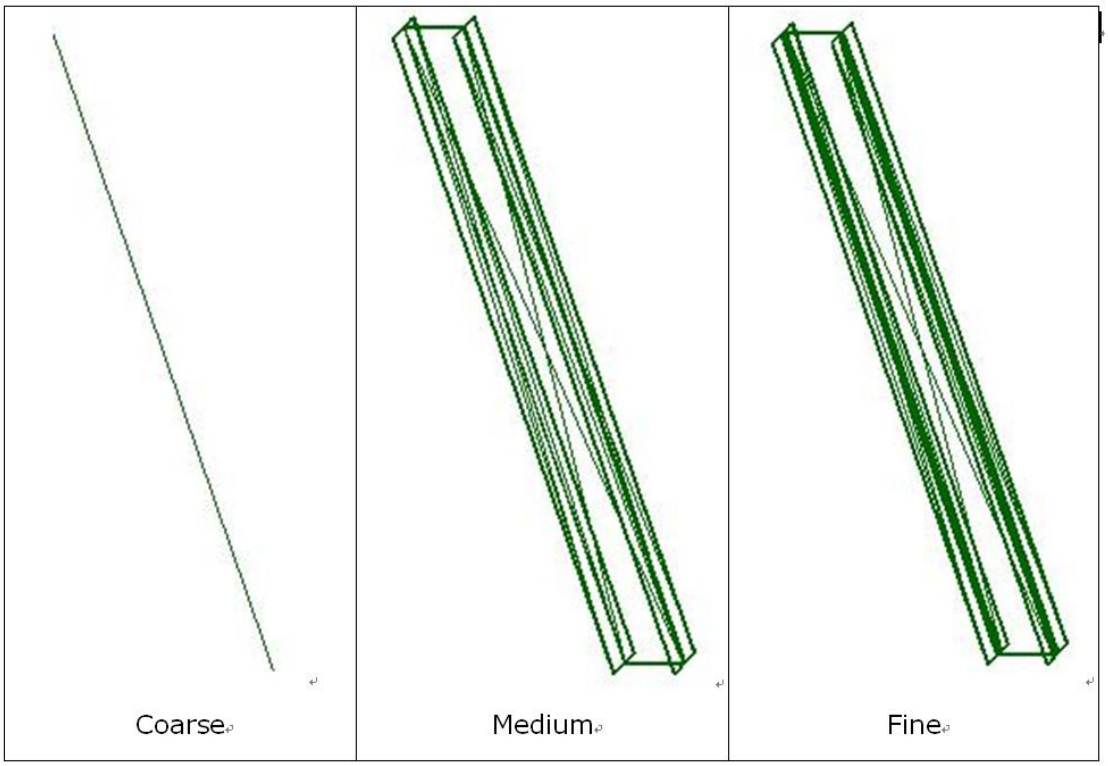


Figure 115: Detail geometry for a beam

20.3.4 Geometry.BoundingBoxXYZ

BoundingBoxXYZ defines a 3D rectangular box that is required to be parallel to any coordinate axis. Similar to the Instance class, the BoundingBoxXYZ stores data in the local coordinate space. It has a Transform property that transforms the data from the box local coordinate space to the model space. In other words, to get the box boundary in the model space (the same one in Revit), transform each data member using the Transform property. The following sections illustrate how to use BoundingBoxXYZ.

20.3.4.1 Define the View Boundaries

BoundingBoxXYZ can be used to define the view boundaries through View.CropBox property. The following pictures use a section view to show how BoundingBoxXYZ is used in the Revit application.

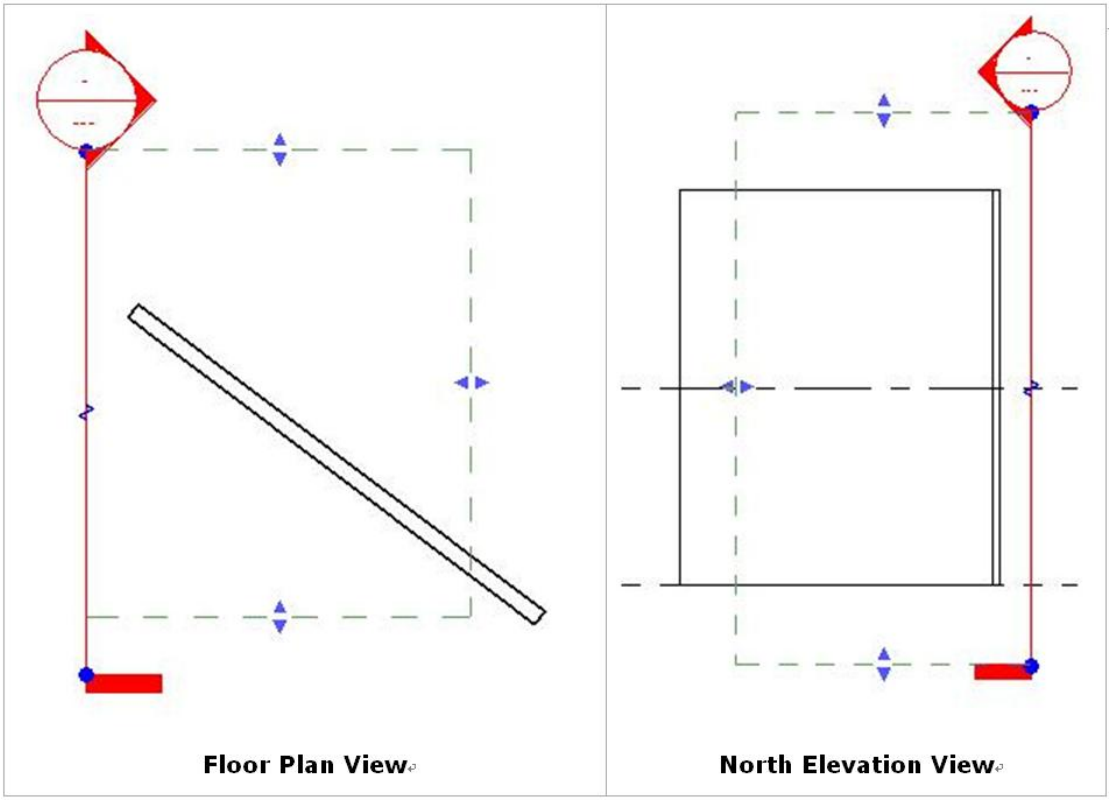


Figure 116: BoundingBoxXYZ in section view

The dash lines in the previous pictures show the section view boundary exposed as the CropBox property (a BoundingBoxXYZ instance).

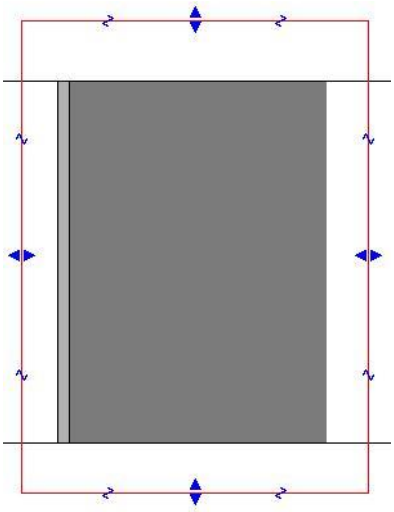


Figure 117: Created section view

The previous picture displays the corresponding section view. The wall outside the view boundary is not displayed.

20.3.4.2**Define a Section Box**

BoundingBoxXYZ is also used to define a section box for a 3D view retrieved from the View3D.SectionBox property. Select the Section Box property in the Properties Dialog box. The section box is shown as follows:

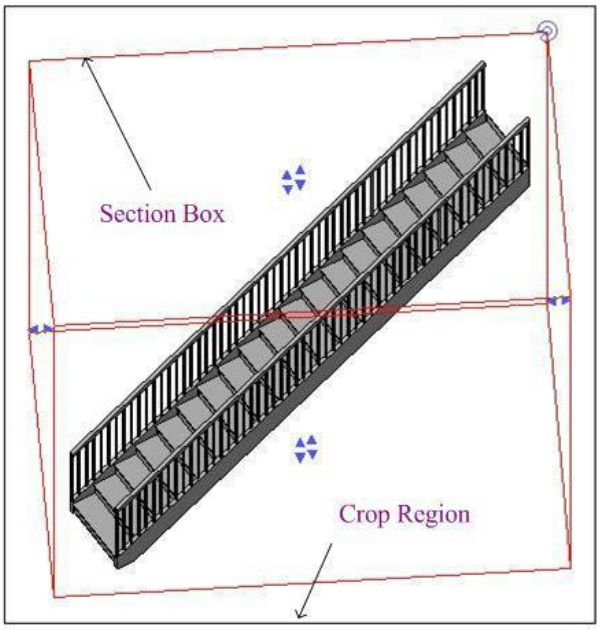


Figure 118: 3D view section box

20.3.4.3**Other Uses**

- Defines a box around an element's geometry. (Element.BoundingBox Property). The BoundingBoxXYZ instance retrieved in this way is parallel to the coordinate axes.
- Used in the NewViewSection() method in the Creation.Document class.

The following table identifies the main uses for this class.

Property Name	Usage
Max/Min	Maximum/Minimum coordinates. These two properties define a 3D box parallel to any coordinate axis. The Transform property provides a transform matrix that can transform the box to the appropriate position.
Transform	Transform from the box coordinate space to the model space.
Enabled	Indicates whether the bounding box is turned on.

<table>
<tr><th>Property Name</th><th>Usage</th></tr>
<tr><td>MaxEnabled/ MinEnabled</td><td>Defines whether the maximum/minimum bound is active for a given dimension.

If the Enable property is false, these two properties should also return false.

<table>
<tr><td> 
</td><td>If the crop view is turned on, both MaxEnabled property and MinEnabled property return true.</td></tr>
<tr><td> 
</td><td>If the crop view is turned off, both MaxEnabled property and MinEnabled property return false.</td></tr>
</table>

This property indicates whether the view’s crop box face can be used to clip the element’s view.
If BoundingBoxXYZ is retrieved from the View3D.SectionBox property, the return value depends on whether the Section Box property is selected in the 3D view Properties dialog box. If so, all Enabled properties return true.
If BoundingBoxXYZ is retrieved from the Element.BoundingBox property, all the Enabled properties are true.
</td></tr>
<tr><td>Bounds</td><td>Wrapper for the Max/Min properties.</td></tr>
<tr><td>BoundEnabled</td><td>Wrapper for the MaxEnabled/MinEnabled properties.</td></tr>
</table>

Table 48: BoundingBoxXYZ properties

The following code sample illustrates how to rotate BoundingBoxXYZ to modify the 3D view section box.

Code Region 20-9: Rotating BoundingBoxXYZ

```
private void RotateBoundingBox(View3D view3d)
{
    BoundingBoxXYZ box = view3d.SectionBox;
    if (false == box.Enabled)
    {
        TaskDialog.Show("Revit", "The section box for View3D isn't Enable.");
        return;
    }
    // Create a rotation transform,
    XYZ origin = new XYZ(0, 0, 0);
    XYZ axis = new XYZ(0, 0, 1);
    Transform rotate = Transform.get_Rotation(origin, axis, 2);
    // Transform the View3D's SectionBox with the rotation transfrom
    box.Transform = box.Transform.Multiply(rotate);
    view3d.SectionBox = box;
}
```

20.3.5 Geometry.BoundingBoxUV

BoundingBoxUV is a value class that defines a 2D rectangle parallel to the coordinate axes. It supports the Min and Max data members. Together they define the BoundingBoxUV's boundary. BoundingBoxUV is retrieved from the View.Outline property which is the boundary view in the paper space view.

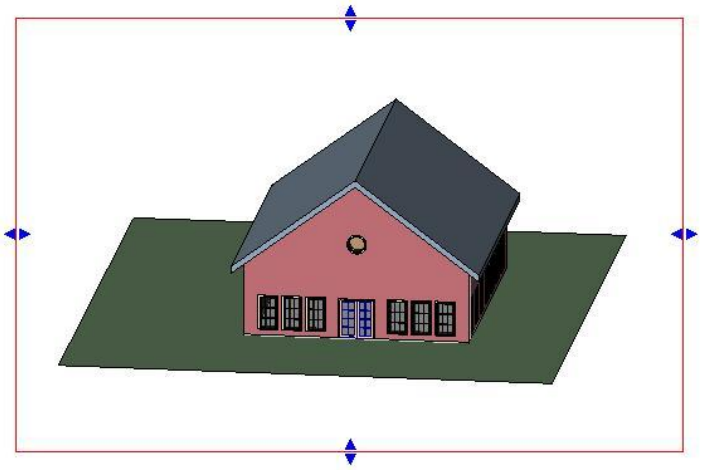


Figure 119: View outline

Two points define a BoundingBoxUV.

- Min point - The bottom-left endpoint.
- Max point - The upper-right endpoint.

Figure 120: BoundingBoxUV Max and Min

Note: BoundingBoxUV cannot present a gradient rectangle as the following picture shows.

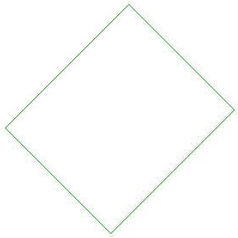


Figure 121: Gradient rectangle

20.4 Collection Classes

The API provides the following collection classes based on the items they contain:

Class/Type	Corresponding Collection Classes	Corresponding Iterators
Curve	CurveArray	CurveArrayIterator

Class/Type	Corresponding Collection Classes	Corresponding Iterators
Edge	EdgeArray, EdgeArrayArray	EdgeArrayIterator, EdgeArrayArrayIterator
Face	FaceArray	FaceArrayIterator
GeometryObject	GeometryObjectArray	GeometryObjectArrayIterator
Instance	InstanceArray	InstanceArrayIterator
Mesh	MeshArray	MeshArrayIterator
Reference	ReferenceArray	ReferenceArrayIterator
Solid	SolidArray	SolidArrayIterator
Double value	DoubleArray	DoubleArrayIterator

Table 49: Geometry Collection Classes

All of these classes use very similar methods and properties to do similar work. For more details, refer to the Collection chapter.

20.5 Example: Retrieve Geometry Data from a Beam

This section illustrates how to get solids and curves from a beam. You can retrieve column and brace geometry data in a similar way.

Note: If you want to get the beam and brace driving curve, call the FamilyInstance Location property where a LocationCurve is available.

The sample code is shown as follows:

Code Region 20-10: Getting solids and curves from a beam

```
public void GetCurvesFromABeam(Autodesk.Revit.DB.FamilyInstance beam,
                               Autodesk.Revit.DB.Options options)
{
    Autodesk.Revit.DB.GeometryElement geomElem = beam.get_Geometry(options);

    Autodesk.Revit.DB.CurveArray curves = new CurveArray();
    Autodesk.Revit.DB.SolidArray solids = new SolidArray();

    //Find all solids and insert them into solid array
    AddCurvesAndSolids(geomElem, ref curves, ref solids);
}

private void AddCurvesAndSolids(Autodesk.Revit.DB.GeometryElement geomElem,
                                ref Autodesk.Revit.DB.CurveArray curves,
                                ref Autodesk.Revit.DB.SolidArray solids)
{
    foreach (Autodesk.Revit.DB.GeometryObject geomObj in geomElem.Objects)
    {
        Autodesk.Revit.DB.Curve curve = geomObj as Autodesk.Revit.DB.Curve;
        if (null != curve)
        {
            curves.Append(curve);
            continue;
```

```
        }
        Autodesk.Revit.DB.Solid solid = geomObj as Autodesk.Revit.DB.Solid;
        if (null != solid)
        {
            solids.Append(solid);
            continue;
        }
        //If this GeometryObject is Instance, call AddCurvesAndSolids
        Autodesk.Revit.DB.GeometryInstance geomInst =
            geomObj as Autodesk.Revit.DB.GeometryInstance;
        if (null != geomInst)
        {
            Autodesk.Revit.DB.GeometryElement transformedGeomElem
              = geomInst.GetInstanceGeometry(geomInst.Transform);
            AddCurvesAndSolids(transformedGeomElem, ref curves, ref solids);
        }
    }
}
```

Note: For more information about how to retrieve the Geometry.Options type object, refer to the Geometry.Options section in this chapter.

21 Place and Locations

Every building has a unique place in the world because the Latitude and Longitude are unique. In addition, a building can have many locations in relation to other buildings. The Revit Platform API Site namespace uses certain classes to save the geographical location information for Revit projects.

Note: The Revit Platform API does not expose the Site menu functions. Only Site namespace provides functions corresponding to the Location options found on the Project Location panel on the Manage tab.

21.1 Place

In the Revit Platform API, the SiteLocation class contains place information including Latitude, Longitude, and Time Zone. This information identifies where the project is located in the world.

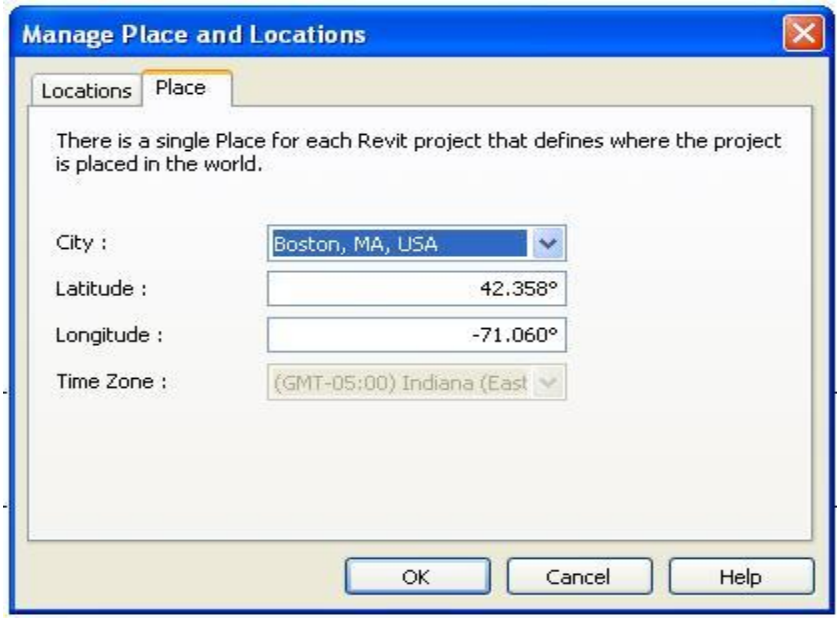


Figure 122: Project Place

21.2 City

City is an object that contains geographical location information for a known city in the world. It contains longitude, latitude, and time zone information. The city list is retrieved by the Cities property in the Application object. New cities cannot be added to the existing list in Revit. The city where the current project is located is not exposed by the Revit Platform API.

21.3 ProjectLocation

A project only has one site which is the absolute location on the earth. However, it can have different locations relative to the projects around it. Depending on the coordinates and origins in use, there can be many ProjectLocation objects in one project.

By default each Revit project contains at least one named location, Internal. It is the active project location. You can retrieve it using the Document.ActiveProjectLocation property. All existing ProjectLocation objects are retrieved using the Document.ProjectLocations property.

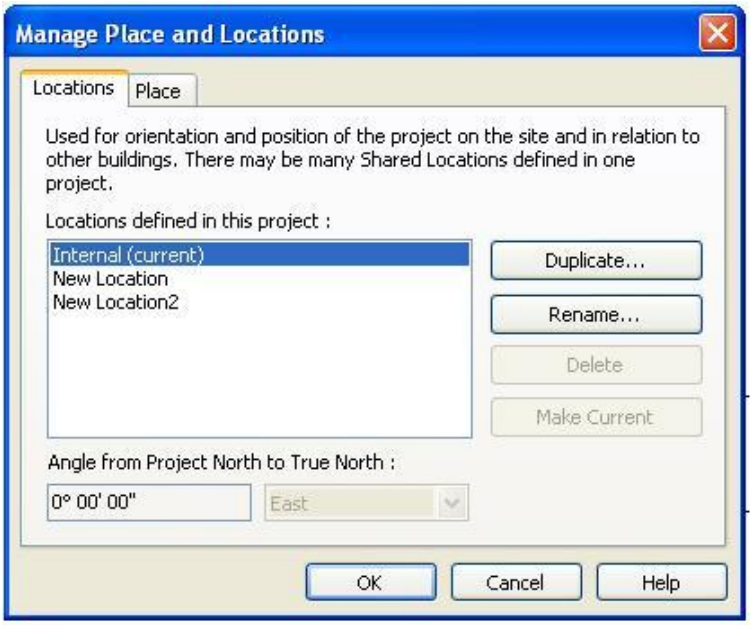


Figure 123: Project locations

21.4 Project Position

Project position is an object that represents a geographical offset and rotation. It is usually used by the ProjectLocation object to get and set geographical information. The following figure shows the results after changing the ProjectLocation geographical rotation and the coordinates for the same point. However, you cannot see the result of changing the ProjectLocation geographical offset directly.

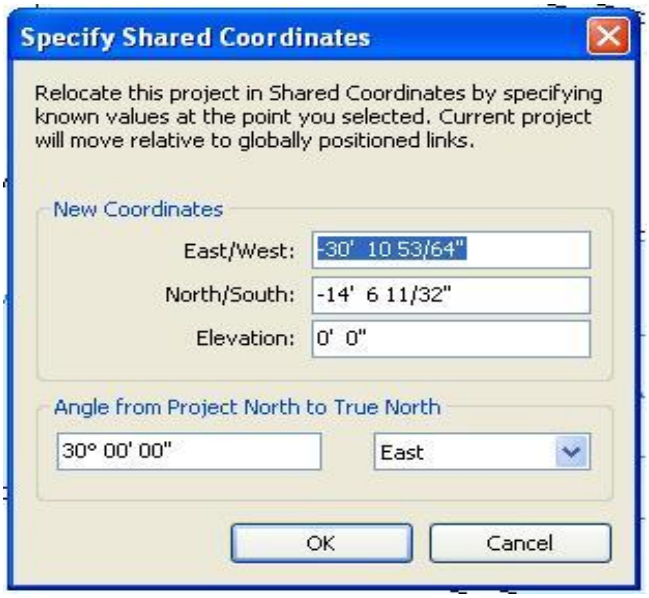


Figure 124: Point coordinates

Note: East indicates that the Location is rotated counterclockwise; West indicates that the location is rotated clockwise. If the Angle value is between 180 and 360 degrees, Revit transforms it automatically. For example, if you select East and type 200 degrees for Angle, Revit transforms it to West 160 degrees

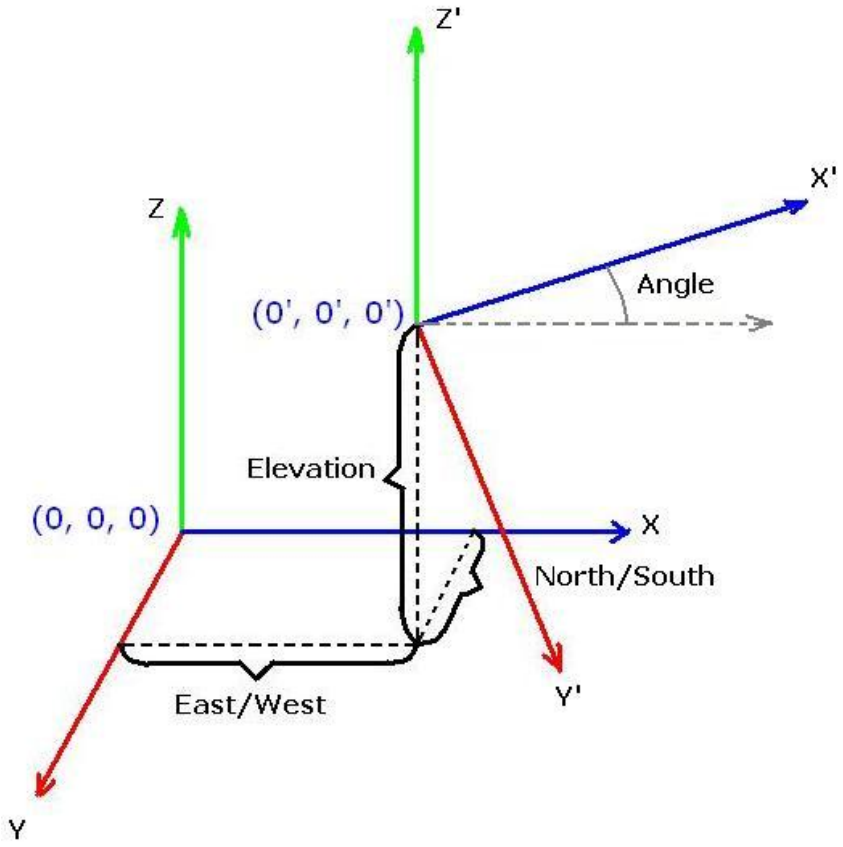


Figure 125: Geographical offset and rotation sketch map

The following sample code illustrates how to retrieve the ProjectLocation object.

Code Region 21-1: Retrieving the ProjectLocation object

```
public void ShowActiveProjectLocationUsage(Autodesk.Revit.DB.Document document)
{
    // Get the project location handle
    ProjectLocation projectLocation = document.ActiveProjectLocation;

    // Show the information of current project location
    XYZ origin = new XYZ(0, 0, 0);
    ProjectPosition position = projectLocation.get_ProjectPosition(origin);
    if (null == position)
    {
        throw new Exception("No project position in origin point.");
    }

    // Format the prompt string to show the message.
    String prompt = "Current project location information:\n";
    prompt += "\n\t" + "Origin point position:";
    prompt += "\n\t\t" + "Angle: " + position.Angle;
    prompt += "\n\t\t" + "East to West offset: " + position.EastWest;
    prompt += "\n\t\t" + "Elevation: " + position.Elevation;
    prompt += "\n\t\t" + "North to South offset: " + position.NorthSouth;
```

```
    // Angles are in radians when coming from Revit API, so we
    // convert to degrees for display
    const double angleRatio = Math.PI / 180;   // angle conversion factor

    SiteLocation site = projectLocation.SiteLocation;
    prompt += "\n\t" + "Site location:";
    prompt += "\n\t\t" + "Latitude: " + site.Latitude / angleRatio + "¡ã";
    prompt += "\n\t\t" + "Longitude: " + site.Longitude / angleRatio + "¡ã";
    prompt += "\n\t\t" + "TimeZone: " + site.TimeZone;

    // Give the user some information
    TaskDialog.Show("Revit", prompt);
}
```

Note: There is only one active project location at a time. To see the result after changing the ProjectLocation geographical offset and rotation, change the Orientation property from Project North to True North in the plan view Properties dialog box.

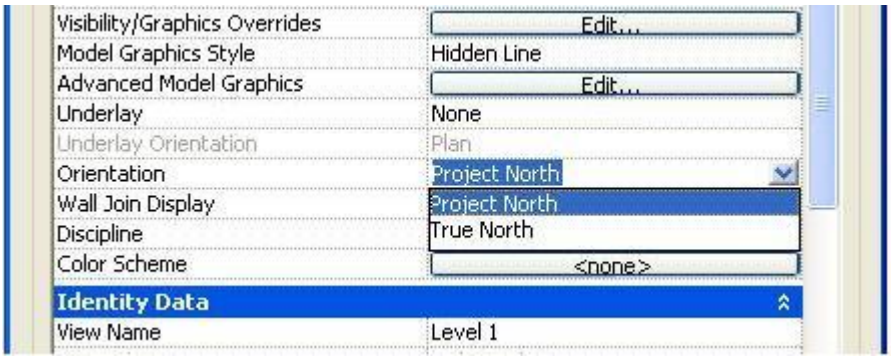


Figure 126: Set orientation value in plan view Properties dialog box

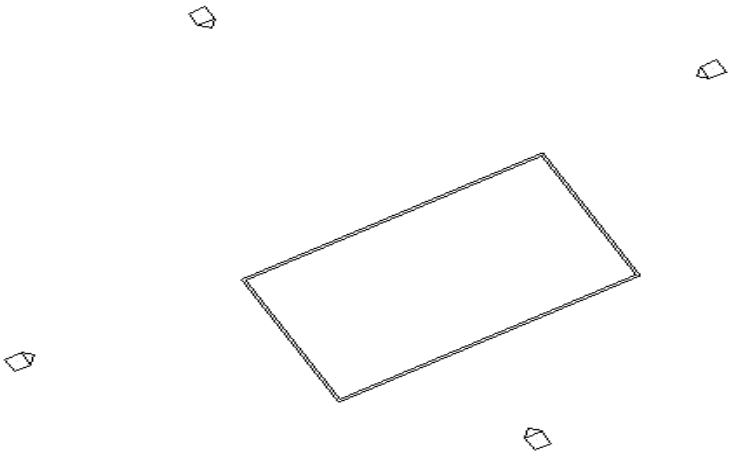


Figure 127: Project is rotated 30 degrees from Project North to True North

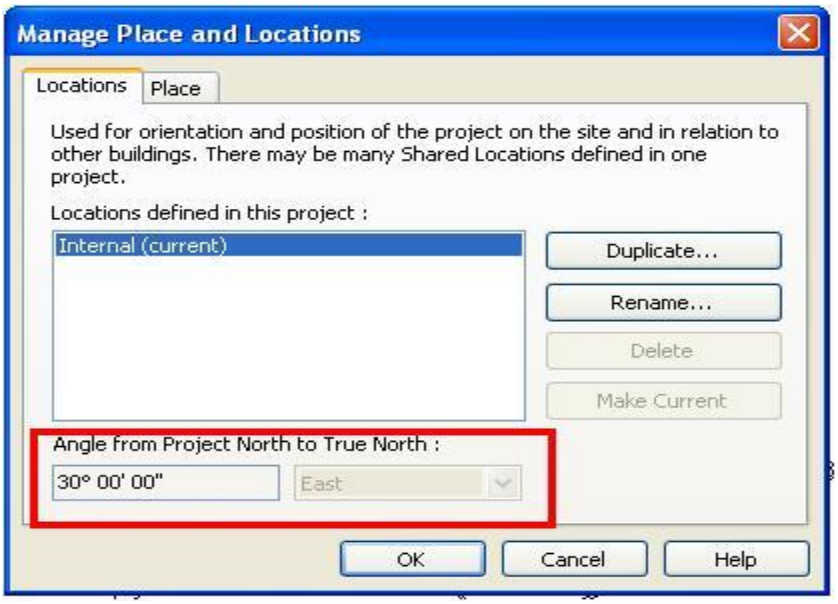


Figure 128: Project location information

21.4.1 Creating and Deleting Project Locations

Create new project locations by duplicating an existing project location using the Duplicate() method. The following code sample illustrates how to create a new project location using the Duplicate() method.

Code Region 21-2: Creating a project location

```
public ProjectLocation DuplicateLocation(Autodesk.Revit.DB.Document document, string newName)
{
    ProjectLocation currentLocation = document.ActiveProjectLocation;
    ProjectLocationSet locations = document.ProjectLocations;
    foreach (ProjectLocation projectLocation in locations)
    {
        if (projectLocation.Name == newName)
        {
            throw new Exception("The name is same as a project location's name, please change
one.");
        }
    }
    return currentLocation.Duplicate(newName);
}
```

The following code sample illustrates how to delete an existing project location from the current project.

Code Region 21-3: Deleting a project location

```
public void DeleteLocation(Autodesk.Revit.DB.Document document)
{
    ProjectLocation currentLocation = document.ActiveProjectLocation;
    //There must be at least one project location in the project.
    ProjectLocationSet locations = document.ProjectLocations;
```

```
    if (1 == locations.Size)
    {
        return;
    }

    string name = "location";
    if (name != currentLocation.Name)
    {
        foreach (ProjectLocation projectLocation in locations)
        {
            if (projectLocation.Name == name)
            {
                ICollection<ElementId> elemSet = document.Delete(projectLocation);
                if (elemSet.Count > 0)
                {
                    TaskDialog.Show("Revit", "Project Location Deleted!");
                }
            }
        }
    }
}
```

Note: The following rules apply to deleting a project location:

- The active project location cannot be deleted because there must be at least one project location in the project.
- You cannot delete the project location if the ProjectLocationSet class instance is read-only.

22 Shared Parameters

Shared Parameters are parameter definitions stored in an external text file. The definitions are identified by a unique identifier generated when the definition is created and can be used in multiple projects.

This chapter introduces how to gain access to shared parameters through the Revit Platform API. The following overview shows how to get a shared parameter and bind it to Elements in certain Categories:

- Set SharedParametersFileName
- Get the External Definition
- Binding

22.1 Definition File

The DefinitionFile object represents a shared parameter file. The definition file is a common text file. Do not edit the definition file directly; instead, edit it using the UI or the API.

22.1.1 Definition File Format

The shared parameter definition file is a text file (.txt) with two blocks: GROUP and PARAM.

Code Region 22-1: Parameter definition file example

```
# This is a Revit shared parameter file.
# Do not edit manually.
*GROUP ID     NAME
GROUP  1      MyGroup
GROUP  2      AnotherGroup
*PARAM GUID   NAME   DATATYPE      DATACATEGORY  GROUP  VISIBLE
PARAM  4b217a6d-87d0-4e64-9bbc-42b69d37dda6    MyParam       TEXT          1      1
PARAM  34b5cb95-a526-4406-806d-dae3e8c66fa9    Price  INTEGER             2      1
PARAM  05569bb2-9488-4be4-ae21-b065f93f7dd6    areaTags      FAMILYTYPE    -2005020      1
       1
```

- The GROUP block contains group entries that associate every parameter definition with a group. The following fields appear in the GROUP block:
  - ID - Uniquely identifies the group and associates the parameter definition with a group.
  - Name - The group name displayed in the UI.

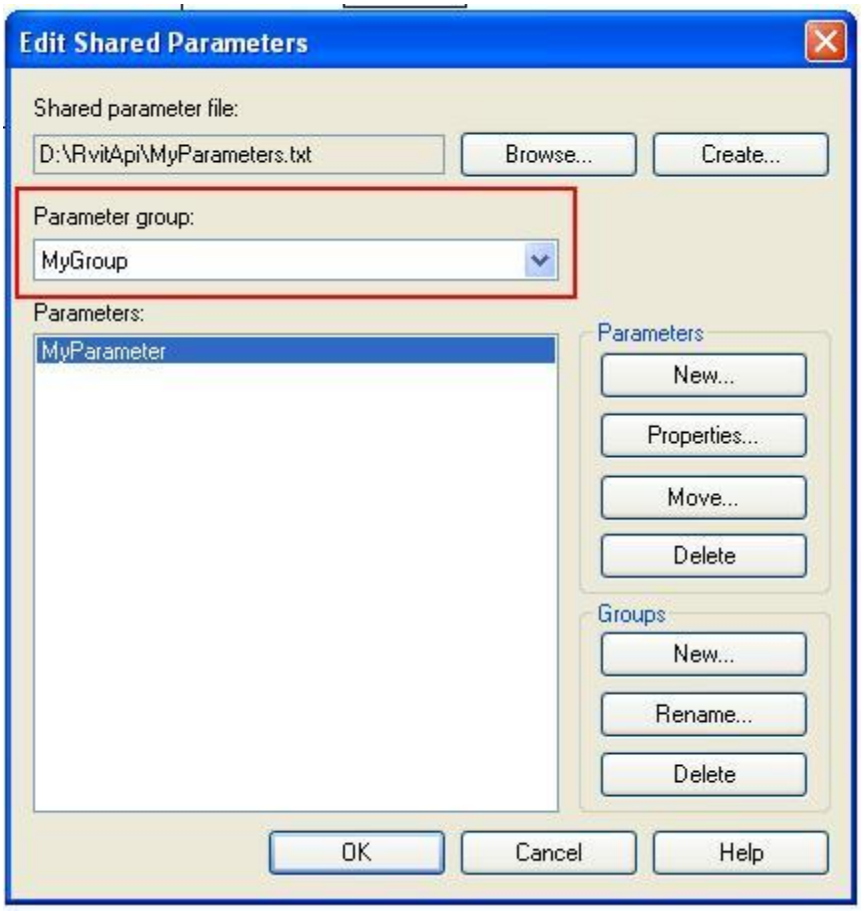


Figure 129: Edit Shared Parameters (Group and Parameter)

- The PARAM block contains parameter definitions. The following fields appear in the PARAM block:
  - GUID - Identifies the parameter definition.
  - NAME - Parameter definition name.
  - DATATYPE - Parameter type. This field can be a common type (TEXT, INTEGER, etc.), structural type (FORCE, MOMENT, etc.) or common family type (Area Tags, etc). Common type and structural type parameters are specified in the text file directly (e.g.: TEXT, FORCE). If the value of the DATATYPE field is FAMILYTYPE, an extra number is added. For example, FAMILYTYPE followed by -2005020 represents Family type: Area Tags.

Code Region 22-2: Shared Parameter FAMILYTYPE example

```
FAMILYTYPE    -2005020
```

Figure 130: New parameter definition

- GROUP - A group ID used to identify the group that includes the current parameter definition.
- VISIBLE - Identifies whether the parameter is visible. The value of this field is 0 or 1.

  0 = invisible

  1 = visible

As the definition file sample shows, there are two groups:

- MyGroup - ID 1 - Contains the parameter definition for MyParam which is a Text type parameter.
- AnotherGroup - ID 2 - Contains the parameter definition for Price which is an Integer type parameter.

22.2 Definition File Access

In the add-in code, complete the following steps to gain access to the definition file:

1. Specify the Autodesk.Revit.Options.Application.SharedParametersFilename property with an existing text file or a new one.
2. Open the shared parameters file, using the Application.OpenSharedParameterFile() method.
3. Open an existing group or create a new group using the DefinitionFile.Groups property.
4. Open an existing external parameter definition or create a new definition using the DefinitionGroup.Definitions property.

The following list provides more information about the classes and methods in the previous diagram.

- Autodesk.Revit.Parameters.DefinitionFile Class - The DefinitionFile object represents one shared parameter file.
  - The object contains a number of Group objects.
  - Shared parameters are grouped for easy management and contain shared parameter definitions.
  - You can add new definitions as needed.
  - The DefinitionFile object is retrieved using the Application.OpenSharedParameterFile method.
- Autodesk.Revit.Parameters.ExternalDefinition Class - The ExternalDefinition class is derived from the Definition class.

    - The ExternalDefinition object is created by a DefinitionGroup object from a shared parameter file.
    - External parameter definitions must belong to a Group which is a collection of shared parameter definitions.
- Autodesk.Revit.Options.Application.SharedParametersFilename Property - Get and set the shared parameter file path using the Autodesk.Revit.Options.SharedParametersFilename property.
    - By default, Revit does not have a shared parameter file.
    - Initialize this property before using. If it is not initialized, an exception is thrown.
- Autodesk.Revit.Application.OpenSharedParameterFile Method - This method returns an object representing a Revit shared parameter file.
    - Revit uses one shared parameter file at a time.
    - The file name for the shared parameter file is set in the Revit Application Options object. If the file does not exist, an exception is thrown.

22.2.1 Create a Shared Parameter File

Because the shared parameter file is a text file, you can create it using code or create it manually.

Code Region 22-3: Creating a shared parameter file

```
private void CreateExternalSharedParamFile(string sharedParameterFile)
{
   System.IO.FileStream fileStream = System.IO.File.Create(sharedParameterFile);
   fileStream.Close();
}
```

22.2.2 Access an Existing Shared Parameter File

Because you can have many shared parameter files for Revit, it is necessary to specifically identify the file and external parameters you want to access. The following two procedures illustrate how to access an existing shared parameter file.

22.2.2.1 Get DefinitionFile from an External Parameter File

Set the shared parameter file path to `app.Options.SharedParametersFilename` as the following code illustrates, then invoke the Autodesk.Revit.Application.OpenSharedParameterFile method.

Code Region 22-4: Getting the definition file from an external parameter file

```
private DefinitionFile SetAndOpenExternalSharedParamFile(
    Autodesk.Revit.ApplicationServices.Application application, string sharedParameterFile)
{
    // set the path of shared parameter file to current Revit
    application.Options.SharedParametersFilename = sharedParameterFile;
    // open the file
    return application.OpenSharedParameterFile();
}
```

Note: Consider the following points when you set the shared parameter path:

- During each installation, Revit cannot detect whether the shared parameter file was set in other versions. You must bind the shared parameter file for the new Revit installation again.
- If Options.SharedParametersFilename is set to a wrong path, an exception is thrown only when OpenSharedParameterFile is called.
- Revit can work with multiple shared parameter files. Even though only one parameter file is used when loading a parameter, the current file can be changed freely.

22.2.2.2 Traverse All Parameter Entries

The following sample illustrates how to traverse the parameter entries and display the results in a message box.

Code Region 22-5: Traversing parameter entries

```
private void ShowDefinitionFileInfo(DefinitionFile myDefinitionFile)
{
    StringBuilder fileInformation = new StringBuilder(500);

    // get the file name
    fileInformation.AppendLine("File Name: " + myDefinitionFile.Filename);

    // iterate the Definition groups of this file
    foreach (DefinitionGroup myGroup in myDefinitionFile.Groups)
    {
        // get the group name
        fileInformation.AppendLine("Group Name: " + myGroup.Name);

        // iterate the difinitions
        foreach (Definition definition in myGroup.Definitions)
        {
            // get definition name
            fileInformation.AppendLine("Definition Name: " + definition.Name);
        }
    }
    TaskDialog.Show("Revit", fileInformation.ToString());
}
```

22.2.3 Change the Parameter Definition Owner Group

The following sample shows how to change the parameter definition group owner.

Code Region 22-6: Changing parameter definition group owner

```
private void ReadEditExternalParam(DefinitionFile file)
{
    // get ExternalDefinition from shared parameter file
    DefinitionGroups myGroups = file.Groups;
    DefinitionGroup myGroup = myGroups.get_Item("MyGroup");
    if (null == myGroup)
        return;
```

```
    Definitions myDefinitions = myGroup.Definitions;
    ExternalDefinition myExtDef =
                            myDefinitions.get_Item("MyParam") as ExternalDefinition;
    if (null == myExtDef)
        return;

    StringBuilder strBuilder = new StringBuilder();
    // iterate every property of the ExternalDefinition
    strBuilder.AppendLine("GUID: " + myExtDef.GUID.ToString())
        .AppendLine("Name: " + myExtDef.Name)
        .AppendLine("OwnerGroup: " + myExtDef.OwnerGroup.Name)
        .AppendLine("Parameter Group" + myExtDef.ParameterGroup.ToString())
        .AppendLine("Parameter Type" + myExtDef.ParameterType.ToString())
        .AppendLine("Is Visible: " + myExtDef.Visible.ToString());

    TaskDialog.Show("Revit", strBuilder.ToString());

    // change the OwnerGroup of the ExternalDefinition
    myExtDef.OwnerGroup = myGroups.get_Item("AnotherGroup");
}
```

22.3 Binding

In the add-in code, complete the following steps to bind a specific parameter:

1. Use an InstanceBinding or a TypeBinding object to create a new Binding object that includes the categories to which the parameter is bound.
2. Add the binding and definition to the document using the Document.ParameterBindings object.

The following list provides more information about the classes and methods in the previous diagram.

- Autodesk.Revit.Parameters.BindingMap Class - The BindingMap object is retrieved from the Document.ParameterBindings property.
  - Parameter binding connects a parameter definition to elements within one or more categories.
  - The map is used to interrogate existing bindings as well as generate new parameter bindings using the Insert method.
- Parameters.BindingMap.Insert (Definition, Binding) Method - The binding object type dictates whether the parameter is bound to all instances or just types.
  - A parameter definition cannot be bound to both instances and types.
  - If the parameter binding exists, the method returns false.

22.3.1 Type Binding

The Autodesk.Revit.Parameters.TypeBinding objects are used to bind a property to a Revit type, such as a wall type. It differs from Instance bindings in that the property is shared by all instances identified in type binding. Changing the parameter for one type affects all instances of the same type.

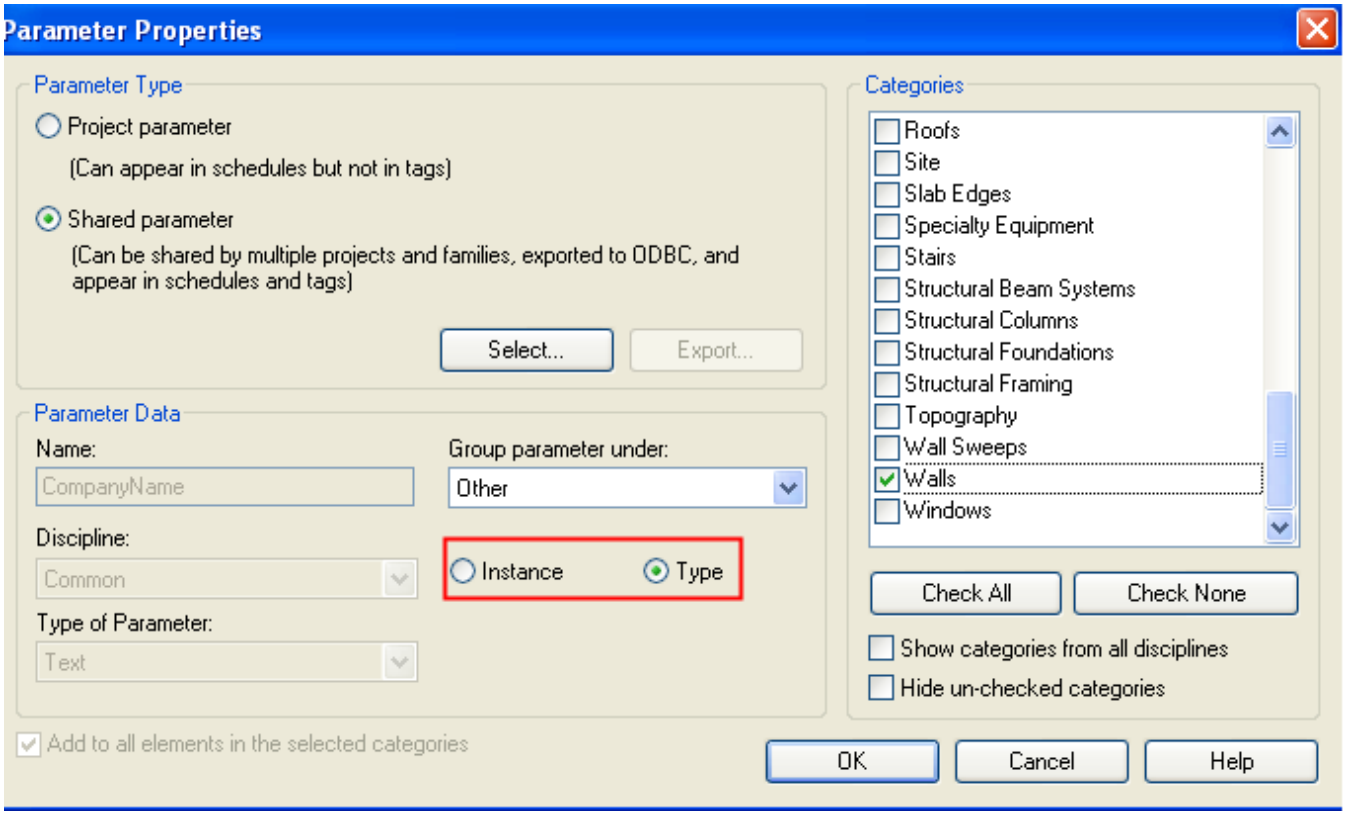


Figure 131: Parameter Properties dialog box Type Binding

The following code segment demonstrates how to add parameter definitions using a shared parameter file. The following code performs the same actions as using the dialog box in the previous picture. Parameter definitions are created in the following order:

1. A shared parameter file is created.
2. A definition group and a parameter definition are created for the Walls type.
3. The definition is bound to the wall type parameter in the current document based on the wall category.

Code Region 22-7: Adding type parameter definitions using a shared parameter file

```
public bool SetNewParameterToTypeWall(UIApplication app, DefinitionFile myDefinitionFile)
{
    // Create a new group in the shared parameters file
    DefinitionGroups myGroups = myDefinitionFile.Groups;
    DefinitionGroup myGroup = myGroups.Create("MyParameters");

    // Create a type definition
    Definition myDefinition_CompanyName =
           myGroup.Definitions.Create("CompanyName", ParameterType.Text);

    // Create a category set and insert category of wall to it
    CategorySet myCategories = app.Application.Create.NewCategorySet();
        // Use BuiltInCategory to get category of wall
    Category myCategory = app.ActiveUIDocument.Document.Settings.Categories.get_Item(
                                  BuiltInCategory.OST_Walls);
    myCategories.Insert(myCategory);
```

```
    //Create an object of TypeBinding according to the Categories
    TypeBinding typeBinding = app.Application.Create.NewTypeBinding(myCategories);

    // Get the BingdingMap of current document.
    BindingMap bindingMap = app.ActiveUIDocument.Document.ParameterBindings;

    // Bind the definitions to the document
    bool typeBindOK = bindingMap.Insert(myDefinition_CompanyName, typeBinding,
      BuiltInParameterGroup.PG_TEXT);
    return typeBindOK;
}
```

22.3.2 Instance Binding

The Autodesk.Revit.Parameters.InstanceBinding object indicates binding between a parameter definition and a parameter in certain category instances. The following diagram illustrates Instance Binding in the Walls category.

Once bound, the parameter appears in all property dialog boxes for the instance. Changing the parameter in any one instance does not change the value in any other instance.

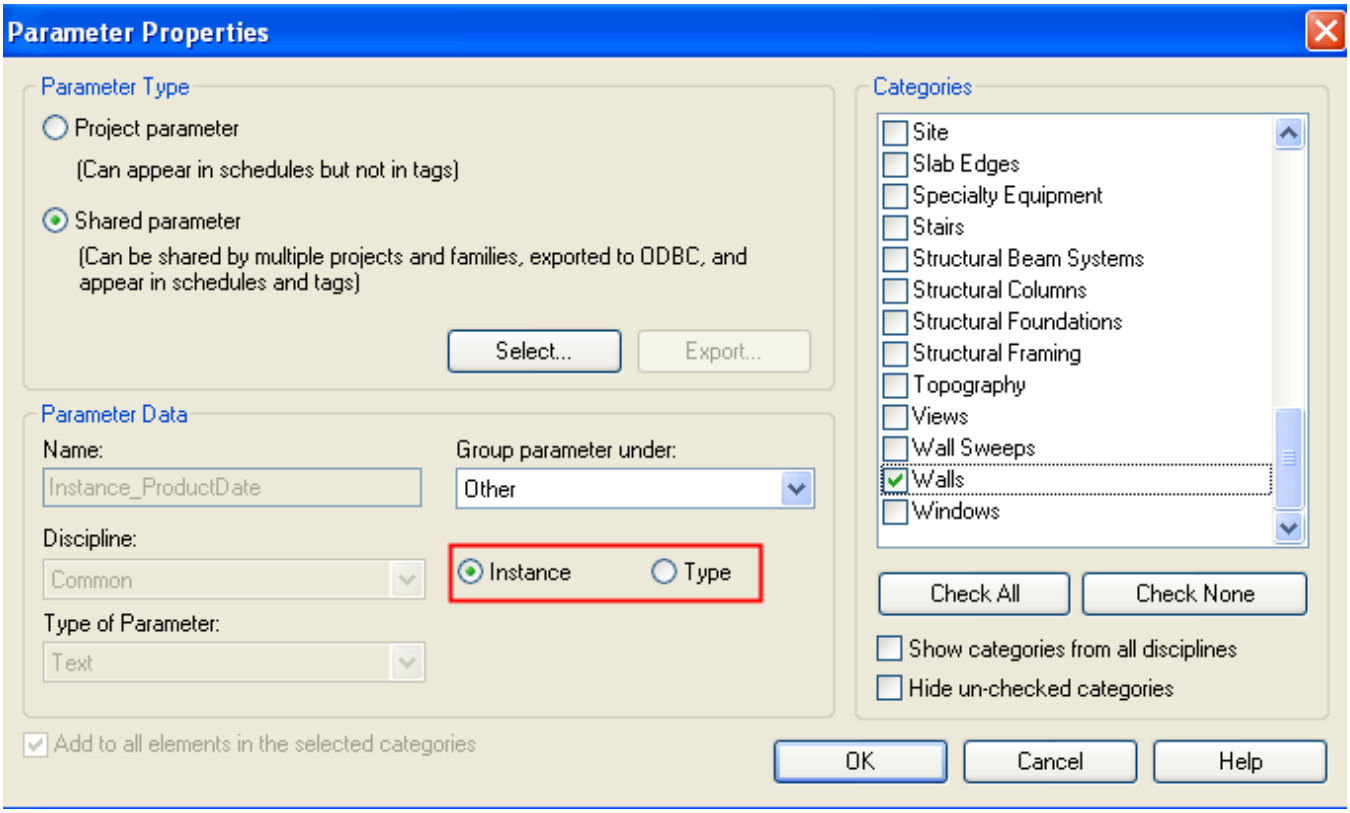


Figure 132: Parameter Properties dialog box Instance Binding

The following code sample demonstrates how to add parameter definitions using a shared parameter file. Parameter definitions are added in the following order:

1. A shared parameter file is created
2. A definition group and a definition for all Walls instances is created

3. Definitions are bound to each wall instance parameter in the current document based on the wall category.

Code Region 22-8: Adding instance parameter definitions using a shared parameter file

```
public bool SetNewParameterToInsanceWall(UIApplication app, DefinitionFile myDefinitionFile)
{
    // create a new group in the shared parameters file
    DefinitionGroups myGroups = myDefinitionFile.Groups;
    DefinitionGroup myGroup = myGroups.Create("MyParameters1");

    // create an instance definition in definition group MyParameters
    Definition myDefinition_ProductDate =
       myGroup.Definitions.Create("Instance_ProductDate", ParameterType.Text);

    // create a category set and insert category of wall to it
    CategorySet myCategories = app.Application.Create.NewCategorySet();
    // use BuiltInCategory to get category of wall
    Category myCategory = app.ActiveUIDocument.Document.Settings.Categories.get_Item(
                                BuiltInCategory.OST_Walls);
    myCategories.Insert(myCategory);

    //Create an instance of InstanceBinding
    InstanceBinding instanceBinding =
        app.Application.Create.NewInstanceBinding(myCategories);

    // Get the BingdingMap of current document.
    BindingMap bindingMap = app.ActiveUIDocument.Document.ParameterBindings;

    // Bind the definitions to the document
    bool instanceBindOK = bindingMap.Insert(myDefinition_ProductDate,
                                    instanceBinding, BuiltInParameterGroup.PG_TEXT);
    return instanceBindOK;
}
```

23 Transactions

Transactions are context-like objects that encapsulate any changes to a Revit model. Any change to a document can only be made while there is an active transaction open for that document. Attempting to change the document outside of a transaction will throw an exception. Changes do not become a part of the model until the active transaction is committed. Consequently, all changes made in a transaction can be rolled back either explicitly or implicitly (by the destructor). Only one transaction per document can be open at any given time. A transaction may consist of one or more operations.

There are three main classes in the Revit API related to transactions:

- Transaction
- SubTransaction
- TransactionGroup

This chapter will discuss each of these classes in more depth. Only the Transaction class is required to make changes to a document. The other classes can be used to better organize changes. Keep in mind that the TransactionMode attribute applied to the IExternalCommand definition will affect how Revit expects transactions to be handled when the command is invoked. Review the TransactionAttribute section for more information.

23.1 Transaction Classes

All three transaction objects share some common methods:

Method	Description
Start	Will start the context
Commit	Ends the context and commits all changes to the document
Rollback	Ends the context and discards all changes to the document
GetStatus	Returns the current status of the transaction object

Table 50: Common Transaction Object Methods

In addition to the GetStatus() method returning the current status, the Start, Commit and RollBack methods also return a TransactionStatus indicating whether or not the method was successful. Available TransactionStatus values include:

Status	Description
Uninitialized	The initial value after object is instantiated; the context has not started yet
Started	Transaction object has successfully started (Start was called)
RolledBack	Transaction object was successfully rolled back (Rollback was called)
Committed	Transaction object was successfully committed (Commit was called)
Pending	Transaction object was attempted to be either submitted or rolled back, but due to failures that process could not be finished yet and is waiting for the end-user's response (in a modeless dialog). Once the failure processing is finished, the status will be automatically updated (to either Committed or RolledBack status).

Table 51: TransactionStatus values

23.1.1 Transaction

A transaction is a context required in order to make any changes to a Revit model. Only one transaction can be open at a time; nesting is not allowed. Each transaction must have a name, which will be listed on the Undo menu in Revit once a transaction is successfully committed.

Code Region 23-1: Using transactions

```
// Get the document
Autodesk.Revit.DB.Document document = uiApplication.ActiveUIDocument.Document;

// Create a geometry line in Revit application
XYZ Point1 = new XYZ(0, 0, 0);
XYZ Point2 = new XYZ(10, 0, 0);
XYZ Point3 = new XYZ(10, 10, 0);
XYZ Point4 = new XYZ(0, 10, 0);

// Transient Elements,so they are not managed by transactions
Line geomLine1 = application.Create.NewLine(Point1, Point2, true);
Line geomLine2 = application.Create.NewLine(Point4, Point3, true);
Line geomLine3 = application.Create.NewLine(Point1, Point4, true);

// Create a geometry plane in Revit application
XYZ origin = new XYZ(0, 0, 0);
XYZ normal = new XYZ(1, 1, 0);
Plane geomPlane = application.Create.NewPlane(normal, origin);

// Create a sketch plane in current document
SketchPlane sketch = document.Create.NewSketchPlane(geomPlane);

// non-transient elements,so they are grouped into transaction as atomic user actions
Transaction transaction = new Transaction(document);
if (transaction.Start("Create new curves") == TransactionStatus.Started)
{
    // Create a sketch plane in current document
    SketchPlane sketch = document.Create.NewSketchPlane(geomPlane);

    // Create a ModelLine element using the created geometry line and sketch plane
    ModelLine line1 = document.Create.NewModelCurve(geomLine1, sketch) as ModelLine;
    ModelLine line2 = document.Create.NewModelCurve(geomLine2, sketch) as ModelLine;
    ModelLine line3 = document.Create.NewModelCurve(geomLine3, sketch) as ModelLine;

    TaskDialog taskDialog = new TaskDialog("Revit");
    taskDialog.MainContent =
        "Click OK to call Transaction.Commit(), otherwise Transaction.Rollback().";
    TaskDialogCommonButtons buttons = TaskDialogCommonButtons.Ok |
                                      TaskDialogCommonButtons.Cancel;
    taskDialog.CommonButtons = buttons;
    TaskDialogResult result = taskDialog.Show();
```

```
    if (TaskDialogResult.Ok == result)
    {
        transaction.Commit();
    }
    else
    {
         transaction.RollBack();
    }
}
```

23.1.2 **SubTransaction**

A SubTransaction can be used to enclose a set of model-modifying operations. Sub-transactions are optional. They are not required in order to modify the model. They are a convenience tool to allow logical splitting of larger tasks into smaller ones. Sub-transactions can only be created within an already opened transaction and must be closed (either committed or rolled back) before the transaction is closed (committed or rolled back). Unlike transactions, sub-transaction may be nested, but any nested sub-transaction must be closed before the enclosing sub-transaction is closed. Sub-transactions do not have a name, for they do not appear on the Undo menu in Revit.

23.1.3 **TransactionGroup**

TransactionGroup allows grouping together several independent transactions, which gives the owner of a group an opportunity to address many transactions at once. When a transaction group is to be closed, it can be rolled back, which means that all previously committed transactions belonging to the group will be rolled back. If not rolled back, a group can be either committed or assimilated. In the former case, all committed transactions (within the group) will be left as they were. In the later case, transactions within the group will be merged together into one single transaction that will bear the group's name.

A transaction group can only be started when there is no transaction open yet, and must be closed only after all enclosed transactions are closed (rolled back or committed). Transaction groups can be nested, but any nested group must be closed before the enclosing group is closed. Transaction groups are optional. They are not required in order to make modifications to a model.

The following example shows the use of a TransactionGroup to combine two separate Transactions using the Assimilate() method. The following code will result in a single Undo item added to the Undo menu with the name "Group Change".

Code Region 23-2: Combining multiple transactions into a TransactionGroup

```
public TransactionStatus TransactionGroupDemo(Autodesk.Revit.DB.Document document)
{
    TransactionStatus status = TransactionStatus.Uninitialized;
    bool bAssimilate = true;
    TransactionGroup transGroup = new TransactionGroup(document, "Group Change");
    if (transGroup.Start() == TransactionStatus.Started)
    {
        Transaction levelTransaction = new Transaction(document, "Add Level");
        try
        {
            if (levelTransaction.Start() == TransactionStatus.Started)
            {
```

```
                document.Create.NewLevel(25.0);
                levelTransaction.Commit();
            }
        }
        catch
        {
            // if either internal transaction fails, rollback group, too
            levelTransaction.RollBack();
            bAssimilate = false;
        }

        // if first transaction was successful, continue with next
        if (bAssimilate == true)
        {
            Transaction gridTransaction = new Transaction(document, "Add Grid");
            try
            {
                if (gridTransaction.Start() == TransactionStatus.Started)
                {
                    XYZ Point1 = new XYZ(0, 0, 0);
                    XYZ Point2 = new XYZ(10, 0, 0);

                    Line gridLine =
                        document.Application.Create.NewLine(Point1, Point2, true);
                    document.Create.NewGrid(gridLine);
                    gridTransaction.Commit();
                }
            }
            catch
            {
                // if either internal transaction fails, rollback group, too
                gridTransaction.RollBack();
                bAssimilate = false;
            }
        }

        // if both transactions are successful, assimilate the group
        if (bAssimilate == true)
        {
            status = transGroup.Assimilate();
        }
        else
        {
            status = transGroup.RollBack();
        }
    }

    return status;
```

```
}
```

23.2 Transactions in Events

23.2.1 **Modifying the document during an event**

Events do not automatically open transactions. Therefore, the document will not be modified during an event unless one of the event's handlers modifies it by making changes inside a transaction. If an event handler opens a transaction it is required that it will also close it (commit it or roll it back), otherwise all changes will be discarded.

Please be aware that modifying the active document is not permitted during some events (e.g. the DocumentClosing event). If an event handler attempts to make modifications during such an event, an exception will be thrown. The event documentation indicates whether or not the event is read-only.

23.2.2 **DocumentChanged Event**

The DocumentChanged event is raised after every transaction gets committed, undone, or redone. This is a read-only event, designed to allow you to keep external data in synch with the state of the Revit database. To update the Revit database in response to changes in elements, use the Dynamic Model Update framework.

23.3 Failure Handling Options

Failure handling options are options for how failures, if any, should be handled at the end of a transaction. Failure handling options may be set at any time before calling either Transaction.Commit() or Transaction.RollBack() using the Transaction.SetFailureHandlingOptions() method. However, after a transaction is committed or rolled back, the options return to their respective default settings.

The SetFailureHandlingOptions() method takes a FailureHandlingOptions object as a parameter. This object cannot be created, it must be obtained from the transaction using the GetFailureHandlingOptions() method. Options are set by calling the corresponding Set method, such as SetClearAfterRollback(). The following sections discuss the failure handling options in more detail.

23.3.1 **ClearAfterRollback**

This option controls whether all warnings should be cleared after a transaction is rolled back. The default value is False.

23.3.2 **DelayedMiniWarnings**

This options controls whether mini-warnings, if any, are displayed at the end of the transaction currently being ended, or if they should be postponed until the end of next transaction. This is typically used within a chain of transactions when it is not desirable to show intermediate warnings at the end of each step, but rather to wait until the completion of the entire chain.

Warnings may be delayed for more than one transaction. The first transaction that does not have this option set to True will display all of its own warnings, if any, as well as all warnings that might have accumulated from previous transactions. The default value is False.

Note that this option is ignored in modal mode (see ForcedModalHandling below).

23.3.3 ForcedModalHandling

This options controls whether eventual failures will be handled modally or modelessly. The default is True. Be aware that if the modeless failure handling is set, processing the transaction may be done asynchronously, which means that upon returning from the Commit or RollBack calls, the transaction will not be finished yet (the status will be 'Pending').

23.3.4 SetFailuresPreprocessor

This interface, if provided, is invoked when there are failures found at the end of a transaction. The preprocessor may examine current failures and even try to resolve them. See the chapter on Failure Posting and Handling for more information.

23.3.5 SetTransactionFinalizer

A finalizer is an interface, which, if provided, can be used to perform a custom action at the end of a transaction. Note that it is not invoked when the Commit() or RollBack() methods are called, but only after the process of committing or rolling back is completed. Transaction finalizers must implement the ITransactionFinalizer interface, which requires two functions to be defined:

- OnCommitted – called at the end of committing a transaction
- OnRolledBack – called at the end of rolling back a transaction

Note that since the finalizer is called after the transaction has finished, the document is not modifiable from the finalizer unless a new transaction is started.

23.4 Getting Element Geometry and AnalyticalModel

After new elements are created or elements are modified, regeneration and auto-joining of elements is required to propagate the changes throughout the model. Without a regeneration (and auto-join, when relevant), the Geometry property and the AnalyticalModel for Elements are either unobtainable (in the case of creating a new element) or they may be invalid. It is important to understand how and when regeneration occurs before accessing the Geometry or AnalyticalModel of an Element.

Although regeneration and auto-join are necessary to propagate changes made in the model, it can be time consuming. It is best if these events occur only as often as necessary. When using RegenerationOption.Automatic, regeneration and auto-joining happen automatically after each and every API call that modifies the model. In this case, the Geometry and AnalyticalModel are assigned immediately and no further action is required prior to accessing them. However, due to the slow performance of this option, it is obsolete and will be removed in a future version of Revit.

In RegenerationOption.Manual, regeneration and auto-joining occur automatically when a transaction that modifies the model is committed successfully, or whenever the Document.Regenerate() or Document.AutoJoinElements() methods are called. Regenerate() and AutoJoinElements() may only be called inside an open transaction. It should be noted that the Regeneration() method can fail, in which case the RegenerationFailedException will be thrown. If this happens, the changes to the document need to be rolled back by rolling back the current transaction or subtransaction.

For more details about the AnalyticalModel object, refer to the AnalyticalModel section in the Revit Structure chapter. For more details about the Geometry property, refer to the Geometry chapter.

The following sample program demonstrates how a transaction populates these properties when using RegenerationOption.Manual:

Code Region 23-3: Transaction populating Geometry and AnalyticalModel properties

```
public void TransactionDuringElementCreation(UIApplication uiApplication, Level level)
{
```

```
    Autodesk.Revit.DB.Document document = uiApplication.ActiveUIDocument.Document;

    // Build a location line for the wall creation
    XYZ start = new XYZ(0, 0, 0);
    XYZ end = new XYZ(10, 10, 0);
    Autodesk.Revit.DB.Line geomLine =
        uiApplication.Application.Create.NewLineBound(start, end);

    // To create a wall, a transaction must be first started
    Transaction wallTransaction = new Transaction(document, "Creating wall");

    if (wallTransaction.Start() == TransactionStatus.Started)
    {
        // Create a wall using the location line
        Wall wall = document.Create.NewWall(geomLine, level, true);

        // the transaction must be committed before you can
        // get the value of Geometry and AnalyticalModel.

        if (wallTransaction.Commit() == TransactionStatus.Committed)
        {
            Autodesk.Revit.DB.Options options =
                uiApplication.Application.Create.NewGeometryOptions();
            Autodesk.Revit.DB.GeometryElement geoelem = wall.get_Geometry(options);
            AnalyticalModel analyticalmodel = wall.GetAnalyticalModel();
        }
    }
}
```

The transaction timeline for this sample is as follows:

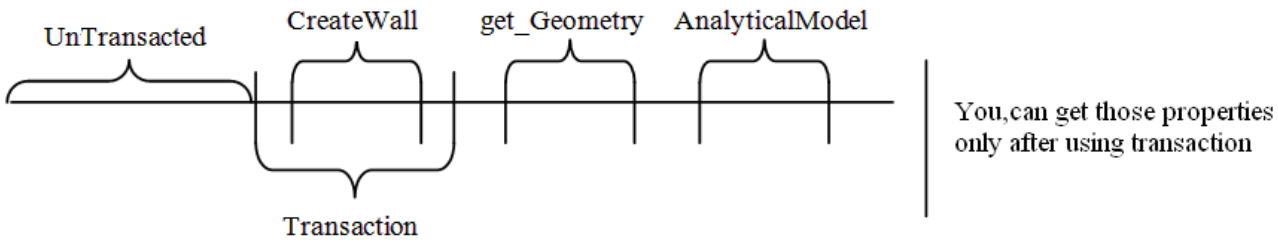


Figure 133: Transaction timeline

24 Events

Events are notifications that are triggered on specific actions in the Revit user interface or API workflows. By subscribing to events, an add-in application can be notified when an action is about to happen or has just happened and take some action related to that event. Some events come in pairs around actions, one occurring before the action takes place ("pre" event) and the other happening after the action takes place ("post" event). Events that do not occur in these pre/post pairs are called "single" events.

Revit provides access to events at both the Application level (such as ApplicationClosing or DocumentOpened) and the Document level (such as DocumentClosing and DocumentPrinting). The same application level events available from the Application class are also available from the ControlledApplication class, which represents the Revit application with no access to documents. It is ControlledApplication that is available to add-ins from the OnStartup() and OnShutdown() methods. In terms of subscribing and unsubscribing to events, these classes are interchangeable; subscribing to an event from the ControlledApplication class is the same as subscribing from the Application class.

Events can also be categorized as database (DB) events or user interface (UI) events. DB events are available from the Application and Document classes, while UI events are available from the UIApplication class. (Currently all UI events are at the application level only).

Some events are considered read-only, which means that during their execution the model may not be modified. The fact that an event is read-only is documented in the API help file. It is important to know that even during regular events (i.e. not read-only events), the model may be in a state in which it cannot be modified. The programmer should check the properties Document.IsModifiable and Document.IsReadOnly to determine whether the model may be modified.

24.1 Database Events

The following table lists database events, their type and whether they are available at the application and/or document level:

Event	Type	Application	Document
DocumentChanged	single	X	
DocumentClosing	pre	X	X
DocumentClosed	post	X	
DocumentCreating	pre	X	
DocumentCreated	post	X	
DocumentOpening	pre	X	
DocumentOpened	post	X	
DocumentPrinting	pre	X	X
DocumentPrinted	post	X	X
DocumentSaving	pre	X	X
DocumentSaved	post	X	X
DocumentSavingAs	pre	X	X
DocumentSavedAs	post	X	X
DocumentSynchronizingWithCentral	pre	X	

Event	Type	Application	Document
DocumentSynchronizedWithCentral	post	X	
FailuresProcessing	single	X	
FileExporting	pre	X	
FileExported	post	X	
FileImporting	pre	X	
FileImported	post	X	
ViewPrinting	pre	X	X
ViewPrinted	post	X	X

Table 52: DB Event Types

- DocumentChanged – notification when a transaction is committed, undone or redone
- DocumentClosing – notification when Revit is about to close a document
- DocumentClosed – notification just after Revit has closed a document
- DocumentCreating – notification when Revit is about to create a new document
- DocumentCreated – notification when Revit has finished creating a new document
- DocumentOpening – notification when Revit is about to open a document
- DocumentOpened – notification after Revit has opened a document
- DocumentPrinting – notification when Revit is about to print a view or ViewSet of the document
- DocumentPrinted – notification just after Revit has printed a view or ViewSet of the document
- DocumentSaving – notification when Revit is about to save the document
- DocumentSaved – notification just after Revit has saved the document
- DocumentSavingAs – notification when Revit is about to save the document with a new name
- DocumentSavedAs – notification when Revit has just saved the document with a new name
- DocumentSynchronizingWithCentral – notification when Revit is about to synchronize a document with the central file
- DocumentSynchronizedWithCentral – notification just after Revit has synchronized a document with the central file
- FailuresProcessing – notification when Revit is processing failures at the end of a transaction
- FileExporting – notification when Revit is about to export to a file format supported by the API
- FileExported – notification after Revit has exported to a file format supported by the API
- FileImporting – notification when Revit is about to import a file format supported by the API
- FileImported – notification after Revit has imported a file format supported by the API
- ViewPrinting – notification when Revit is about to print a view of the document
- ViewPrinted – notification just after Revit has printed a view of the document

24.2 User Inteface Events

The following table lists user interface events, their type and whether they are available at the application and/or document level:

Event	Type	UIApplication	UIDocument
ApplicationClosing	pre	X	
DialogBoxShowing	single	X	
Idling	single	X	
ViewActivating	pre	X	
ViewActivated	post	X	

Table 53: UI Event Types

- ApplicationClosing – notification when the Revit application is about to be closed
- DialogBoxShowing – notification when Revit is showing a dialog or message box
- Idling – notification when Revit is not in an active tool or transaction
- ViewActivating – notification when Revit is about to activate a view of the document
- ViewActivated – notification just after Revit has activated a view of the document

24.3 Registering Events

Using events is a two step process. First, you must have a function that will handle the event notification. This function must take two parameters, the first is an Object that denotes the "sender" of the event notification, the second is an event-specific object that contains event arguments specific to that event. For example, to register the DocumentSavingAs event, your event handler must take a second parameter that is a DocumentSavingAsEventArgs object.

The second part of using an event is registering the event with Revit. This can be done as early as in the OnStartup() function through the ControlledApplication parameter, or at any time after Revit starts up. Although events can be registered for External Commands as well as External Applications, it is not recommended unless the External Command registers and unregisters the event in the same external command.

The following example registers the DocumentOpened event, and when that event is triggered, this application will set the address of the project.

Code Region 24-1: Registering Application.DocumentOpened

```
public class Application_DocumentOpened : IExternalApplication
{
    public IExternalApplication.Result OnStartup(ControlledApplication application)
    {
        try
        {
            // Register event.
            application.DocumentOpened += new EventHandler
                 <Autodesk.Revit.Events.DocumentOpenedEventArgs>(application_DocumentOpened);
        }
        catch (Exception)
        {
```

```
            return IExternalApplication.Result.Failed;
        }

        return IExternalApplication.Result.Succeeded;
    }

    public IExternalApplication.Result OnShutdown(ControlledApplication application)
    {
        // remove the event.
        application.DocumentOpened -= application_DocumentOpened;
        return IExternalApplication.Result.Succeeded;
    }

    public void application_DocumentOpened(object sender, DocumentOpenedEventArgs args)
    {
        // get document from event args.
        Document doc = args.Document;

        Transaction transaction = new Transaction(doc, "Edit Address");
        if (transaction.Start() == TransactionStatus.Started)
        {
            doc.ProjectInformation.Address =
                "United States - Massachusetts - Waltham - 1560 Trapelo Road";
            transaction.Commit();
        }
    }

}
```

24.4 Canceling Events

Events that are triggered before an action has taken place (i.e. DocumentSaving) are often cancellable. (Use the Cancellable property to determine if the event can be cancelled.) For example, you may want to check some criteria are met in a model before it is saved. By registering for the DocumentSaving or DocumentSavingAs event, for example, you can check for certain criteria in the document and cancel the Save or Save As action. Once cancelled, an event cannot be un-cancelled.

Note that if a pre-event is cancelled, other event handlers that have subscribed to the event will not be notified. However, handlers that have subscribed to a post-event related to the pre-event will be notified.

The following event handler for the DocumentSavingAs event checks if the ProjectInformation Status parameter is empty, and if it is, cancels the SaveAs event. Note that if your application cancels an event, it should offer an explanation to the user.

Code Region 24-2: Canceling an Event

```
private void CheckProjectStatusInitial(Object sender, DocumentSavingAsEventArgs args)
{
    Document doc = args.Document;
    ProjectInfo proInfo = doc.ProjectInformation;
```

```
    // Project information is only available for project document.
    if (null != proInfo)
    {
        if (string.IsNullOrEmpty(proInfo.Status))
        {
            // cancel the save as process.
            args.Cancel = true;
            MessageBox.Show("Status project parameter is not set.  Save is aborted.");
        }
    }
}
```

Note that although most event arguments have the Cancel and Cancellable properties, the DocumentChanged and FailuresProcessing events have corresponding Cancel() and IsCancellable() methods.

25 Dynamic Model Update

Dynamic model update offers the ability for a Revit API application to modify the Revit model as a reaction to changes happening in the model when those changes are about to be committed at the end of a transaction. Revit API applications can create updaters by implementing the IUpdater interface and registering it with the UpdaterRegistry class. Registering includes specifying what changes in the model should trigger the updater.

25.1 Implementing IUpdater

The IUpdater interface requires that the following 5 methods to be implemented:

- GetUpdaterId() - This method should return a globally unique Id for the Updater consisting of the application Id plus a GUID for this Updater. This method is called once during registration of the Updater.
- GetUpdaterName() - This returns a name by which the Updater can be identified to the user, if there is a problem with the Updater at runtime.
- GetAdditionalInformation() - This method should return auxiliary text that Revit will use to inform the end user when the Updater is not loaded.
- GetChangePriority() - This method identifies the nature of the changes the Updater will be performing. It is used to identify the order of execution of updaters. This method is called once during registration of the Updater.
- Execute() - This is the method that Revit will invoke to perform an update. See the next section for more information on the Execute() method.

If a document is modified by an Updater, the document will store the unique Id of the updater. If the user later opens the document and the Updater is not present, Revit will warn the user that the 3rd party updater which previously edited the document is not available.

The following code is a simple example of implementing the IUpdater interface (to change the WallType for newly added walls) and registering the updater in the OnStartup() method. It demonstrates all the key aspects of creating and using an updater.

Code Region 25-1: Example of implementing IUpdater

```
[Autodesk.Revit.Attributes.Regeneration(Autodesk.Revit.Attributes.RegenerationOption.Manual)]
public class WallUpdaterApplication : Autodesk.Revit.UI.IExternalApplication
{
    public Result OnStartup(Autodesk.Revit.UI.UIControlledApplication application)
    {
        // Register wall updater with Revit
        WallUpdater updater = new WallUpdater(application.ActiveAddInId);
        UpdaterRegistry.RegisterUpdater(updater);

        // Change Scope = any Wall element
        ElementClassFilter wallFilter = new ElementClassFilter(typeof(Wall));

        // Change type = element addition
        UpdaterRegistry.AddTrigger(updater.GetUpdaterId(), wallFilter,
                                   Element.GetChangeTypeElementAddition());
        return Result.Succeeded;
    }
```

```
    public Result OnShutdown(Autodesk.Revit.UI.UIControlledApplication application)
    {
        WallUpdater updater = new WallUpdater(application.ActiveAddInId);
        UpdaterRegistry.UnregisterUpdater(updater.GetUpdaterId());
        return Result.Succeeded;
    }
}

public class WallUpdater : IUpdater
{
    static AddInId m_appId;
    static UpdaterId m_updaterId;
    WallType m_wallType = null;

    // constructor takes the AddInId for the add-in associated with this updater
    public WallUpdater(AddInId id)
    {
        m_appId = id;
        m_updaterId = new UpdaterId(m_appId, new Guid("FBFBF6B2-4C06-42d4-97C1-
D1B4EB593EFF"));
    }

    public void Execute(UpdaterData data)
    {
        Document doc = data.GetDocument();

        // Cache the wall type
        if (m_wallType == null)
        {
            FilteredElementCollector collector = new FilteredElementCollector(doc);
            collector.OfClass(typeof(WallType));
            var wallTypes = from element in collector
                            where
                                element.Name == "Exterior - Brick on CMU"
                            select element;
            if (wallTypes.Count<Element>() > 0)
            {
                m_wallType = wallTypes.Cast<WallType>().ElementAt<WallType>(0);
            }
        }

        if (m_wallType != null)
        {
            // Change the wall to the cached wall type.
            foreach (ElementId addedElemId in data.GetAddedElementIds())
            {
                Wall wall = doc.get_Element(addedElemId) as Wall;
                if (wall != null)
```

```
                {
                    wall.WallType = m_wallType;
                }
            }
        }
    }

    public string GetAdditionalInformation()
    {
        return "Wall type updater example: updates all newly created walls to a special
wall";
    }

    public ChangePriority GetChangePriority()
    {
        return ChangePriority.FloorsRoofsStructuralWalls;
    }

    public UpdaterId GetUpdaterId()
    {
        return m_updaterId;
    }

    public string GetUpdaterName()
    {
        return "Wall Type Updater";
    }
}
```

25.2 The Execute method

The purpose of the Execute() method is to allow your Updater to react to changes that have been made to the document, and make appropriate related. This method is invoked by Revit at the end of a document transaction in which elements that matched the UpdateTrigger for this Updater were added, changed or deleted. The method may be invoked more than once for the same transaction due to changes made by other Updaters. Updaters are invoked before the DocumentChanged event, so this event will contain changes made by all updaters.

All changes to the document made during the invocation of this method will become a part of the invoking transaction, and maintained for undo and redo operations. When implementing this method you may not open any new transactions (an exception will be thrown), but you may use sub-transactions as required.

Although it can be used to also update data outside of the document, such changes will not become part of the original transaction and will not be subject to undo or redo when the original transaction is undone or redone. If you do use this method to modify data outside of the document, you should also subscribe to the DocumentChanged event to update your data when the original transaction is undone or redone.

25.2.1 Scope of Changes

The Execute() method has an UpdaterData parameter that provides all necessary data needed to perform the update, including the document and information about the changes that triggered the update. Three basic methods (GetAddedElementIds(),GetDeletedElementIds(), and GetModifiedElementIds()) identify the elements that triggered the update. The Updater can also check specifically if a particular change triggered the update by using the IsChangeTriggered() method.

25.2.2 Forbidden and Cautionary Changes

The following methods may not be called while executing an Updater, because they introduce cross references between elements. (This can result in document corruption when these changes are combined with workset operations). A ForbiddenForDynamicUpdateException will be thrown when an updater attempts to call any of these methods:

- Autodesk.Revit.DB.ViewSheet.AddView()
- Autodesk.Revit.DB.Document.LoadFamily(Autodesk.Revit.DB.Document, Autodesk.Revit.DB.IFamilyLoadOptions)
- Autodesk.Revit.Creation.Document.NewAreaReinforcement()
- Autodesk.Revit.Creation.Document.NewPathReinforcement()

In addition to the forbidden methods listed above, other API methods that require documents to be in transaction-free state may not be called either. Such methods include but are not limited to Save(), SaveAs(), Close(), LoadFamily(), etc. Please refer to the documentation of the respective methods for more information.

Although the following methods are allowed during execution of an Updater, they can also throw ForbiddenForDynamicUpdateException when cross-references between elements are established as a result of the call. One such example could be creating a face wall that intersects with an existing face wall, so those two would have to be joined together. Apply caution when calling these methods from an Updater:

- Autodesk.Revit.Creation.ItemFactoryBase.NewFamilyInstances()
- Autodesk.Revit.Creation.ItemFactoryBase.NewFamilyInstance(Autodesk.Revit.DB.XYZ, Autodesk.Revit.DB.FamilySymbol, Autodesk.Revit.DB.Element,Autodesk.Revit.DB.Structure.StructuralType)
- Autodesk.Revit.Creation.Document.NewFamilyInstance(Autodesk.Revit.DB.XYZ, Autodesk.Revit.DB.FamilySymbol, Autodesk.Revit.DB.Element, Autodesk.Revit.DB.Level, Autodesk.Revit.DB.Structure.StructuralType)
- Autodesk.Revit.DB.FaceWall.Create()

It should also be noted that deleting and recreating existing elements should be avoided if modifying them would suffice. While deleting elements may be a simpler solution, it will not only affect Revit's performance, but it will also destroy any references to "recreated" objects from other elements. This could cause the user to lose work they have done to constrain and annotate the elements in question.

25.2.3 Managing Changes

Updaters need to be able to handle complex issues that may arise from their use, possibly reconciling subsequent changes to an element. Elements modified by an updater may change by the time the updater is next invoked, and those changes may impact information modified by the updater. For example, the element may be explicitly edited by the user, or implicitly edited due to propagated changes triggered by a regeneration.

It is also possible that the same element may be modified by another updater, possibly even within the same transaction. Although explicit changes of exactly the same data is tracked and prohibited, indirect or propagated changes are still possible. Perhaps the most complex case is that an element could be changed by the user and/or the same updater in different versions of the file. After the user reloads the latest or saves to central, the modified target element will be brought from the other file and the updater will need to reconcile changes.

It is also important to realize that when a document synchs with the central file, the ElementId of elements may be affected. If new elements have been added to two versions of the same file and the same ElementId is used in both places, this will be reconciled when the files are synched to the central database. For this reason, when using updaters to cross-reference one element in another element, they should use Element.UniqueId which is guaranteed to be unique.

Another issue to consider is if an updater attaches some data (i.e. as a parameter) to an element, it not only must be sure to maintain that information in the element to which it was added, but also to reconcile data in cases when that element is duplicated via copy/paste or group propagation. For example, if an updater adds a parameter "Total weight of rebar" to a rebar host, that parameter and its value will be copied to the duplicated rebar host even though the rebar itself may be not copied with the host. In this case the updater needs to ensure the parameter value is reset in the newly copied rebar host.

25.3 Registering Updaters

Updaters must be registered in order to be notified of changes to the model. The application level UpdaterRegistry class provides the ability to register/unregister and manipulate the options set for Updaters. Updaters may be registered from any API callback. However, in order to use the UpdaterRegistry functionality, the Revit add-in must be registered using manifest files (rather than the Revit.ini) and the Id returned by UpdaterId.GetAddInId() for any Updater (obtained from GetUpdaterId()) must match the AddInId field in the add-in's manifest file. An add-in cannot add, remove, or modify Updaters that do not belong to it.

25.3.1 Triggers

In addition to calling the UpdaterRegistry.RegisterUpdater() method, Updaters should add one or more update triggers via the AddTrigger() methods. These triggers indicate to the UpdaterRegistry what events should trigger the Updaters Execute() method to run. They can be set application-wide, or can apply to changes made in a specific document. Update triggers are specified by pairing a change scope and a change type.

The change scope is one of two things:

- An explicit list of element Ids in a document - only changes happening to those elements will trigger the Updater
- An implicit list of elements communicated via an ElementFilter - every changed element will be run against the filter, and if any pass, the Updater is triggered

There are several options available for change type. ChangeTypes are obtained from static methods on the Element class.

- Element addition - via Element.GetChangeTypeElementAddition()
- Element deletion - via Element.GetChangeTypeElementDeletion()
- Change of element geometry (shape or position) - via Element.GetChangeTypeGeometry()
- Changing value of a specific parameter - via Element.GetChangeTypeParameter()
- Any change of element - via Element.GetChangeTypeAny().

Note that geometry changes are triggered due to potentially many causes, like a change of element type, modification of properties and parameters, move and rotate, or changes imposed on the element from other modified elements during regeneration.

Also note that the last option, any change of element, only triggers the Updater for modifications of pre-existing elements, and does not trigger the Updater for newly added or deleted elements. Additionally, when using this trigger for an instance, only certain modifications to its type will trigger the Updater. Changes that affect the instance itself, such as modification of the instance's geometry, will trigger the Updater. However, changes that do not modify the instance directly and do not result in any discernable change to the instance, such as changes to text parameters, will not trigger the Updater for the instance. To trigger based on these changes, the Type must also be included in the trigger's change scope.

25.3.2 Order of Execution

The primary way that Revit sorts multiple Updaters to execute in the correct order is by looking at the ChangePriority returned by a given Updater. An Updater reporting a priority for a more fundamental set of elements (e.g. GridsLevelsReferencePlanes) will execute prior to Updaters reporting a priority for elements driven by these fundamental elements (e.g. Annotations). Reporting a proper change priority for the elements which your Updater modifies benefits users of your application: Revit is less likely to have to execute the Updater a second time due to changes made by another Updater.

For Updaters which report the same change priority, execution is ordered based on a sorting of UpdaterId. The method UpdaterRegistry.SetExecutionOrder() allows you set the execution order between any two registered Updaters (even updaters registered by other API add-ins) so long as your code knows the ids of the two Updaters.

25.4 Exposure to End-User

When updaters work as they should, they are transparent to the user. In some special cases though, Revit will display a warning to the user concerning a 3rd party updater. Such messages will use the value of the GetUpdaterName() method to refer to the updater.

25.4.1 Updater not installed

If a document is modified by an updater and later loaded when that updater is not installed, a task dialog similar to the following is displayed:

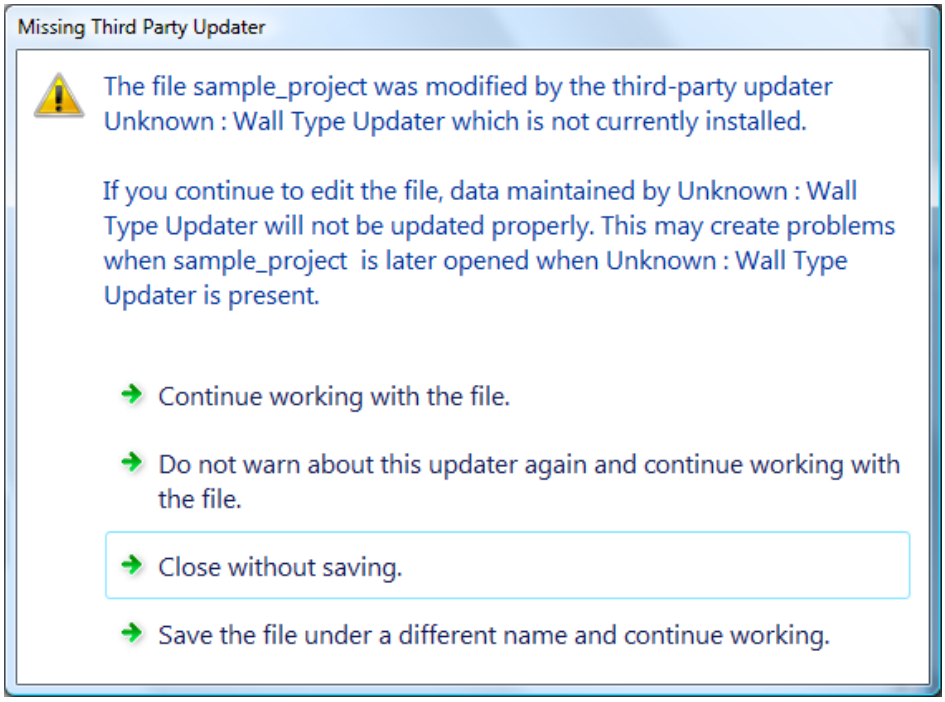


Figure 134: Missing Third Party Updater Warning

25.4.2 Updater performs invalid operation

If an updater has an error, such as an unhandled exception, a message similar to the following is displayed giving the user the option to disable the updater:

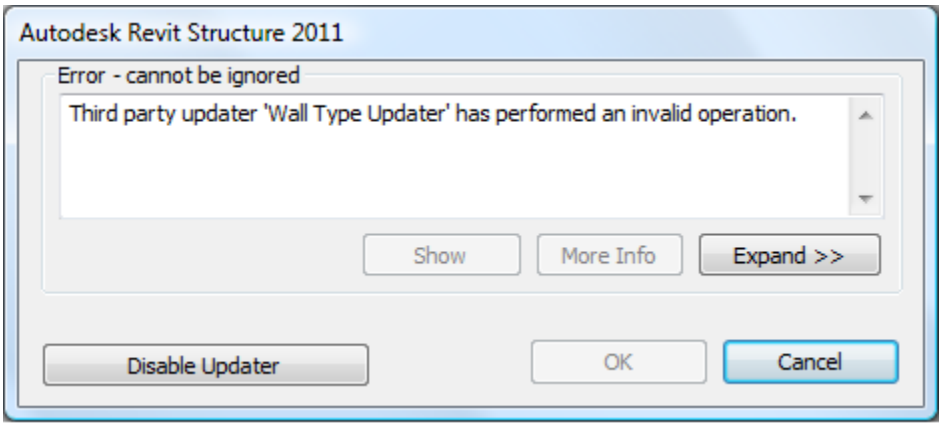


Figure 135: Updater performed an invalid operation

If the user selects Cancel, the entire transaction is rolled back. In the Wall Updater example from earlier in this chapter, the newly added wall is removed. If the user selects Disable Updater, the updater is no longer called, but the transaction is not rolled back.

25.4.3 Infinite loop

In the event that an updater falls into an infinite loop, Revit will notify the user and disable the updater for the duration of the Revit session.

25.4.4 Two updaters attempt to edit same element

If an updater attempts to edit the same parameter of an element that was updated by another updater in the same transaction, or if an updater attempts to edit the geometry of an element in a

way that conflicts with a change made by another updater, the updater is canceled, an error message is displayed and the user is given the option to disable the updater.

25.4.5 Central document modified by updater not present locally

If the user reloads latest or saves to central with a central file that was modified by an updater that is not installed locally, a task dialog is presented giving them the option to continue or cancel the synchronization. The warning indicates that proceeding may cause problems with the central model when it is used with the third party updater at a later time.

26 Failure Posting and Handling

The Revit API provides the ability to post failures when a user-visible problem has occurred and to respond to failures posted by Revit or Revit add-ins.

26.1 Posting Failures

To use the failure posting mechanism to report problems, the following steps are required:

1. New failures not already defined in Revit must be defined and registered in the FailureDefinitionRegistry during the OnStartup() call of the ExternalApplication.
2. Find the failure definition id, either from the BuiltInFailures classes or from the pre-registered custom failures using the class related to FailureDefinition.
3. Post the failure to the document that has a problem using the classes related to FailureMessage to set options and details related to the failure.

26.1.1 Defining and registering a failure

Each possible failure in Revit must be defined and registered during Revit application startup by creating a FailureDefinition object that contains some persistent information about the failure such as identity, severity, basic description, types of resolution and default resolution.

The following example creates two new failures, a warning and an error, that can be used for walls that are too tall. In this example, they are used in conjunction with an Updater that will do the failure posting (in a subsequent code sample in this chapter). The FailureDefinitionIds are saved in the Updater class since they will be required when posting failures. The sections following explain the FailureDefinition.CreateFailureDefinition() method parameters in more detail.

Code Region 26-1: Defining and registering a failure

```
WallWarnUpdater wallUpdater = new WallWarnUpdater();
UpdaterRegistry.RegisterUpdater(wallUpdater);
ElementClassFilter filter = new ElementClassFilter(typeof(Wall));
UpdaterRegistry.AddTrigger(wallUpdater.GetUpdaterId(),
                           filter,
                           Element.GetChangeTypeGeometry());

// define a new failure id for a warning about walls
FailureDefinitionId warnId =
                new FailureDefinitionId(new Guid("FB4F5AF3-42BB-4371-B559-FB1648D5B4D1"));

// register the new warning using FailureDefinition
FailureDefinition failDef = FailureDefinition.CreateFailureDefinition(warnId,
                FailureSeverity.Warning,
                "Wall is too big (>100'). Performance problems may result.");

FailureDefinitionId failId =
                new FailureDefinitionId(new Guid("691E5825-93DC-4f5c-9290-8072A4B631BC"));

FailureDefinition failDefError = FailureDefinition.CreateFailureDefinition(failId,
                FailureSeverity.Error,
                "Wall is WAY too big (>200'). Performance problems may result.");
```

```
// save ids for later reference
wallUpdater.WarnId = warnId;
wallUpdater.FailureId = failId;
```

26.1.1.1 FailureDefinitionId

A unique FailureDefinitionId must be used as a key to register the FailureDefinition. Each unique FailureDefinitionId should be created using a GUID generation tool. Later, the FailureDefinitionId can be used to look up a FailureDefinition in FailureDefinitionRegistry, and to create and post FailureMessages.

26.1.1.2 Severity

When registering a new failure, a severity is specified, along with the FailureDefinitionId and a text description of the failure that can be displayed to the user. The severity determines what actions are allowed in a document and whether the transaction can be committed at all. The severity options are:

- **Warning** - Failure that can be ignored by end-user. Failures of this severity do not prevent transactions from being committed. This severity should be used when Revit needs to communicate a problem to the user, but the problem does not prevent the user from continuing to work on the document
- **Error** - Failure that cannot be ignored. If FailureMeassage of this severity is posted, the current transaction cannot be committed unless the failure is resolved via an appropriate FailureResolution. This severity should be used when work on the document cannot be continued unless the problem is resolved. If the failure has no predefined resolutions available or these resolutions fail to resolve the problem, the transaction must be aborted in order to continue working with the document. It is strongly encouraged to have at least one resolution in each failure of this severity.
- **DocumentCorruption** - Failure that forces the Transaction to be rolled back as soon as possible due to known corruption to a document. When failure of this severity is posted, reading of information from a document is not allowed. The current transaction must be rolled back first in order to work with the document. This severity is used only if there is known data corruption in the document. This type of failure should generally be avoided unless there is no way to prevent corruption or to recover from it locally.

A fourth severity, None, cannot be specified when defining a new FailureDefinition.

26.1.1.3 Failure Resolutions

When a failure can be resolved, all possible resolutions should be predefined in the FailureDefinition class. This informs Revit what failure resolutions can possibly be used with a given failure. The FailureDefinition contains a full list of resolution types applicable to the failure, including a user-visible caption of the resolution.

The number of resolutions is not limited, however as of the 2011 Revit API, the only exposed failure resolution is DeleteElements. When more than one resolution is specified, unless explicitly changed using the SetDefaultResolutionType() method, the first resolution added becomes the default resolution. The default resolution is used by the Revit failure processing mechanism to resolve failures automatically when applicable. The Revit UI only uses the default resolution, but Revit add-ins, via the Revit API, can use any applicable resolution, and can provide an alternative UI for doing that (as described in the Handling Failures section later in this chapter).

In the case of a failure with a severity of DocumentCorruption, by the time failure resolution could occur, the transaction is already aborted, so there is nothing to resolve. Therefore, FailureResolutions should not be added to API-defined Failures of severity DocumentCorruption.

26.1.2 Posting a failure

The Document.PostFailure() method is used to notify the document of a problem. Failures will be validated and possibly resolved at the end of the transaction. Warnings posted via this method will not be stored in the document after they are resolved. Failure posting is used to address a state of the document which may change before the end of the transaction or when it makes sense to defer resolution until the end of the transaction. Not all failures encountered by an external command should post a failure. If the failure is unrelated to the document, a task dialog should be used. For example, if the Revit UI is in an invalid state to perform the external command.

To post a failure, create a new FailureMessage using the FailureDefinitionId from when the custom failure was defined, or use a BuiltInFailure provided by the Revit API. Set any additional information in the FailureMessage object, such as failing elements, and then call Document.PostFailure() passing in the new FailureMessage. Note that the document must be modifiable in order to post a failure.

A unique FailureMessageKey returned by PostFailure() can be stored for the lifetime of transaction and used to remove a failure message if it is no longer relevant. If the same FailureMessage is posted two or more times, the same FailureMessageKey is returned. If a posted failure has a severity of DocumentCorruption, an invalid FailureMessageKey is returned. This is because a DocumentCorruption failure cannot be unposted.

The following example shows an IUpdate class (referenced in the "Defining and registering a failure" code region above) that posts a new failure based on information received in the Execute() method.

Code Region 26-2: Posting a failure

```
public class WallWarnUpdater : IUpdater
{
    static AddInId m_appId;
    UpdaterId m_updaterId;
    FailureDefinitionId m_failureId = null;
    FailureDefinitionId m_warnId = null;

    // constructor takes the AddInId for the add-in associated with this updater
    public WallWarnUpdater(AddInId id)
    {
        m_appId = id;
        m_updaterId = new UpdaterId(m_appId,
            new Guid("69797663-7BCB-44f9-B756-E4189FE0DED8"));
    }

    public void Execute(UpdaterData data)
    {
        Document doc = data.GetDocument();
        Autodesk.Revit.ApplicationServices.Application app = doc.Application;
        foreach (ElementId id in data.GetModifiedElementIds())
        {
            Wall wall = doc.get_Element(id) as Wall;
            Autodesk.Revit.DB.Parameter p = wall.get_Parameter("Unconnected Height");
            if (p != null)
            {
                if (p.AsDouble() > 200)
```

```
                {
                    FailureMessage failMessage = new FailureMessage(FailureId);
                    failMessage.SetFailingElement(id);
                    doc.PostFailure(failMessage);
                }
                else if (p.AsDouble() > 100)
                {
                    FailureMessage failMessage = new FailureMessage(WarnId);
                    failMessage.SetFailingElement(id);
                    doc.PostFailure(failMessage);
                }
            }
        }
    }

    public FailureDefinitionId FailureId
    {
        get { return m_failureId; }
        set { m_failureId = value; }
    }

    public FailureDefinitionId WarnId
    {
        get { return m_warnId; }
        set { m_warnId = value; }
    }

    public string GetAdditionalInformation()
    {
        return "Give warning and error if wall is too tall";
    }

    public ChangePriority GetChangePriority()
    {
        return ChangePriority.FloorsRoofsStructuralWalls;
    }

    public UpdaterId GetUpdaterId()
    {
        return m_updaterId;
    }

    public string GetUpdaterName()
    {
        return "Wall Height Check";
    }
}
```

26.1.3 Removal of posted failures

Because there may be multiple changes to a document and multiple regenerations in the same transaction, it is possible that some failures are no longer relevant and they may need to be removed to prevent "false alarms". Specific messages can be un-posted by calling the Document.UnpostFailure() method and passing in the FailureMessageKey obtained when PostFailure() was called. UnpostFailure() will throw an exception if the severity of the failure is DocumentCorruption.

It is also possible to automatically remove all posted failures when a transaction is about to be rolled back (so that the user is not bothered to hit Cancel) by using the Transaction.SetFailureHandlingOptions() method.

26.2 Handling Failures

Normally posted failures are processed by Revit's standard failure resolution UI at the end of a transaction (specifically when Transaction.Commit() or Transaction.Rollback() are invoked). The user is presented information and options to deal with the failures.

If an operation (or set of operations) on the document requires some special treatment from a Revit add-in for certain errors, failure handling can be customized to carry out this resolution. Custom failure handling can be supplied:

- For a given transaction using the interface IFailuresPreprocessor.
- For all possible errors using the FailuresProcessing event.

Finally, the API offers the ability to completely replace the standard failure processing user interface using the interface IFailuresProcessor. Although the first two methods for handling failures should be sufficient in most cases, this last option can be used in special cases, such as to provide a better failure processing UI or when an application is used as a front-end on top of Revit.

26.2.1 Overview of Failure Processing

It is important to remember there are many things happening between the call to Transaction.Commit() and the actual processing of failures. Auto-join, overlap checks, group checks and workset editability checks are just to name a few. These checks and changes may make some failures disappear or, more likely, can post new failures. Therefore, conclusions cannot be drawn about the state of failures to be processed when Transaction.Commit() is called. To process failures correctly, it is necessary to hook up to the actual failures processing mechanism.

When failures processing begins, all changes to a document that are supposed to be made in the transaction are made, and all failures are posted. Therefore, no uncontrolled changes to a document are allowed during failures processing. There is a limited ability to resolve failures via the restricted interface provided by FailuresAccessor. If this has happened, all end of transaction checks and failures processing have to be repeated. So there may be a few failure resolution cycles at the end of one transaction.

Each cycle of failures processing includes 3 steps:

1. Preprocessing of failures (FailuresPreprocessor)
2. Broadcasting of failures processing event (FailuresProcessing event)
3. Final processing (FailuresProcessor)

Each of these 3 steps can control what happens next by returning different FailureProcessingResults. The options are:

- **Continue** - has no impact on execution flow. If FailuresProcessor returns "Continue" with unresolved failures, Revit will instead act as if "ProceedWithRollBack" was returned.

- **ProceedWithCommit** - interrupts failures processing and immediately triggers another loop of end-of-transaction checks followed by another failures processing. Should be returned after an attempt to resolve failures. Can easily lead to an infinite loop if returned without any successful failure resolution. Cannot be returned if transaction is already being rolled back and will be treated as "ProceedWithRollBack" in this case.
- **ProceedWithRollback** - continues execution of failure processing, but forces transaction to be rolled back, even if it was originally requested to commit. If before ProceedWithRollBack is returned FailureHandlingOptions are set to clear errors after rollback, no further error processing will take place, all failures will be deleted and transaction is rolled back silently. Otherwise default failure processing will continue, failures may be delivered to the user, but transaction is guaranteed to be rolled back.
- **WaitForUserInput** - Can be returned only by FailuresProcessor if it is waiting for an external event (typically user input) to complete failures processing.

Depending on the severity of failures posted in the transaction and whether the transaction is being committed or rolled back, each of these 3 steps may have certain options to resolve errors. All information about failures posted in a document, information about ability to perform certain operations to resolve failures and API to perform such operations are provided via the FailuresAccessor class. The Document can be used to obtain additional information, but it cannot be changed other than via FailuresAccessor.

26.2.2 FailuresAccessor

A FailuresAccessor object is passed to each of failure processing steps as an argument and is the only available interface to fetch information about failures in a document. While reading from a document during failure processing is allowed, the only way to modify a document during failure resolution is via methods provided by this class. After returning from failure processing, the instance of the class is deactivated and cannot be used any longer.

26.2.2.1 Information Available from FailuresAccessor

The FailuresAccessor object offers some generic information such as:

- Document for which failures are being processed or preprocessed
- Highest severity of failures posted in the document
- Transaction name and failure handling options for transaction being finished
- Whether transaction was requested to be committed or rolled back.

The FailuresAccessor object also offers information about specific failures via the GetFailuresMessages() method.

26.2.2.2 Options to resolve failures

The FailuresAccessor object provides a few ways to resolve failures:

- Failure messages with a severity of Warning can be deleted with the DeleteWarning() or DeleteAllWarnings() methods.
- ResolveFailure() or ResolveFailures() methods can be used to resolve one or more failures using the last failure resolution type set for each failure.
- DeleteElements() can resolve failures by deleting elements related to the failure.
- Or delete all failure messages and replace them with one “generic” failure using the ReplaceFailures() method.

26.2.3 IFailuresPreprocessor

The IFailuresPreprocessor interface can be used to provide custom failure handling for a specific transaction only. IFailuresPreprocessor is an interface that may be used to perform a preprocessing step to either filter out anticipated transaction failures or to post new failures. Failures can be "filtered out" by:

- silently removing warnings that are known to be posted for the transaction and are deemed as irrelevant for the user in the context of a particular transaction
- silently resolving certain failures that are known to be posted for the transaction and that should always be resolved in a context of a given transaction
- silently aborting the transaction in cases where "imperfect" transactions should not be committed or aborting the transaction is preferable over user interaction for a given workflow.

The IFailuresPreprocessor interface gets control first during the failure resolution process. It is nearly equivalent to checking and resolving failures before finishing a transaction, except that IFailuresPreprocessor gets control at the right time, after all failures guaranteed to be posted and/or after all irrelevant ones are deleted.

There may be only one IFailuresPreprocessor per transaction and there is no default failure preprocessor. If one is not attached to the transaction (via the failure handling options), this first step of failure resolution is simply omitted.

Code Region 26-3: Handling failures from IFailuresPreprocessor

```
public class SwallowTransactionWarning : IExternalCommand
{
    public Autodesk.Revit.UI.Result Execute(ExternalCommandData commandData,
        ref string message, ElementSet elements)
    {
        Autodesk.Revit.ApplicationServices.Application app =
                commandData.Application.Application;
        Document doc = commandData.Application.ActiveUIDocument.Document;
        UIDocument uidoc = commandData.Application.ActiveUIDocument;

        FilteredElementCollector collector = new FilteredElementCollector(doc);
        ICollection<Element> elementCollection =
                collector.OfClass(typeof(Level)).ToElements();
        Level level = elementCollection.Cast<Level>().ElementAt<Level>(0);

        Transaction t = new Transaction(doc);
        t.Start("room");
        FailureHandlingOptions failOpt = t.GetFailureHandlingOptions();
        failOpt.SetFailuresPreprocessor(new RoomWarningSwallower());
        t.SetFailureHandlingOptions(failOpt);

        doc.Create.NewRoom(level, new UV(0, 0));
        t.Commit();

        return Autodesk.Revit.UI.Result.Succeeded;
    }
}
```

```
public class RoomWarningSwallower : IFailuresPreprocessor
{
    public FailureProcessingResult PreprocessFailures(FailuresAccessor failuresAccessor)
    {
        IList<FailureMessageAccessor> failList = new List<FailureMessageAccessor>();
        // Inside event handler, get all warnings
        failList = failuresAccessor.GetFailureMessages();
        foreach (FailureMessageAccessor failure in failList)
        {
            // check FailureDefinitionIds against ones that you want to dismiss,
            FailureDefinitionId failID = failure.GetFailureDefinitionId();
            // prevent Revit from showing Unenclosed room warnings
            if (failID == BuiltInFailures.RoomFailures.RoomNotEnclosed)
            {
                failuresAccessor.DeleteWarning(failure);
            }
        }

        return FailureProcessingResult.Continue;
    }
}
```

26.2.4 **FailuresProcessing Event**

The FailuresProcessing event is most suitable for applications that want to provide custom failure handling without a user interface, either for the entire session or for many unrelated transactions. Some use cases for handling failures via this event are:

- automatic removal of certain warnings and/or automatic resolving of certain errors based on office standards (or other criteria)
- custom logging of failures

The FailuresProcessing event is raised after IFailuresPreprocessor (if any) has finished. It can have any number of handlers, and all of them will be invoked. Since event handlers have no way to return a value, the SetProcessingResult() on the event argument should be used to communicate status. Only Continue, ProceedWithRollback or ProceedWithCommit can be set.

The following example shows an event handler for the FailuresProcessing event.

Code Region 26-4: Handling the FailuresProcessing Event

```
private void CheckWarnings(object sender, FailuresProcessingEventArgs e)
{
    FailuresAccessor fa = e.GetFailuresAccessor();
    IList<FailureMessageAccessor> failList = new List<FailureMessageAccessor>();
    failList = fa.GetFailureMessages(); // Inside event handler, get all warnings
    foreach (FailureMessageAccessor failure in failList)
    {
        // check FailureDefinitionIds against ones that you want to dismiss,
        FailureDefinitionId failID = failure.GetFailureDefinitionId();
        // prevent Revit from showing Unenclosed room warnings
```

```
        if (failID == BuiltInFailures.RoomFailures.RoomNotEnclosed)
        {
            fa.DeleteWarning(failure);
        }
    }
}
```

26.2.5 **FailuresProcessor**

The IFailuresProcessor interface gets control last, after the FailuresProcessing event is processed. There is only one active IFailuresProcessor in a Revit session. To register a failures processor, derive a class from IFailuresProcessor and register it using the Application.RegisterFailuresProcessor() method. If there is previously registered failures processor, it is discarded. If a Revit add-in opts to register a failures processor for Revit that processor will become the default error handler for all Revit errors for the session and the standard Revit error dialog will not appear. If no failures processors are set, there is a default one in the Revit UI that invokes all regular Revit error dialogs. FailuresProcessor should only be overridden to replace the existing Revit failure UI with a custom failure resolution handler, which can be interactive or have no user interface.

If the RegisterFailuresProcessor() method is passed NULL, any transaction that has any failures is silently aborted (unless failures are resolved by first two steps of failures processing).

The IFailuresProcessor.ProcessFailures() method is allowed to return WaitForUserInput, which leaves the transaction pending. It is expected that in this case, FailuresProcessor leaves some UI on the screen that will eventually commit or rollback a pending transaction - otherwise the pending state will last indefinitely, essentially freezing the document.

The following example of implementing the IFailuresProcessor checks for a failure, deletes the failing elements and sets an appropriate message for the user.

Code Region 26-5: IFailuresProcessor

```
[Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.Automatic)]
[Autodesk.Revit.Attributes.Regeneration(Autodesk.Revit.Attributes.RegenerationOption.Manual)]
public class MyFailuresUI : IExternalApplication
{
    static AddInId m_appId = new AddInId(new Guid("9F179363-B349-4541-823F-A2DDB2B86AF3"));
    public Autodesk.Revit.UI.Result OnStartup(UIControlledApplication
uiControlledApplication)
    {
        IFailuresProcessor myFailUI = new FailUI();
        Autodesk.Revit.ApplicationServices.Application.RegisterFailuresProcessor(myFailUI);
        return Result.Succeeded;
    }

    public Autodesk.Revit.UI.Result OnShutdown(UIControlledApplication application)
    {
        return Result.Succeeded;
    }

    public class FailUI : IFailuresProcessor
    {
        public void Dismiss(Document document)
```

```
        {
            // This method is being called in case of exception or document destruction to
            // dismiss any possible pending failure UI that may have left on the screen
        }

        public FailureProcessingResult ProcessFailures(FailuresAccessor failuresAccessor)
        {
            IList<FailureResolutionType> resolutionTypeList =
                new List<FailureResolutionType>();
            IList<FailureMessageAccessor> failList = new List<FailureMessageAccessor>();
            // Inside event handler, get all warnings
            failList = failuresAccessor.GetFailureMessages();
            string errorString = "";
            bool hasFailures = false;
            foreach (FailureMessageAccessor failure in failList)
            {
                // check how many resolutions types were attempted to try to prevent
                // entering infinite loop
                resolutionTypeList =
                   failuresAccessor.GetAttemptedResolutionTypes(failure);
                if (resolutionTypeList.Count >= 3)
                {
                    TaskDialog.Show("Error",
                        "Cannot resolve failures - transaction will be rolled back.");
                    return FailureProcessingResult.ProceedWithRollBack;
                }

                errorString += "IDs ";
                foreach (ElementId id in failure.GetFailingElementIds())
                {
                    errorString += id + ", ";
                    hasFailures = true;
                }
                errorString += "\nWill be deleted because: " +
                               failure.GetDescriptionText() + "\n";
                failuresAccessor.DeleteElements(
                             failure.GetFailingElementIds() as IList<ElementId>);
            }
            if (hasFailures)
            {
                TaskDialog.Show("Error", errorString);
                return FailureProcessingResult.ProceedWithCommit;
            }

            return FailureProcessingResult.Continue;
        }
    }
}
```

27 Analysis Visualization

The Revit API provides a mechanism for external analysis applications to easily display the results of their computation in the Revit model. The SpatialFieldManager class is the main class for communicating analysis results back to Revit. It is used to create, delete, and modify the "containers" in which the analysis results are stored. The AnalysisDisplayStyle class can then be used to control the appearance of the results. Creation and modification of AnalysisDisplayStyle from a plug-in is optional; end users can have the same control over the presentation of the analysis results with the Revit UI.

The data model supported by Revit API requires that analysis results are specified at a certain set of points, and that at each point one or more distinct numbers ("measurements") are computed. The number of measurements must be the same at all model points. The results data is transient; it is stored only in the model until the document is closed. If the model is saved, closed, and reopened the analysis results will not be present.

27.1 Manager for analysis results

A new SpatialFieldManager can be added to a view using the static SpatialFieldManager.CreateSpatialFieldManager() method. Only one manager can be associated with a view. If a view already has a SpatialFieldManager, it can be retrieved with the static method GetSpatialFieldManager().

CreateSpatialFieldManager() takes a parameter for the number of measurements that will be calculated for each point. This number defines how many results values will be associated with each point at which results are calculated. For example, if average solar radiation is computed for every month of the year, each point would have 12 corresponding values.

To add analysis results to the view, call AddSpatialFieldPrimitive() to create a new analysis results container. Four overloads of this method exist to create primitives associated with:

- A Reference (to a curve or a face)
- A curve and a transform
- A face and a transform
- No Revit geometry – To improve performance when creating many data points that are not related to Revit geometry, it is recommended to create multiple primitives with no more than 500 points each instead of one large primitive containing all points.

A typical use of the transform overloads will be to locate the results data offset from geometry in the Revit model, for example, 3 feet above a floor.

The AddSpatialFieldPrimitive() method returns a unique integer identifier of the primitive within the SpatialFieldManager, which can later be used to remove (RemoveSpatialFieldPrimitive()) or modify the primitive (UpdateSpatialFieldPrimitive()).

27.2 Creating analysis results data

Once a primitive has been added to the SpatialFieldManager, analysis results can be created and added to the analysis results container using the UpdateSpatialFieldPrimitive() method. This method takes a set of domain points (FieldDomainPoints) where results are calculated and a set of values (FieldValues) for each point. The number of FieldValues must correspond to the number of domain points. However, each domain point can have an array of values, each for a separate measurement at this point.

The following example creates a simple set of analysis results on an element face selected by the user. The SDK sample SpatialFieldGradient demonstrates a more complex use case where each point has multiple associated values.

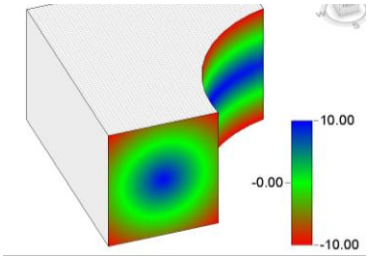
Code Region 27-1: Creating Analysis Results

```
Document doc = commandData.Application.ActiveUIDocument.Document;
UIDocument uiDoc = commandData.Application.ActiveUIDocument;

SpatialFieldManager sfm = SpatialFieldManager.GetSpatialFieldManager(doc.ActiveView);
if (null == sfm)
{
    sfm = SpatialFieldManager.CreateSpatialFieldManager(doc.ActiveView, 1);
}

Reference reference = uiDoc.Selection.PickObject(ObjectType.Face, "Select a face");
int idx = sfm.AddSpatialFieldPrimitive(reference);

Face face = reference.GeometryObject as Face;

IList<UV> uvPts = new List<UV>();
BoundingBoxUV bb = face.GetBoundingBox();
UV min = bb.Min;
UV max = bb.Max;
uvPts.Add(new UV(min.U,min.V));
uvPts.Add(new UV(max.U,max.V));

FieldDomainPointsByUV pnts = new FieldDomainPointsByUV(uvPts);

List<double> doubleList = new List<double>();
IList<ValueAtPoint> valList = new List<ValueAtPoint>();
doubleList.Add(0);
valList.Add(new ValueAtPoint(doubleList));
doubleList.Clear();
doubleList.Add(10);
valList.Add(new ValueAtPoint(doubleList));

FieldValues vals = new FieldValues(valList);

sfm.UpdateSpatialFieldPrimitive(idx, pnts, vals);
```

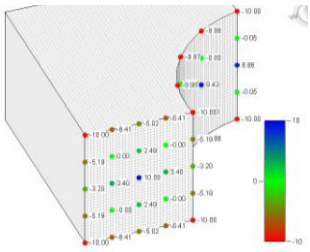
27.3 Analysis Results Display

The AnalysisDisplayStyle class can be used to control how the analysis results are displayed in the view. The static CreateAnalysisDisplayStyle() method can create either a colored surface display style or a markers with text style. For either style, the color and legend settings can also be set.

Once a new AnalysisDisplayStyle is created, use the View.AnalysisDisplayStyleId to assign the style to a view. Although the analysis results are not saved with the document, analysis display styles and their assignment to a view are saved with the model.



"Colored surface" Display Style



"Markers with text" Display Style

The following example creates a new analysis display style (if not already found in the document) and then assigns it to the current view.

Code Region 27-2: Setting analysis display style for view

```
Document doc = commandData.Application.ActiveUIDocument.Document;

AnalysisDisplayStyle analysisDisplayStyle = null;
// Look for an existing analysis display style with a specific name
FilteredElementCollector collector1 = new FilteredElementCollector(doc);
ICollection<Element> collection =
    collector1.OfClass(typeof(AnalysisDisplayStyle)).ToElements();
var displayStyle = from element in collection
                   where element.Name == "Display Style 1"
                   select element;

// If display style does not already exist in the document, create it
if (displayStyle.Count() == 0)
{
    AnalysisDisplayColoredSurfaceSettings coloredSurfaceSettings =
        new AnalysisDisplayColoredSurfaceSettings();
    coloredSurfaceSettings.ShowGridLines = true;

    AnalysisDisplayColorSettings colorSettings = new AnalysisDisplayColorSettings();
    Color orange = new Color(255, 205, 0);
    Color purple = new Color(200, 0, 200);
    colorSettings.MaxColor = orange;
    colorSettings.MinColor = purple;

    AnalysisDisplayLegendSettings legendSettings = new AnalysisDisplayLegendSettings();
    legendSettings.NumberOfSteps = 10;
    legendSettings.Rounding = 0.05;
    legendSettings.ShowDataDescription = false;
    legendSettings.ShowLegend = true;

    FilteredElementCollector collector2 = new FilteredElementCollector(doc);
    ICollection<Element> elementCollection =
        collector2.OfClass(typeof(TextNoteType)).ToElements();
    var textElements = from element in collector2
                       where element.Name == "LegendText"
                       select element;
```

```
    // if LegendText exists, use it for this Display Style
    if (textElements.Count() > 0)
    {
        TextNoteType textType =
            textElements.Cast<TextNoteType>().ElementAt<TextNoteType>(0);
        legendSettings.SetTextTypeId(textType.Id, doc);
    }
    analysisDisplayStyle = AnalysisDisplayStyle.CreateAnalysisDisplayStyle(doc,
        "Display Style 1", coloredSurfaceSettings, colorSettings, legendSettings);
}
else
{
    analysisDisplayStyle =
        displayStyle.Cast<AnalysisDisplayStyle>().ElementAt<AnalysisDisplayStyle>(0);
}

// now assign the display style to the view
doc.ActiveView.AnalysisDisplayStyleId = analysisDisplayStyle.Id;
```

27.4 Updating Analysis Results

The Revit analysis framework does not update results automatically, and potentially any change to Revit model can invalidate results.

In order to keep results up to date, API developers should use Dynamic Model Update triggers or subscribe to the DocumentChanged event to be notified when the Revit model has changed and previously calculated results may be invalid and in need of recalculation. For an example showing Dynamic Model Update together with Analysis Visualization, see the DistanceToSurfaces sample in the Revit SDK.

28 **Revit** Architecture

This chapter covers API functionality that is specific to Revit Architecture, namely:

- Functionality related to rooms (Element.Room, RoomTag, etc.)
- Limited energy analysis functions

28.1 Rooms

The following sections cover information about the room class, its parameters, and how to use the room class in the API.

28.1.1 Room, Area, and Tags

The Room class is used to represent rooms and elements such as room schedule and area plans. The properties and create function for different rooms, areas, and their corresponding tags in the API are listed in the following table:

Element	Class	Category	Boundary	Location	Can Create
Room in Plan View	Room	OST_Rooms	Has if in an enclosed region	LocationPoint	NewRoom() and NewRooms() except for NewRoom(Phase)
Room in Schedule View	Room	OST_Rooms	Null	Null	NewRoom(Phase)
Area	Room	OST_Areas	Always has	LocationPoint	No
Room Tag	RoomTag	OST_RoomTags		LocationPoint	Creation.Document. NewRoomTag()
Area Tag	FamilySymbol	OST_AreaTags		LocationPoint	No

Table 54: Room, Area, and Tags relationship

Note: Room.Name is the combination of the room name and room number. Use the ROOM_NAME BuiltInParameter to get the room name. In the following picture, the Room.Name returns "Master Bedroom 2".

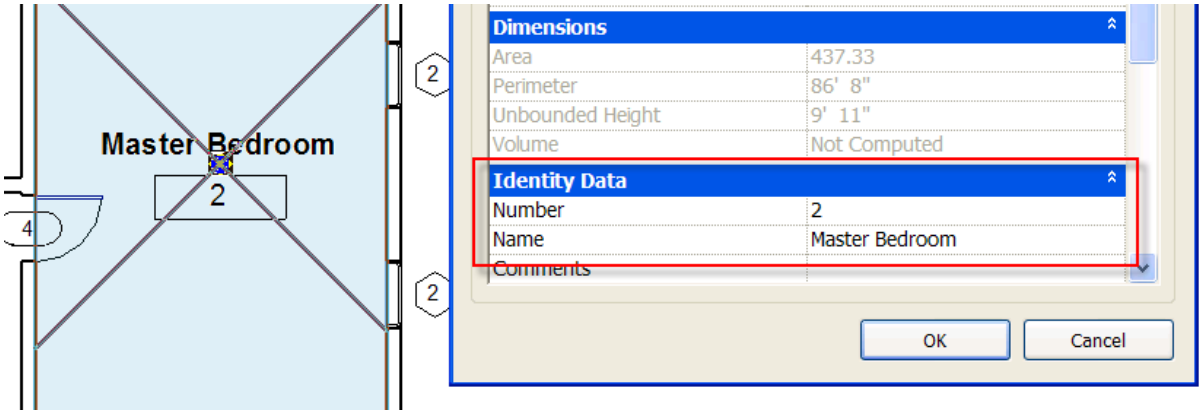


Figure 136: Room name and number

Note: As an Annotation Element, the specific view is available using RoomTag.View. Never try to set the RoomTag.Name property because the name is automatically assigned; otherwise an exception is thrown.

28.1.2 Creating a room

The following code illustrates the simplest way to create a room at a certain point in a specific level:

Code Region 28-1: Creating a room

```
Room CreateRoom(Autodesk.Revit.DB.Document document, Level level)
{
    // Create a UV structure which determines the room location
    UV roomLocation = new UV(0, 0);

    // Create a new room
    Room room = document.Create.NewRoom(level, roomLocation);
    if (null == room)
    {
        throw new Exception("Create a new room failed.");
    }

    return room;
}
```

Rooms can be created in a room schedule then inserted into a plan circuit.

- The Document.NewRoom(Phase) method is used to create a new room, not associated with any specific location, and insert it into an existing schedule. Make sure the room schedule exists or create a room schedule in the specified phase before you make the call.
- The Document.NewRoom(Room room, PlanCircuit circuit) method is used to create a room from a room in a schedule and a PlanCircuit.
  - The input room must exist only in the room schedule, meaning that it does not display in any plan view.
  - After invoking the method, a model room with the same name and number is created in the view where the PlanCircuit is located.

For more details about PlanCircuit, see the Plan Topology section in this chapter.

The following code illustrates the entire process:

Code Region 28-2: Creating and inserting a room into a plan circuit

```
Room InsertNewRoomInPlanCircuit(Autodesk.Revit.DB.Document document, Level level, Phase
newConstructionPhase)
{
    // create room using Phase
    Room newScheduleRoom = document.Create.NewRoom(newConstructionPhase);

    // set the Room Number and Name
    string newRoomNumber = "101";
    string newRoomName = "Class Room 1";
    newScheduleRoom.Name = newRoomName;
    newScheduleRoom.Number = newRoomNumber;

    // Get a PlanCircuit
```

```
    PlanCircuit planCircuit = null;
    // first get the plan topology for given level
    PlanTopology planTopology = document.get_PlanTopology(level);

    // Iterate circuits in this plan topology
    foreach (PlanCircuit circuit in planTopology.Circuits)
    {
        // get the first circuit we find
        if (null != circuit)
        {
            planCircuit = circuit;
            break;
        }
    }

    Room newRoom2 = null;
    Transaction transaction = new Transaction(document, "Create Room");
    if (transaction.Start() == TransactionStatus.Started)
    {
        // The input room must exist only in the room schedule,
        // meaning that it does not display in any plan view.
        newRoom2 = document.Create.NewRoom(newScheduleRoom, planCircuit);
        // a model room with the same name and number is created in the
        // view where the PlanCircuit is located
        if (null != newRoom2)
        {
            // Give the user some information
            TaskDialog.Show("Revit", "Room placed in Plan Circuit successfully.");
        }
        transaction.Commit();
     }

    return newRoom2;
}
```

You can also create rooms in certain levels in batches. The Document.NewRooms() method can create rooms for every enclosed region in a given level, to a given phase, or based on a list of RoomCreationData objects.

Once a room has been created and added to a location, it can be removed from the location (but still remains in available in the project) by using the Room.Unplace() method. It can then be placed in a new location.

28.1.3 **Room Boundary**

Rooms have boundaries that create an enclosed region where the room is located.

- Boundaries include the following elements:
  - Walls
  - Model lines

- Columns
- Roofs

28.1.3.1 Retrieving Room Boundaries

The boundary around a room is contained in its Boundary property. The Room.Boundary property is null when the room is not in an enclosed region or only exists in the schedule. The Room.BoundaryArray is an array of BoundarySegment objects.

The following figure shows a room boundary selected in the Revit UI:

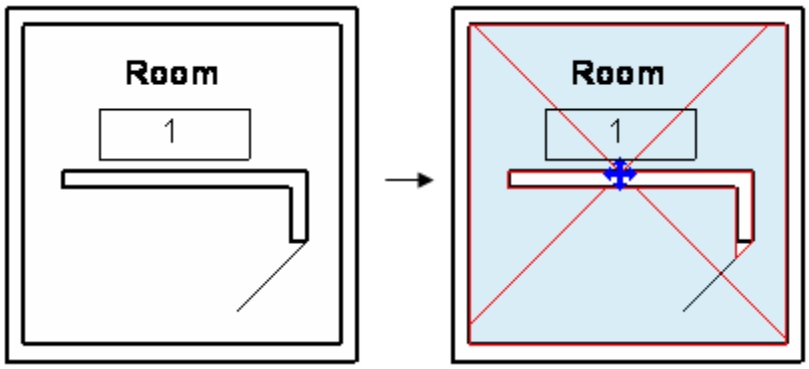


Figure 137: Room boundary

The Room.BoundaryArray segment array size depends on the enclosed region topology. Each BoundarySegmentArray makes a circuit or a continuous line in which one segment joins the next. The following pictures provide several examples. In the following pictures, all walls are Room-Bounding and the model lines category is OST_AreaSeparationLines. If an element is not Room-Bounding, it is excluded from the elements to make the boundary.

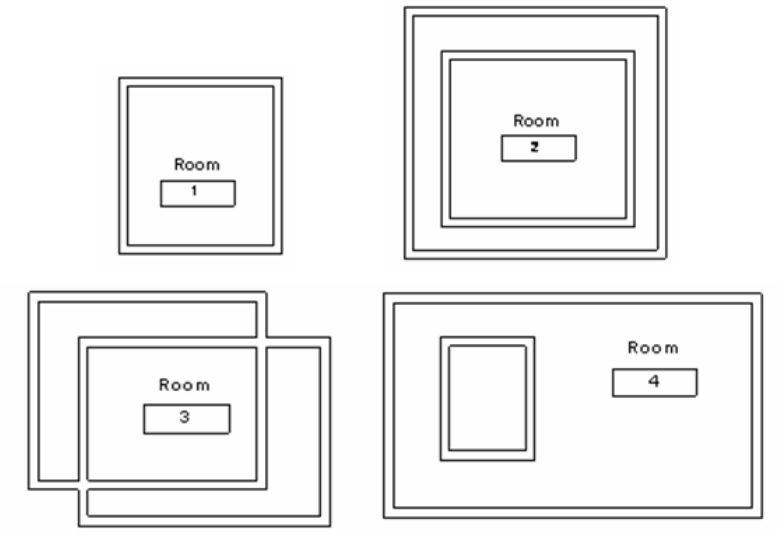


Figure 138: Rooms 1, 2, 3, 4

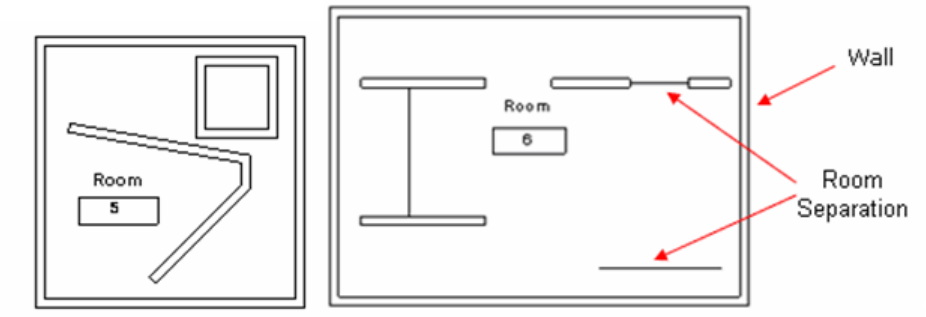


Figure 139: Room 5, 6

The following table provides the Room.Boundary.Size results for the previous rooms:

Room	Room.Boundary.Size
Room 1 Room 2 Room 3	1
Room 4	2
Room 5 Room 6	3

Table 55: Room.Boundary.Size

Note: Walls joined by model lines are considered continuous line segments. Single model lines are ignored.

After getting BoundarySegmentArray, get the BoundarySegment by iterating the array.

28.1.3.2 **BoundarySegment**

The segments that make the region are represented by the BoundarySegment class; its Element property returns the corresponding element with the following conditions:

- For a ModelCurve element, the category must be BuiltInCategory.OST_AreaSeparationLines meaning that it represents a Room Separator.
- For other elements such as wall, column, and roof, if the element is a room boundary, the Room Bounding parameter (BuiltInParameter.WALL_ATTR_ROOM_BOUNDING) must be true as in the following picture.

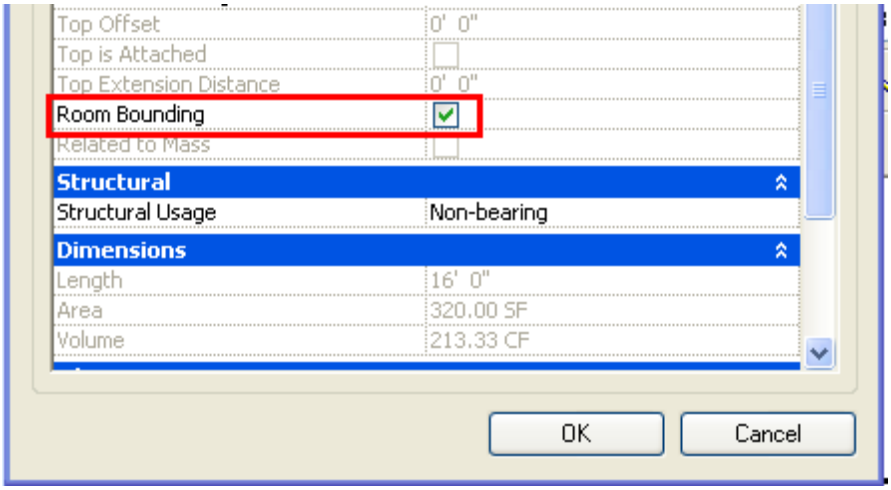


Figure 140: Room Bounding property

The WALL_ATTR_ROOM_BOUNDING BuiltInParameter is set through the API:

Code Region 28-3: Setting room bounding

```
public void SetRoomBounding(Wall wall)
{
    Parameter parameter = wall.get_Parameter(BuiltInParameter.WALL_ATTR_ROOM_BOUNDING);
    parameter.Set(1);    //set "Room Bounding" to true
    parameter.Set(0);    //set "Room Bounding" to false
}
```

Notice how the roof forms the BoundarySegment for a room in the following pictures. The first picture shows Level 3 in the elevation view. The room is created in the Level 3 floor view. The latter two pictures show the boundary of the room and the house in 3D view.

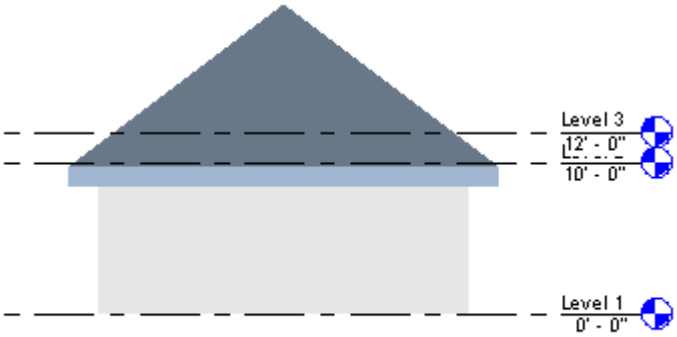


Figure 141: Room created in level 3 view

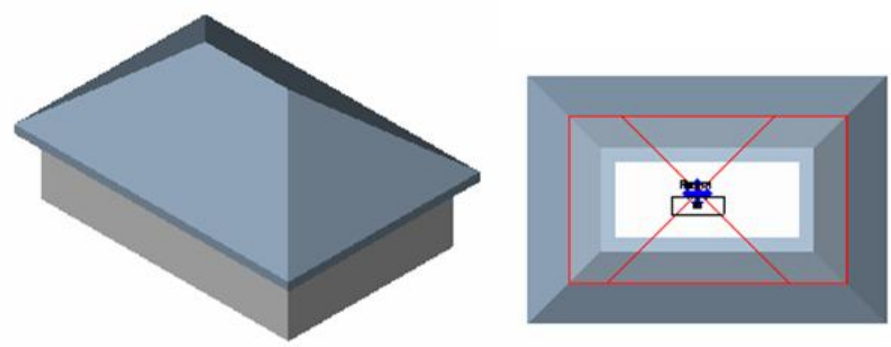


Figure 142: Room boundary formed by roof

The area boundary can only be a ModelCurve with the category Area Boundary (BuiltInCategory.OST_AreaSchemeLines) while the boundary of the displayed room can be walls and other elements.

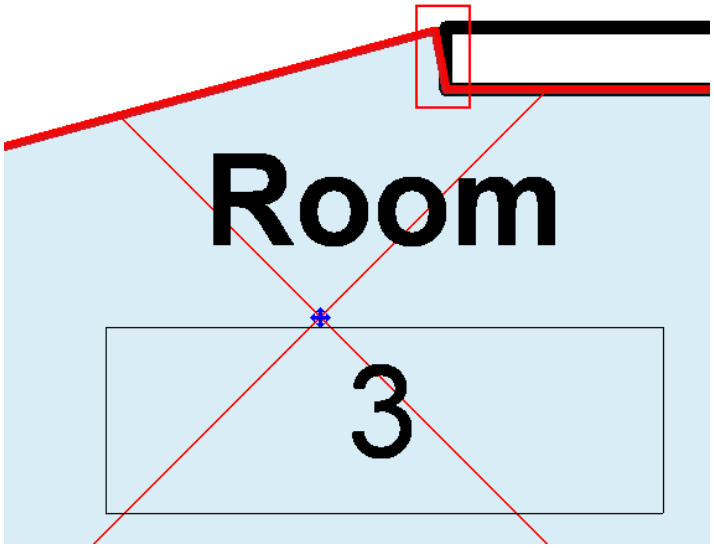


Figure 143: Wall end edge

If the BoundarySegment corresponds to the curve between the room separation and wall as the previous picture shows:

- The Element property returns null
- The Curve is not null.

28.1.3.3 Boundary and Transaction

When you call Room.Boundary after creating an Element using the API such as a wall, the wall can change the room boundary. You must make sure the data is updated.

The following illustrations show how the room changes after a wall is created using the Revit Platform API.

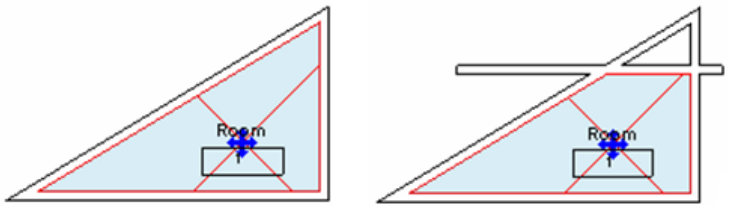


Figure 144: Added wall changes the room boundary

To update the room boundary data, use the transaction mechanism in the following code:

Code Region 28-4: Using a transaction to update room boundary

```
public void UpdateRoomBoundary(UIApplication application, Room room, Level level)
{
    Document document = application.ActiveUIDocument.Document;

    //Get the size before creating a wall
    int size = room.Boundary.get_Item(0).Size;
    string message = "Room boundary size before wall: " + size;

    //Prepare a line
    XYZ startPos = new XYZ(-10, 0, 0);
    XYZ endPos = new XYZ(10, 0, 0);
    Line line = application.Application.Create.NewLine(startPos, endPos, true);

    //Create a new wall and enclose the creating into a single transaction
    Transaction transaction = new Transaction(document, "Create Wall");
    if (transaction.Start() == TransactionStatus.Started)
    {
        Wall wall = document.Create.NewWall(line, level, false);
        transaction.Commit();

        //Get the new size
        size = room.Boundary.get_Item(0).Size;
        message += "\nRoom boundary size after wall: " + size;

        TaskDialog.Show("Revit", message);
        transaction.Commit();
    }
}
```

For more details, see the Transaction chapter.

28.1.4**Plan Topology**

The level plan that rooms lie in have a topology made by elements such as walls and room separators. The PlanTopology and PlanCircuit classes are used to present the level topology.

- Get the PlanTopology object from the Document object using the Level. In each plan view, there is one PlanTopology corresponding to every phase.
- The same condition applies to BoundarySegment, except room separators and Elements whose Room Bounding parameter is true can be a side (boundary) in the PlanCircuit.

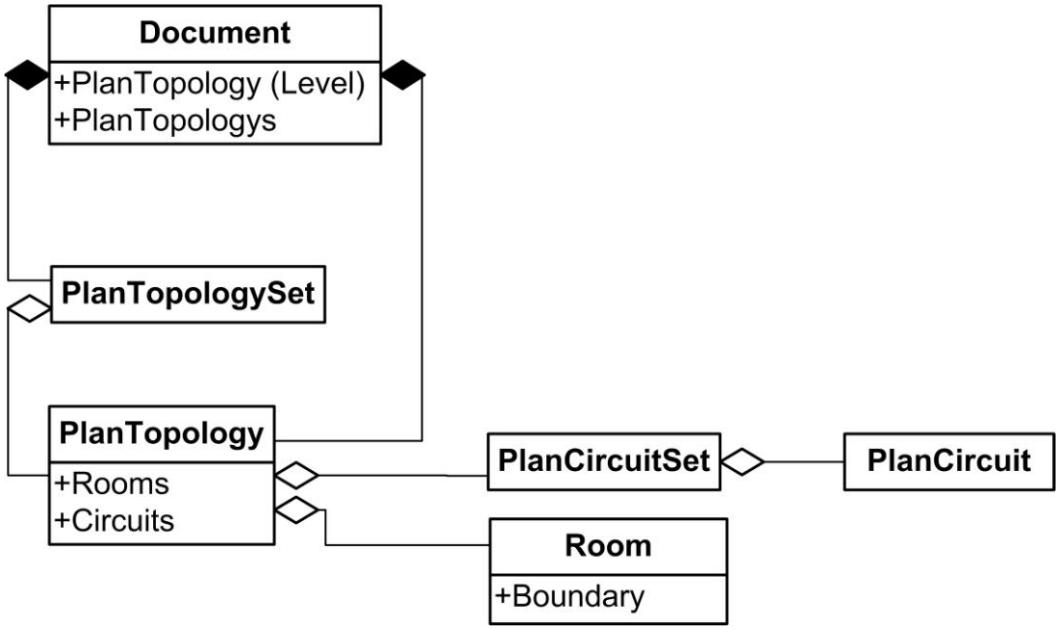


Figure 145: Room and Plan Topology diagram

The PlanCircuit.SideNum property returns the circuit side number, which is different from the BoundarySegmentArray.Size.

- The BoundarySegmentArray.Size recognizes the bottom wall as two walls if there is a branch on the wall.
- PlanCircuit.SideNum always sees the bottom wall in the picture as one regardless of the number of branches.

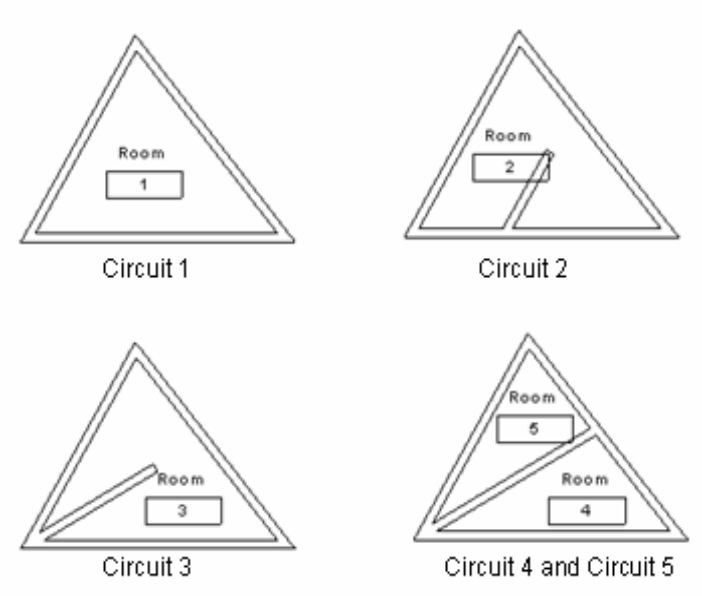


Figure 146: Compare room boundary with PlanCircuit

Circuit	Circuit.SideNum	BoundarySegmentArray.Size for Room
Circuit 1	3	3 (Room1)
Circuit 2	4 +2 = 6	4 +3 = 7 (Room2)
Circuit 3	3 +2 = 5	3 +3 = 6 (Room3)
Circuit 4	3	3 (Room4)
Circuit 5	3	3 (Room5)

Table 56: Compare Room Boundary with PlanCircuit

28.1.5**Room and FamilyInstance**

Doors and Windows are special family instances related to Room. Only doors are discussed here since the only difference is that windows have no handle to flip.

The following characteristics apply to doors:

- Door elements can exist without a room.
- In the API (and only in the API), a Door element has two additional properties that refer to the regions on the two opposite sides of a door: ToRoom and FromRoom
- If the region is a room, the property's value would be a Room element.
- If the region is not a room, the property will return null. Both properties may be null at the same time.
- The region on the side into which a door opens, will be ToRoom. The room on the other side will be FromRoom.
- Both properties get dynamically updated whenever the corresponding regions change.

In the following pictures, five doors are inserted into walls without flipping the facing. The table lists the FromRoom, ToRoom, and Room properties for each door. The Room property belongs to all Family Instances.

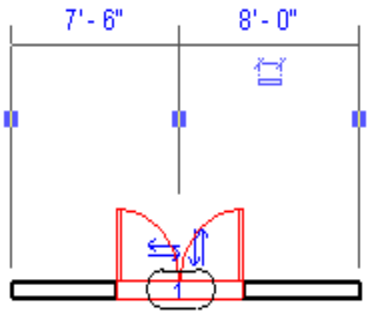


Figure 147: Door 1

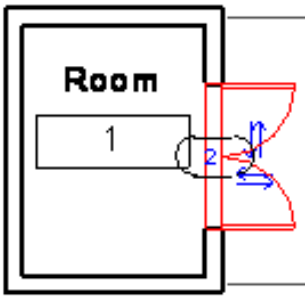


Figure 148: Door 2

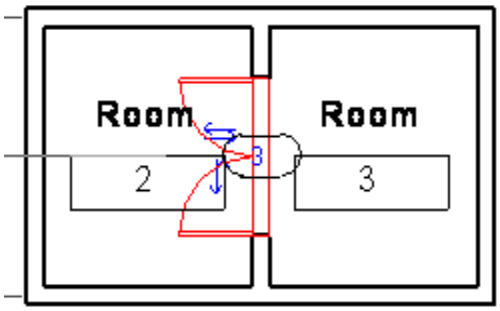


Figure 149: Door 3

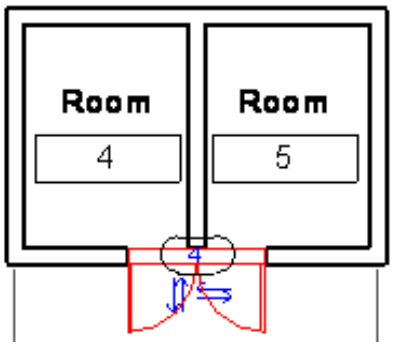


Figure 150: Door 4

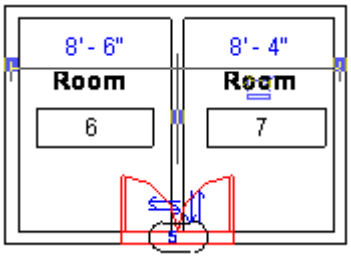


Figure 151: Door 5

Door	FromRoom	ToRoom	Room
Door 1	null	null	null
Door 2	Room 1	null	null
Door 3	Room 3	Room 2	Room 2
Door 4	Room 4	null	null
Door 5	null	Room 6	Room 6

Table 57: Door Properties

All family instances have the Room property, which is the room where an instance is located in the last project phase. Windows and doors face into a room. Change the room by flipping the door or window facing, or by calling FamilyInstance.FlipFromToRoom(). For other kinds of instances, such as beams and columns, the Room is the room that has the same boundary as the instance.

The following code illustrates how to get the Room from the family instance. It is necessary to check if the result is null or not.

Code Region 28-5: Getting a room from the family instance

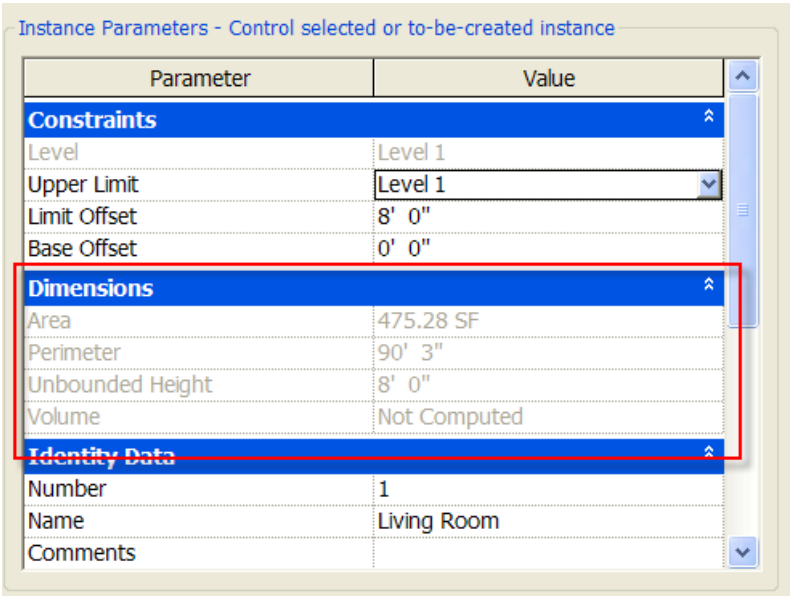
```
public void GetRoomInfo(FamilyInstance familyInstance)
{
    Room room = familyInstance.Room;
    room = familyInstance.FromRoom;  //for door and window family only
    room = familyInstance.ToRoom;    //for door and window family only
    if (null != room)
    {
        //use the room...
    }
}
```

28.1.6 Other Room Properties

The Room class has several other properties you can use to get information about the object. Rooms have these read-only dimension properties:

- Area
- Perimeter
- UnboundedHeight
- Volume

- ClosedShell



This example displays the dimension information for a selected room. Note that the volume calculations setting must be enabled, or the room volume is returned as 0.

Code Region 28-6: Getting a room's dimensions

```
public void GetRoomDimensions(Document doc, Room room)
{
    String roominfo = "Room dimensions:\n";
    // turn on volume calculations:
    VolumeCalculationOptions options = new VolumeCalculationOptions();
    options.VolumeComputationEnable = true;
    doc.Settings.VolumeCalculationSetting.VolumeCalculationOptions = options;
    roominfo += "Vol: " + room.Volume + "\n";
    roominfo += "Area: " + room.Area + "\n";
    roominfo += "Perimiter: " + room.Perimeter + "\n";
    roominfo += "Unbounded height: " + room.UnboundedHeight + "\n";
    TaskDialog.Show("Revit",roominfo);
}
```

The ClosedShell property for a Room (or Space) is the geometry formed by the boundaries of the open space of the room (walls, floors, ceilings, roofs, and boundary lines). This property is useful if you need to check for intersection with other physical element in the model with the room, for example, to see if part or all of the element is located in the room. For an example, see the RoofsRooms sample application, included with the Revit SDK, where ClosedShell is used to check whether a room is vertically unbounded.

In addition, you can get or set the base offset and limit offset for rooms with these properties:

- BaseOffset
- LimitOffset

You can get or set the level that defines the upper limit of the room with the UpperLimit property.

28.2 Energy Data

The EnergyDataSettings object represents the gbXML Parameters in the Revit project. To view the parameters, from the Revit UI, select Project Information from the Project Settings panel on the Manage tab. The Project Information dialog box appears.

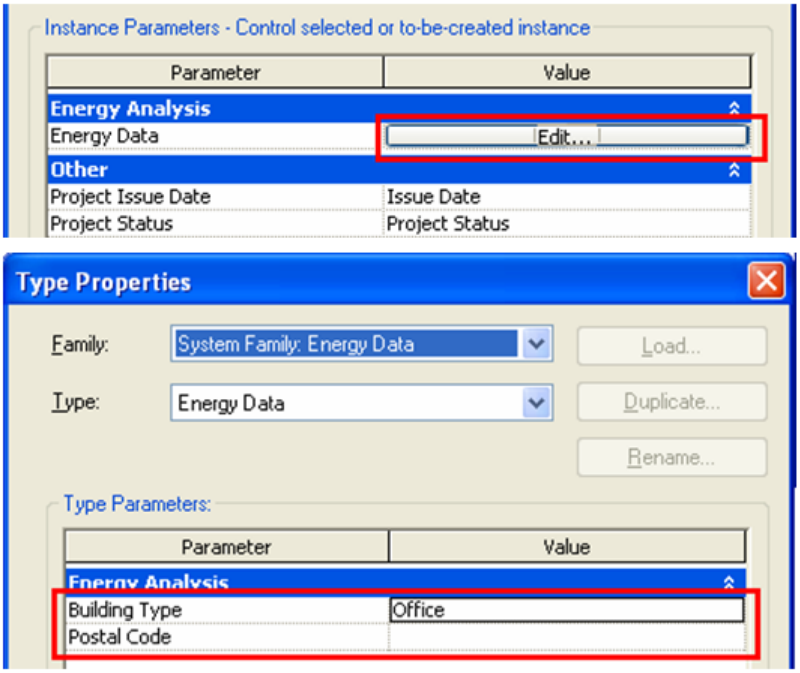


Figure 152: EnergyDataSettings

The EnergyDataSettings object is derived from the Element base object. It is unique in each project, similar to ProjectInformation. Though EnergyDataSettings is a subclass of the Element class, most of the members inherited from the Element return null or an empty set except for Name, Id, UniqueId, and Parameters.

The following code sample uses the EnergyDataSettings class. The result appears in a TaskDialog after invoking the command.

Code Region 28-7: Using the EnergyDataSettings class

```
public void GetInfo_EnergyData(Document document)
{
    EnergyDataSettings energyData = EnergyDataSettings.GetFromDocument(document);

    if (null != energyData)
    {
        string message = "energyData : ";
        message += "\nBuildingType : " + energyData.BuildingType;
        TaskDialog.Show("Revit", message);
    }
}
```

29 Revit Structure

Certain API features that only exist in Revit Structure products are discussed in the following sections:

- Structural Model Elements - Discusses specific Elements and their properties that only exist in the Revit Structure product.
- AnalyticalModel - Discusses analytical model-related classes such as AnalyticalModel, RigidLink, and AnalyticalModelSupport.
- Loads - Discusses Load Settings and three kinds of Loads.
- Your Analysis Link - Provides suggestions for API users who want to link the Revit Structure product to certain Structural Analysis applications.

This chapter contains some advanced topics. If you are not familiar with the Revit Platform API, read the basic chapters first, such as Getting Started, Elements Essentials, Parameter, and so on.

29.1 Structural Model Elements

Structural Model Elements are, literally, elements that support a structure such as columns, rebar, trusses, and so on. This section discusses how to manipulate these elements.

29.1.1 Column, Beam, and Brace

Structural column, beam, and brace elements do not have a specific class such as the StructuralColumn class but they are in the FamilyInstance class form.

Note: Though the StructuralColumn, StructuralBeam, and StructuralBrace classes do not exist in the current API, they are used in this chapter to indicate the FamilyInstance objects corresponding to structural columns, beams, and braces.

Though Structural column, beam, and brace are all FamilyInstance objects in the API, they are distinguished by the StructuralType property.

Code Region 29-1: Distinguishing between column, beam and brace

```
public void GetStructuralType(FamilyInstance familyInstance)
{
    string message = "";
    switch (familyInstance.StructuralType)
    {
        case StructuralType.Beam:
            message = "FamilyInstance is a beam.";
            break;
        case StructuralType.Brace:
            message = "FamilyInstance is a brace.";
            break;
        case StructuralType.Column:
            message = "FamilyInstance is a column.";
            break;
        case StructuralType.Footing:
            message = "FamilyInstance is a footing.";
            break;
        default:
```

```
            message = "FamilyInstance is non-structural or unknown framing.";
            break;
    }

    TaskDialog.Show("Revit", message);
}
```

You can filter out FamilySymbol objects corresponding to structural columns, beams, and braces in using categories. The category for Structural beams and braces is BuiltInCategory.OST_StructuralFraming.

Code Region 29-2: Using BuiltInCategory.OST_StructuralFraming

```
public void GetBeamAndColumnSymbols(Document document)
{
    FamilySymbolSet columnTypes = new FamilySymbolSet();
    FamilySymbolSet framingTypes = new FamilySymbolSet();
    FilteredElementCollector collector = new FilteredElementCollector(document);
    ICollection<Element> elements = collector.OfClass(typeof(Family)).ToElements();

    foreach(Element element in elements)
    {
        Family tmpFamily = element as Family;
        Category category = tmpFamily.FamilyCategory;
        if (null != category)
        {
            if ((int)BuiltInCategory.OST_StructuralColumns == category.Id.IntegerValue)
            {
                foreach (FamilySymbol tmpSymbol in tmpFamily.Symbols)
                {
                    columnTypes.Insert(tmpSymbol);
                }
            }
            else if ((int)BuiltInCategory.OST_StructuralFraming == category.Id.IntegerValue)
            {
                foreach (FamilySymbol tmpSymbol in tmpFamily.Symbols)
                {
                    framingTypes.Insert(tmpSymbol);
                }
            }
        }
    }

    string message = "Column Types: ";
    FamilySymbolSetIterator fsItor = columnTypes.ForwardIterator();
    fsItor.Reset();
    while (fsItor.MoveNext())
    {
        FamilySymbol familySybmol = fsItor.Current as FamilySymbol;
        message += "\n" + familySybmol.Name;
```

```
    }

    TaskDialog.Show("Revit",message);
}
```

You can get and set beam setback properties with the FamilyInstance.ExtensionUtility property. If this property returns null, the beam setback can't be modified.

29.1.2 AreaReinforcement and PathReinforcement

Find the AreaReinforcementCurves for AreaReinforcement by iterating the Curves property which is an ElementArray. In edit mode, AreaReinforcementCurves are the purple curves (red when selected) in the Revit UI. From the Element Properties dialog box you can see AreaReinforcementCurve parameters. Parameters such as Hook Types and Orientation are editable only if the Override Area Reinforcement Setting parameter is true.

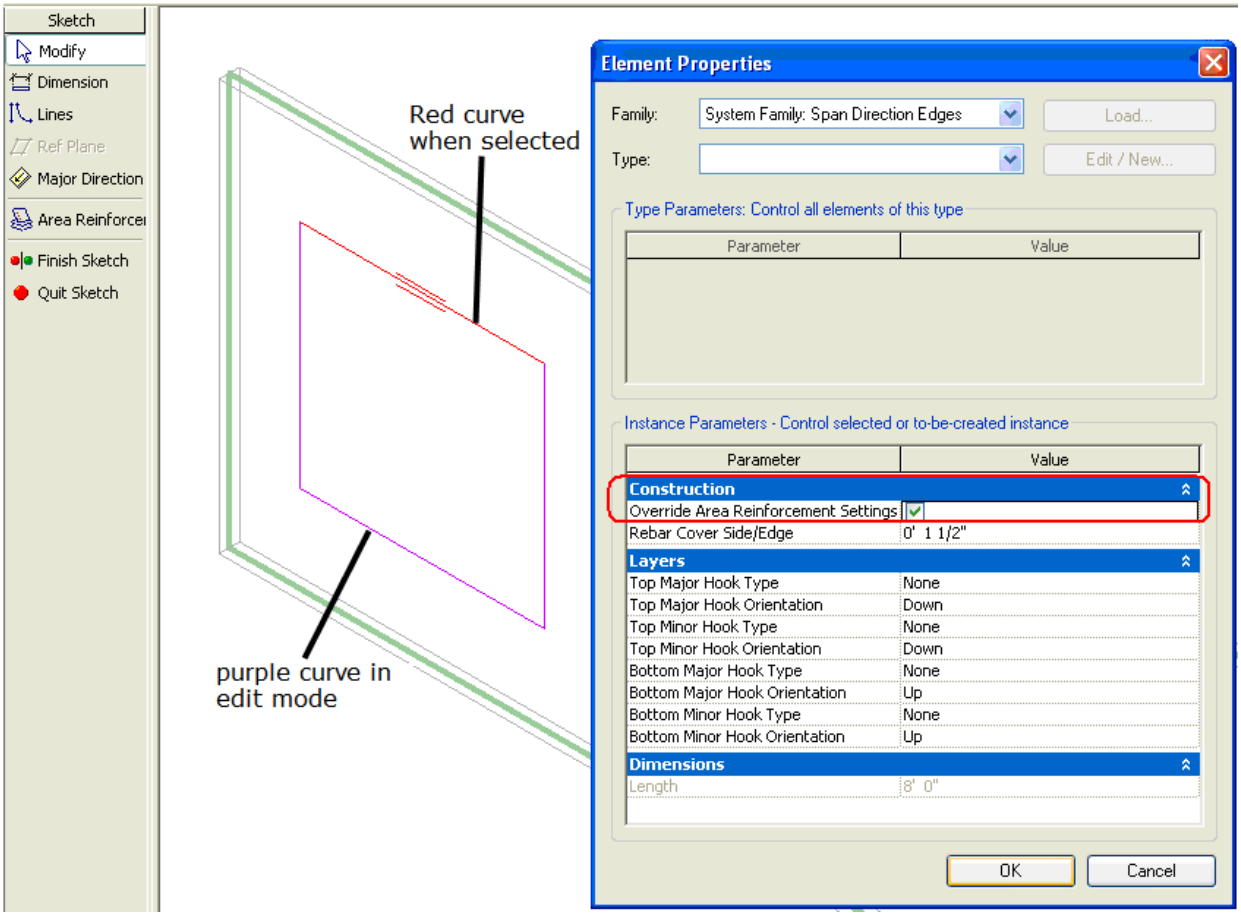


Figure 153: AreaReinforcementCurve in edit mode

There is no way to control the direction except by using the NewAreaReinforcement() method (the last XYZ direction input parameter).

Code Region 29-3: NewAreaReinforcement()

```
public AreaReinforcement NewAreaReinforcement(
   Element host, CurveArray curves, XYZ direction);
```

Although the AreaReinforcement Curves property returns a set of AreaReinforcementCurves, PathReinforcementCurves returns a ModelCurve. There is no way to flip the PathReinforcement except by using the NewPathReinforcement() method (the last input parameter).

Code Region 29-4:NewPathReinforcement()

```
public PathReinforcement NewPathReinforcement(
   Element host, CurveArray curves, bool flip);
```

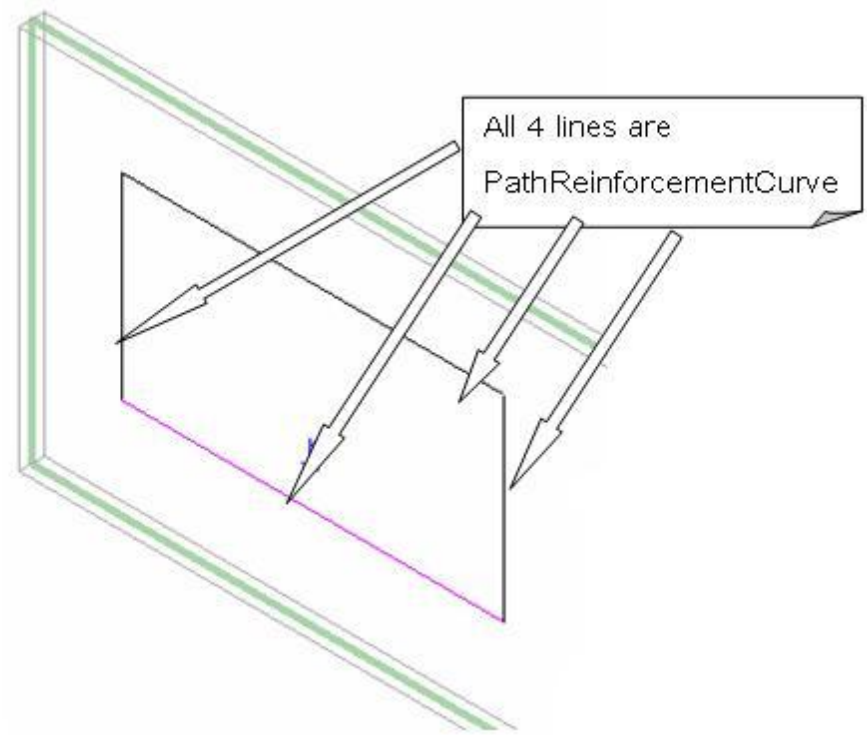


Figure 154: PathReinforcement.PathReinforcementCurve in edit mode

When using NewAreaReinforcement() and NewPathReinforcement() methods to create objects, you must decide on which host Element face it will lay. Currently AreaReinforcement and PathReinforcement are only created on the PlanarFace retrieved from the Wall or Floor object. After removing the faces from the Wall or Floor geometry, you can filter the PlanarFace out as follows:

- Downcast the Face to PlanarFace:

Code Region 29-5: Downcasting Face to PlanarFace

```
PlanarFace planarFace = face as PlanarFace;
```

- If it is a PlanarFace, get its Normal and Origin:

Code Region 29-6: Getting Normal and Origin

```
private void GetPlanarFaceInfo(Face face)
{
    PlanarFace planarFace = face as PlanarFace;
    if (null != planarFace)
    {
        XYZ origin = planarFace.Origin;
        XYZ normal = planarFace.Normal;
```

```
            XYZ vector = planarFace.get_Vector(0);
        }
    }
```

- Filter out the right face based on normal and origin. For example:
  - For a general vertical Wall, the face is located using the following factors:
    - The face is vertical; (normal.Z == 0.0)
    - Parallel face directions are opposite:
    - (normal1.X = - normal2.X; normal1.Y = - normal2.Y)
    - Normal must be parallel to the location line.
  - For a general Floor without slope, the factors are:
    - The face is horizontal; (normal.X == 0.0 && normal.Y ==0.0)
    - Judge the top and bottom face; (distinguish 2 faces by normal.Z)

For more details about retrieving an Element's Geometry, refer to the Geometry chapter.

29.1.3 BeamSystem

BeamSystem provides you with full access and edit ability. You can get and set all of its properties, such as BeamSystemType, BeamType, Direction, and Level. BeamSystem.Direction is not limited to one line of edges. It can be set to any XYZ coordinate on the same plane with the BeamSystem.

Note: You cannot change the StructuralBeam AnalyticalModel after the Elevation property is changed in the UI or by the API. In the following picture the analytical model lines stay in the original location after BeamSystem Elevation is changed to 10 feet.

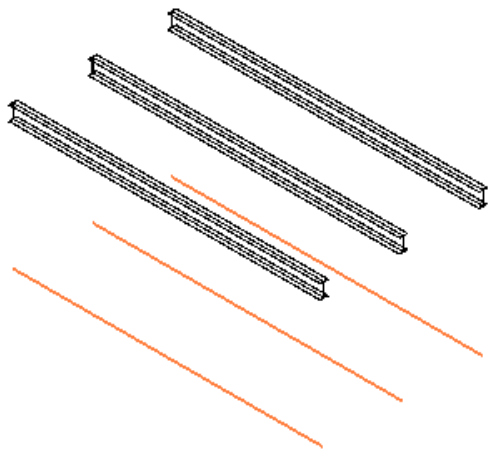


Figure 155: Change BeamSystem elevation

29.1.4 Truss

The Truss class represents all types of trusses in Revit. The TrussType property indicates the type of truss.

Code Region 29-7: Creating a truss over two columns

```
Truss CreateTruss(Autodesk.Revit.DB.Document document,
            FamilyInstance column1, FamilyInstance column2)
{
```

```
    Truss truss = null;
    Transaction transaction = new Transaction(document, "Add Truss");
    if (transaction.Start() == TransactionStatus.Started)
    {
        //sketchPlane
        XYZ origin = new XYZ(0, 0, 0);
        XYZ xDirection = new XYZ(1, 0, 0);
        XYZ yDirection = new XYZ(0, 1, 0);
        Plane plane = document.Application.Create.NewPlane(xDirection, yDirection, origin);
        SketchPlane sketchPlane = document.Create.NewSketchPlane(plane);

        //new base Line - use line that spans two selected columns
        AnalyticalModel frame1 = column1.GetAnalyticalModel() as AnalyticalModel;
        XYZ centerPoint1 = (frame1.GetCurve() as Line).get_EndPoint(0);

        AnalyticalModel frame2 = column2.GetAnalyticalModel() as AnalyticalModel;
        XYZ centerPoint2 = (frame2.GetCurve() as Line).get_EndPoint(0);

        XYZ startPoint = new XYZ(centerPoint1.X, centerPoint1.Y, 0);
        XYZ endPoint = new XYZ(centerPoint2.X, centerPoint2.Y, 0);
        Autodesk.Revit.DB.Line baseLine = null;

        try
        {
            baseLine = document.Application.Create.NewLineBound(startPoint, endPoint);
        }
        catch (System.ArgumentException)
        {
            throw new Exception("Selected columns are too close to create truss.");
        }

        // use the active view for where the truss's tag will be placed; View used in
        // NewTruss should be plan or elevation view parallel to the truss's base line
        Autodesk.Revit.DB.View view = document.ActiveView;

        // Get a truss type for the truss
        TrussType trussType = null;
        foreach (TrussType currentType in document.TrussTypes)
        {
            if (null != currentType)
            {
                trussType = currentType;
                break;
            }
        }

        if (null != trussType)
        {
```

```
            truss = document.Create.NewTruss(trussType, sketchPlane, baseLine, view);
            transaction.Commit();
        }
        else
        {
            transaction.RollBack();
            throw new Exception("No truss types found in document.");
        }
    }

    return truss;
}
```

29.1.5 **Rebar**

The Rebar class represents rebar used to reinforce suitable elements, such as concrete beams, columns, slabs or foundations.

You can create rebar objects using one of two Document.Create.NewRebar() method overloads.

Name	Description

`public Rebar NewRebar(`
`RebarStyle style,`
`RebarBarType rebarType,`
`RebarHookType startHook,`
`RebarHookType endHook,`
`Element host,`
`XYZ norm,`
`CurveArray curves,`
`int startHookOrient,`
`int endHookOrient,`
`bool useExistingShapeIfPossible,`
`bool createNewShape`
`);`
	Creates a new instance of a Rebar element within the project.

`public Rebar NewRebar(`
`RebarShape rebarShape,`
`RebarBarType rebarType,`
`Element host,`
`XYZ origin,`
`XYZ xVec,`
`XYZ yVec`
`);`
	Creates a new Rebar, as an instance of a RebarShape. The instance will have the default shape parameters from the RebarShape, and its location is based on the bounding box of the shape in the shape definition. Hooks are removed from the shape before computing its bounding box. If appropriate hooks can be found in the document, they will be assigned arbitrarily.

The first version creates rebar from an array of curves describing the rebar, while the second creates a Rebar object based on a RebarShape and position.

When using the first NewRebar() method overload, the parameters RebarBarType and RebarHookType are available in the RebarBarTypes and RebarHookTypes property. The RebarBarTypes property and RebarHookTypes property are Document properties.

The following code illustrates how to create Rebar with a specific layout.

Code Region 29-8: Creating rebar with a specific layout

```
Rebar CreateRebar(Autodesk.Revit.DB.Document document, FamilyInstance column, RebarBarType
barType, RebarHookType hookType)
```

```
{
    // Define the rebar geometry information - Line rebar
    LocationPoint location = column.Location as LocationPoint;
    XYZ origin = location.Point;
    XYZ normal = new XYZ(1, 0, 0);
    XYZ rebarLineEnd = new XYZ(origin.X, origin.Y, origin.Z + 9);
    Line rebarLine = document.Application.Create.NewLineBound(origin, rebarLineEnd);

    // Create the line rebar
    CurveArray curves = new CurveArray();
    curves.Append(rebarLine);

    Rebar rebar =
             document.Create.NewRebar(Autodesk.Revit.DB.Structure.RebarStyle.Standard,
                                      barType, hookType, hookType, column, origin, curves,
                                      RebarHookOrientation.Right, RebarHookOrientation.Left,
                                      true, true);

    if (null != rebar)
    {
        // set specific layout for new rebar
        Parameter paramLayout = rebar.get_Parameter(BuiltInParameter.REBAR_ELEM_LAYOUT_RULE);
        paramLayout.Set(1); // 1 = Fixed Number
        Parameter parNum = rebar.get_Parameter(BuiltInParameter.REBAR_ELEM_QUANTITY_OF_BARS);
        parNum.Set(10);
        rebar.ArrayLength = 1.5;
    }

    return rebar;
}
```

Note: For more examples of creating rebar elements, see the Reinforcement and NewRebar sample applications included with the Revit SDK.

The following table lists the integer value for the Parameter REBAR_ELEM_LAYOUT_RULE:

Value	0	1	2	3	4
Description	None	Fixed Number	Maximum Spacing	Number with Spacing	Minimum Clear Spacing

Table 58: Rebar Layout Rule

In the NewRebar() method input parameters, the following rules apply:

- All curves must lie on the same plane as origin.

The Rebar.ScaleToBox() method provides a way to simultaneously set all the shape parameters. The behavior is similar to the UI for placing Rebar.

29.1.5.1 RebarHostData and RebarCoverType

Clear cover is associated with individual faces of valid rebar hosts. You can access the cover settings of a host through the Autodesk.Revit.Elements.RebarHostData object. A simpler, less powerful mechanism for accessing the same settings is provided through parameters.

Cover is defined by a named offset distance, modeled as an element Autodesk.Revit.DB.Structure.RebarCoverType.

29.1.6 BoundaryConditions

There are three types of BoundaryConditions:

- Point
- Curve
- Area

The type and pertinent geometry information is retrieved using the following code:

Code Region 29-9: Getting boundary condition type and geometry

```
public void GetInfo_BoundaryConditions(BoundaryConditions boundaryConditions)
{
    string message = "BoundaryConditions : ";

    Parameter param =
                boundaryConditions.get_Parameter(BuiltInParameter.BOUNDARY_CONDITIONS_TYPE);
    switch (param.AsInteger())
    {
        case 0:
            XYZ point = boundaryConditions.Point;
            message += "\nThis BoundaryConditions is a Point Boundary Conditions.";
            message += "\nLocation point: (" + point.X + ", "
                        + point.Y + ", " + point.Z + ")";
            break;
        case 1:
            message += "\nThis BoundaryConditions is a Line Boundary Conditions.";
            Curve curve = boundaryConditions.get_Curve(0);
            // Get curve start point
            message += "\nLocation Line: start point: (" + curve.get_EndPoint(0).X + ", "
                    + curve.get_EndPoint(0).Y + ", " + curve.get_EndPoint(0).Z + ")";
            // Get curve end point
            message += ";  end point:(" + curve.get_EndPoint(1).X + ", "
                    + curve.get_EndPoint(1).Y + ", " + curve.get_EndPoint(1).Z + ")";
            break;
        case 2:
            message += "\nThis BoundaryConditions is an Area Boundary Conditions.";
            for (int i = 0; i < boundaryConditions.NumCurves; i++)
            {
                Autodesk.Revit.DB.Curve areaCurve = boundaryConditions.get_Curve(i);
                // Get curve start point
                message += "\nCurve start point:(" + areaCurve.get_EndPoint(0).X + ", "
                    + areaCurve.get_EndPoint(0).Y + ", " + areaCurve.get_EndPoint(0).Z + ")";
```

```
                // Get curve end point
                message += "; Curve end point:(" + areaCurve.get_EndPoint(1).X + ", "
                    + areaCurve.get_EndPoint(1).Y + ", " + areaCurve.get_EndPoint(1).Z + ")";
            }
            break;
        default:
            break;
    }

    TaskDialog.Show("Revit", message);
}
```

29.1.7 Other Structural Elements

Some Element derived classes exist in Revit Architecture and Revit Structure products. In this section, methods specific to Revit Structure are introduced. For more information about these classes, see the corresponding parts in the Walls, Floors, Roofs and Openings and Family Instances chapters.

29.1.7.1 Slab

Both Slab (Structural Floor) and Slab Foundation are represented by the Floor class and are distinguished by the IsFoundationSlab property.

The Slab Span Directions are represented by the IndependentTag class in the API and are available as follows:

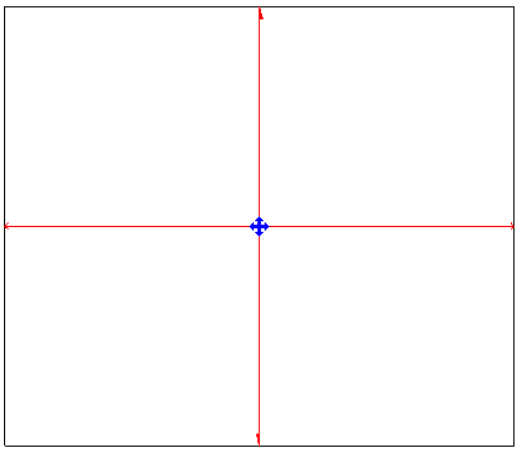


Figure 156: Slab span directions

When using NewSlab() to create a Slab, Span Directions are not automatically created. There is also no way to create them directly.

The Slab compound structure layer Structural Deck properties are exposed by the following properties:

- CompoundStructuralLayer.DeckUsage
- DeckProfile

The properties are outlined in the following dialog box:

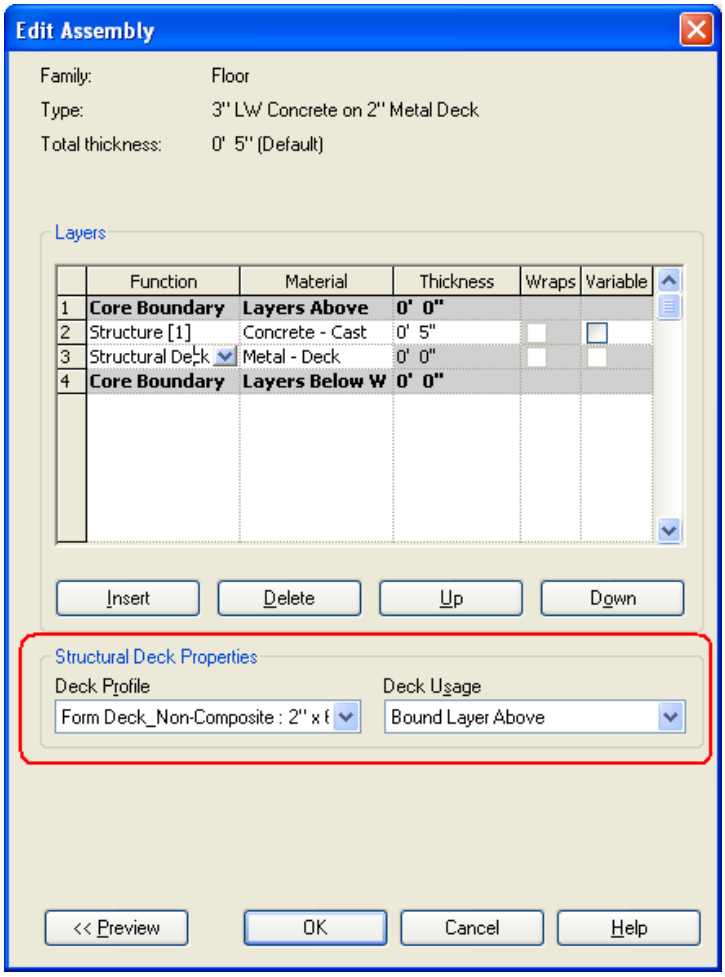


Figure 157: Floor CompoundStructuralLayer properties

29.2 AnalyticalModel

In Revit Structure, an analytical model is the engineering description of a structural physical model.

The following structural elements have structural member analytical models:

- Structural Columns
- Structural Framing elements (such as beams and braces)
- Structural Floors
- Structural Footings
- Structural Walls

An Element's AnalyticalModel can be obtained using the GetAnalyticalModel() method. Depending on the element's family, the AnalyticalModel may not exist. If the AnalyticalModel value does not apply to an element's family, the GetAnalyticalModel() method returns null. Check the value before using this class. AnalyticalModel is made up of the following information:

- Location of the Element with respect to analysis
- Parameter Information, including projection, hard points, approximation, and rigid links
- Support Information
- Adjustment Information, both Manual and Automatic

- Analytical Offset

29.2.1 Analytical Location

Depending on the type of element that corresponds to the AnalyticalModel, the location of the element with respect to analysis can be obtained by one of three methods: GetPoint(), GetCurve() or GetCurves().

Note that the curves retrieved from these methods do not have their Reference properties set. Therefore, they cannot be used for properties such as Curve.EndPointReference. Instead, you can obtain References to the curves and their end points through construction of an AnalyticalModelSelector object containing the necessary information, as the following example demonstrates.

Code Region 29-10: Getting a reference to analytical curve

```
public bool GetReferenceData(FamilyInstance familyInst)
{
    AnalyticalModel analyticalModelFrame = familyInst.GetAnalyticalModel();
    Curve analyticalCurve = analyticalModelFrame.GetCurve();
    if (null != analyticalCurve)
    {
        // test the stable reference to the curve.
        AnalyticalModelSelector amSelector = new AnalyticalModelSelector(analyticalCurve);
        amSelector.CurveSelector = AnalyticalCurveSelector.WholeCurve;
        Reference curveReference = analyticalModelFrame.GetReference(amSelector);

        // test the stable reference to the start point of the curve
        amSelector.CurveSelector = AnalyticalCurveSelector.StartPoint;
        Reference startPointReference = analyticalModelFrame.GetReference(amSelector);

        // test the stable reference to the start point of the curve
        amSelector.CurveSelector = AnalyticalCurveSelector.EndPoint;
        Reference endPointReference = analyticalModelFrame.GetReference(amSelector);
    }

    return true;
}
```

29.2.1.1 GetPoint()

If the AnalyticalModel can be expressed by a single point (i.e. Structural Footing), this method will return that point. Otherwise, it will throw an Autodesk.Revit.Exceptions.InapplicableDataException. The IsSinglePoint() method can be used to determine if the AnalyticalModel can be expressed by a single point.

The following example demonstrates how to get the analytical location for a structural footing.

Code Region 29-11: Getting the location for a structural footing

```
// retrieve and iterate current selected element
UIDocument uidoc = commandData.Application.ActiveUIDocument;
ElementSet selection = uidoc.Selection.Elements;
foreach (Element e in selection)
```

```
{
    // if the element is structural footing
    FamilyInstance familyInst = e as FamilyInstance;
    if (null != familyInst && familyInst.StructuralType == StructuralType.Footing)
    {
        AnalyticalModel model = familyInst.GetAnalyticalModel();
        // structural footing should be expressable as a single point
        if (model.IsSinglePoint() == true)
        {
            XYZ analyticalLocationPoint = model.GetPoint();
        }
    }
}
```

29.2.1.2 GetCurve()

If the AnalyticalModel can be expressed by a single curve (i.e. Structural Column or Structural Framing), this method will return that Curve. Otherwise, it will throw an Autodesk.Revit.Exceptions.InapplicableDataException. The IsSingleCurve() method can be used to determine if the AnalyticalModel can be expressed by a single curve.

Code Region 29-12: Getting the curve for a structural column

```
public void GetColumnCurve(FamilyInstance familyInst)
{
    // get AnalyticalModel from structural column
    if (familyInst.StructuralType == StructuralType.Column)
    {
        AnalyticalModel modelColumn = familyInst.GetAnalyticalModel();
        // column should be represented by a single curve
        if (modelColumn.IsSingleCurve() == true)
        {
            Curve columnCurve = modelColumn.GetCurve();
        }
    }
}
```

29.2.1.3 GetCurves()

This method is required to get the Curves of an AnalyticalModel defined by more than one curve (i.e. Structural Wall), but can be used in all cases. If the AnalyticalModel can be expressed by a single curve, this method will return a List containing only one Curve. If the AnalyticalModel can be expressed by a single point, this method will return a Curve of almost 0 length containing the point. This method takes an AnalyticalCurveType enum as a parameter. The possible values are:

- **RawCurves** - Base Analytical Model curves generated
- **ActiveCurves** - Curves displayed on screen (not including Rigid Links)
- **ApproximatedCurves** - Curves approximated using linear segments

The following values related to Rigid Links are also available. See the Rigid Links section later in this chapter for more information.

- **RigidLinkHead** - Rigid Link at end 0 (head) of the Beam
- **RigidLinkTail** - Rigid Link at end 1 (tail) of the Beam

- **AllRigidLinks** - All Rigid Link curves. Rigid Link at end 0 (head) will be in the first entry. Rigid Link at end 1 (tail) will be in the last entry.

The following example demonstrates the use of AnalyticalModel for structural walls.

Code Region 29-13: Getting the curves for a structural wall

```
// retrieve and iterate current selected element
UIDocument uidoc = commandData.Application.ActiveUIDocument;
ElementSet selection = uidoc.Selection.Elements;
foreach (Element e in selection)
{
    Wall aWall = e as Wall;
    if (null != aWall)
    {
        // get AnalyticalModelWall from Structural Wall
        AnalyticalModel modelWall =
            aWall.GetAnalyticalModel() as AnalyticalModel;
        if (null == modelWall)
        {
            // Architecture wall doesn't have analytical model
            continue;
        }
        // get analytical information
        int modelCurveNum = modelWall.GetCurves(AnalyticalCurveType.ActiveCurves).Count;
        IList<AnalyticalModelSupport> supportList = new List<AnalyticalModelSupport>();
        supportList = modelWall.GetAnalyticalModelSupports();
        int supportInfoNum = supportList.Count;
    }
}
```

29.2.2 Parameter Information

AnalyticalModel provides access to Parameter information such as rigid links, projection, and approximation.

29.2.2.1 Rigid Links

A rigid link connects the analytical model of a beam to the analytical model of a column. Use the CanHaveRigidLinks()method and the AnalyticalModel.RigidLinksOption property to determine if rigid links are applicable to the AnalyticalModel. Additionally, you can use HasRigidLinksWith() to determine whether the AnalyticalModel has rigid links with a specific element.

End links can be retrieved by specifying the AnalyticalCurveType options RigidLinkHead and RigidLinkTail with the AnalyticalModel.GetCurves() Method. Or, use AnalyticalModel.GetRigidLink() with an AnalyticalModelSelector object.

One difference between the AnalyticalModel methods GetCurve() and GetCurves() for a structural beam is that GetCurves() includes the single Curve as well as the structural beam RigidLink Curve if it is present. Pass the AnalyticalCurveType.RigidLinkHead or AnalyticalCurveType.RigidLinkTail enum values to the GetCurves() method to get the RigidLink at the head or tail of a beam.

Although you cannot create a rigid link directly since it is not an independent object, you can create it using the RigidLinksOption property on the analytical model for beams and/or columns. The rigid link option for a beam overrides the option for the column.

For Structural Beams, the RigidLinksOption property can have the following values:

- AnalyticalRigidLinksOption.Enabled - Rigid links will be formed
- AnalyticalRigidLinksOption.Disabled - Rigid links will not be formed
- AnalyticalRigidLinksOption.FromColumn - Rigid links may be formed, depending on the corresponding structural column's value.

For Structural Columns, the RigidLinksOption property can have the following values:

- AnalyticalRigidLinksOption.Enabled - Rigid links will be formed, unless corresponding structural beam's setting overrides.
- AnalyticalRigidLinksOption.Disabled - Rigid links will not be formed, unless corresponding structural beam's setting overrides.

Note that in addition to the correct values being set, the elements must also overlap for the rigid link to be created.

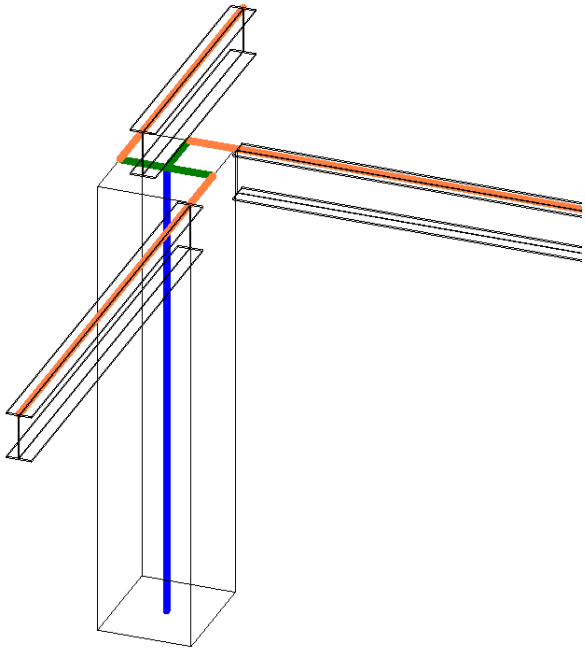


Figure 158: RigidLink

29.2.2.2**Analytical Projection**

The horizontal and vertical projection parameters for an AnalyticalModel can be retrieved and set using the GetAnalyticalProjectionType() and SetAnalyticalProjectionType() methods. These methods take an AnalyticalDirection as an input. The SetAnalyticalProjectionDatumPlane() can also be used to set the specified projection to a specific Datum Plane and GetAnalyticalProjectionDatumPlane() will retrieve the analytical projection when set to a Datum Plane.

29.2.2.3**Approximation**

When an AnalyticalModel is defined by a curve rather than a straight line (i.e. for a curved beam), an approximation (comprised of straight lines) may be preferable. AnalyticalModel has several methods related to curve approximation. If CanApproximate() returns true, use the Approximate() method to switch between non-approximated (curved) Analytical Models and approximated (made up of lines only) Analytical Models. After switching to approximated, use GetCurves() to get the lines of the approximated curve.

The approximation will be based on the approximation deviation value (GetApproximationDeviation()) and the Use Hard-points parameter (UsesHardPoints()). These values have corresponding Set methods as well. The approximation deviation limits the distance between the smooth curve and a line segment generated by an approximation. Hard-points are the locations on the curved beam where other structural elements are touching. When you set this parameter to true, it forces the segmented analytical model to have nodal points at the ends of the members attached to the curved beam

29.2.2.4 **AnalyzeAs**

The Analyze As parameter can be retrieved and set via the AnalyticalModel. This parameter indicates to analysis programs how an element should be analyzed, or whether the element is NotForAnalysis. Since the AnalyzeAs enum used by GetAnalyzeAs() and SetAnalyzeAs() contains enum values used for different types of elements, not all values are applicable for all analytical models. Use the IsAnalyzeAsValid() method to determine if a particular value is applicable for the analytical model.

29.2.3 **Manual Adjustment**

The geometry of the structural member analytical model may also be adjusted in relation to those elements to which it joins (assuming the SupportsManualAdjustment() method returns true). Use the AnalyticalModel.ManuallyAdjust() method to adjust the analytical model in relation to another element. AnalyticalModel also provides methods to determine if the analytical model has been manually adjusted and to reset it back to its original location, relative to its corresponding physical model. Additionally, the GetManualAdjustmentMatchedElements() method retrieves a collection of element Ids against which the Analytical Model has been adjusted.

29.2.4 **Analytical Offset**

Another way to adjust an analytical model is to use an offset. Setting the analytical offset is different than manually adjusting the analytical model. The analytical offset is a basic offset applied to the entire analytical model and is independent of any other elements. AnalyticalModel has methods to get and set the analytical offset as well as to determine if the analytical offset can be changed (CanSetAnalyticalOffset()).

29.2.5 **AnalyticalModelSupport**

AnalyticalModel provides the method IsElementFullySupported() to determine if the analytical model is fully supported. For additional information about what is supporting the analytical model, the GetAnalyticalModelSupports() method retrieves a collection of AnalyticalModelSupport objects that provide information on how the element is supported by other structural elements, including the priority of each support (if multiple elements provide support) and the point, curve or face that provides the support. The following examples illustrate how to use the AnalyticalModelSupport objects in different conditions.

29.2.5.1 **Floor and StructuralBeam Support Information**

When drawing a slab in sketch mode, select Pick Supports on the design bar. As shown in the following picture, a slab has three support beams. By iterating the slab's collection of AnalyticalModelSupports, you get the three Beams as well as the CurveSupport AnalyticalSupportType.

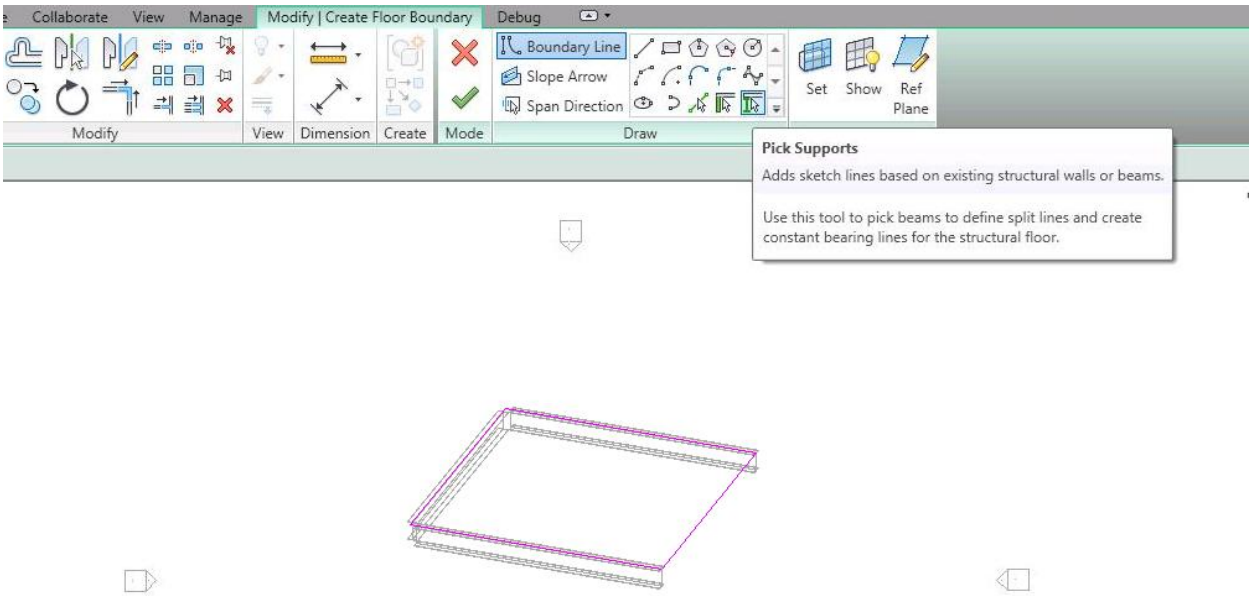


Figure 159: Floor and StructuralBeam Support Information

29.2.5.2**Floor and Wall Support Information**

After drawing a slab by picking walls as the support, you cannot get Walls from the Floor's AnalyticalModelSupport collection. Instead, Floor is available in the Wall's collection of AnalyticalModelSupports.

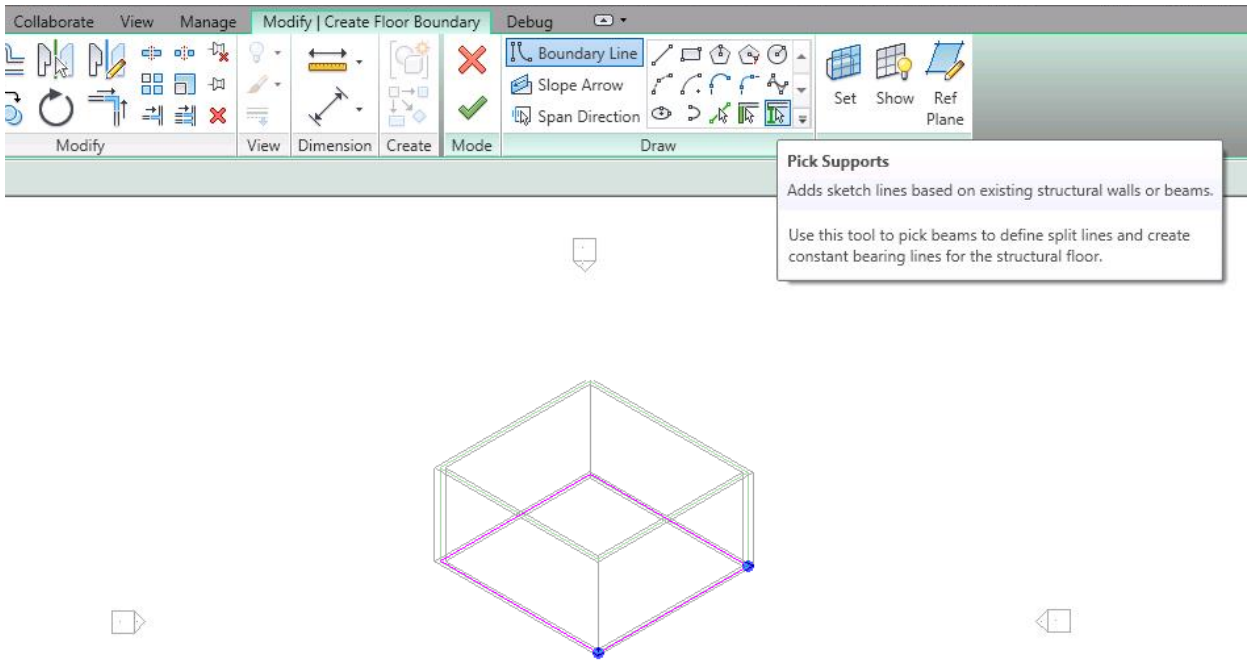


Figure 160: Floor and Wall Support Information

29.2.5.3 Structural Column, Beam and Brace Support Information

In the following picture, the horizontal beam has three PointSupports--two structural columns and one structural brace. The brace has three PointSupports-- two structural columns and one structural beam. Neither column has a support Element.

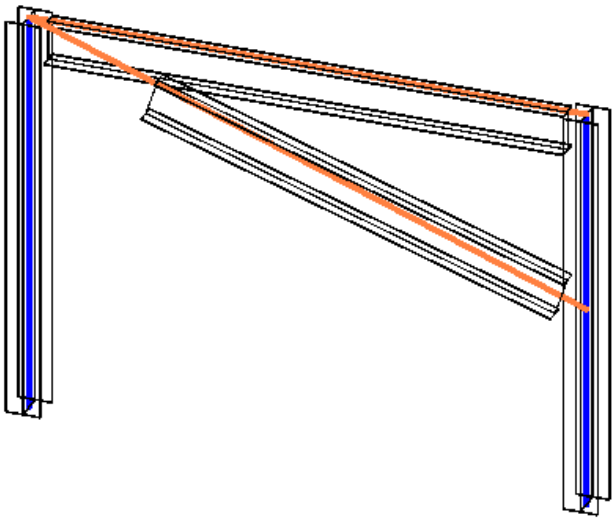


Figure 161: StructuralElements Support Information

29.2.5.4 BeamSystem and Wall Support Information

Though you can pick walls as supports when you draw a BeamSystem, its support information is not directly available because the BeamSystem does not have the AnalyticalModel property. The solution is to call the GetAllBeams() method, to retrieve the AnalyticalModelSupport collection for the Beams.

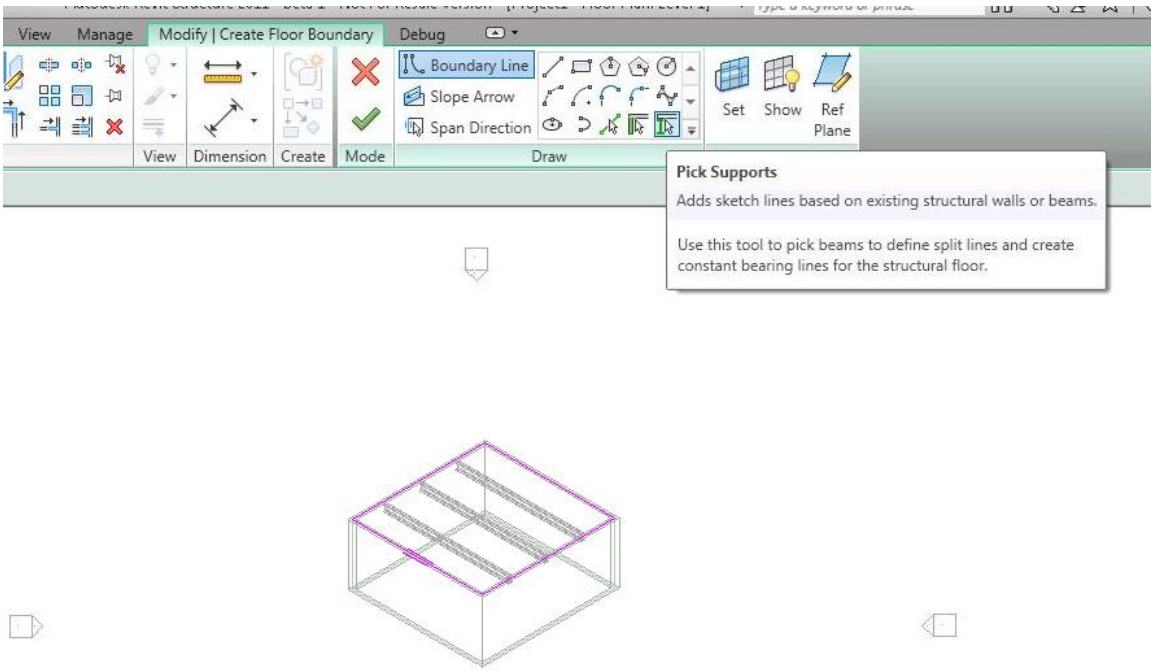


Figure 162: BeamSystem and Wall Support Information

29.2.5.5 ContFooting and Wall Support Information

For a Wall with a continuous Foundation, the Wall has a CurveSupport with ContFooting available. The support curves are available using the AnalyticalModel.GetCurves() method. In the following sample, there are two Arcs in the Curve.

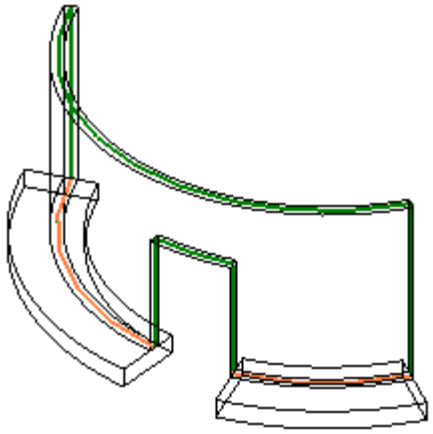


Figure 163: ContFooting and Wall Support Information

29.2.5.6 Isolated Foundation and StructuralColumn Support Information

Structural columns can have an isolated footing as a PointSupport. In this condition, the footing can move with the supported structural column. The ElementId of the FamilyInstance with the OST_StructuralFoundation category is available from the AnalyticalModelSupport.GetSupportingElement() method. Generally, the support point is the bottom point of the curve retrieved from the AnalyticalModel.GetCurve() method. It is also available after you get the isolated footing FamilyInstance and the AnalyticalModel Point available from the GetPoint() method.

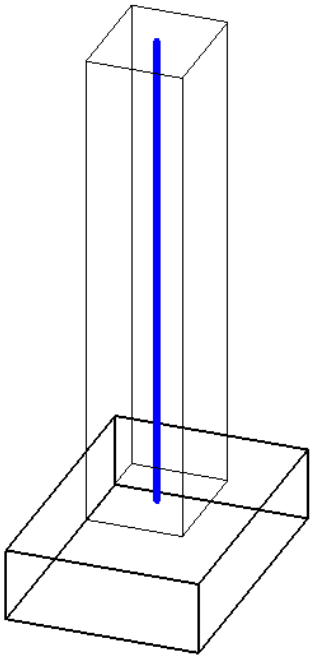


Figure 164: Isolated Foundation (FamilyInstance) and Structural Column Support Information

29.3 Loads

The following sections identify load settings and discuss load limitation guidelines.

29.3.1 Load Settings

All functionality on the Setting dialog box Load Cases and Load Combinations tabs can be accessed by the API.

The following properties are available from the corresponding LoadCase BuiltInParameter:

Property	BuiltInParameter
Case Number	LOAD_CASE _NUMBER
Nature	LOAD_CASE_NATURE
Category	LOAD_CASE_CATEGORY

Table 59 Load Case Properties and Parameters

The LOAD_CASE_CATEGORY parameter returns an ElementId. The following table identifies the mapping between Category and ElementId Value.

Load Case Category	BuiltInCategory
Dead Loads	OST_LoadCasesDead
Live Loads	OST_LoadCasesLive
Wind Loads	OST_LoadCasesWind
Snow Loads	OST_LoadCasesSnow
Roof Live Loads	OST_LoadCasesRoofLive

Load Case Category	BuiltInCategory
Accidental Loads	OST_LoadCasesAccidental
Temperature Loads	OST_LoadCasesTemperature
Seismic Loads	OST_LoadCasesSeismic

Table 60: Load Case Category

The following Document creation methods create corresponding subclasses:

- NewLoadUsage() creates LoadUsage
- NewLoadNature() creates LoadNature
- NewLoadCase() creates LoadCase
- NewLoadCombination() creates LoadCombination.
- NewPointLoad() creates PointLoad (an overload allows you to specify the load host element as a Reference). You can optionally specify a load type and sketch plane.
- NewLineLoad() creates LineLoad (overloads allow you to specify the host element as a Reference or an Element). You can optionally specify a load type and sketch plane.
- NewAreaLoad() creates AreaLoad (overloads allow you to specify the host element as a Reference or an Element). You can optionally specify a load type and sketch plane.

Because they are all Element subclasses, they can be deleted using Document.Delete().

Code Region 29-14: NewLoadCombination()

```
public LoadCombination NewLoadCombination(string name,
   int typeInd, int stateInd, double[] factors, LoadCaseArray cases,
  LoadCombinationArray combinations, LoadUsageArray usages);
```

For the NewLoadCombination() method, the factor size must be equal to or greater than the sum size of cases and combinations. For example,

- If cases.Size is M, combinations.Size is N,
- Factors.Size should not be less than M+N. The first M factors map to M cases in order, and the last N factors map to N combinations.
- Check that LoadCombination does not include itself.

There is no Duplicate() method in the LoadCase and LoadNature classes. To implement this functionality, you must first create a new LoadCase (or LoadNature) object using the NewLoadCase() (or NewLoadNature()) method, and then copy the corresponding properties and parameters from an existing LoadCase (or LoadNature).

29.4 Analysis Link

With Revit Structure, an analytical model is automatically generated as you create the physical model. The analytical model is linked to structural analysis applications and the physical model is automatically updated from the results through the Revit Structure API. Some third-party software developers already provide bi-directional links to their structural analysis applications. These include the following:

- ADAPT-Builder Suite from ADAPT Corporation (www.adaptsoft.com/revitstructure/)
- Fastrak and S-Frame from CSC (www.cscworld.com)
- ETABS from CSI (www.csiberkeley.com/)

- RFEM from Dlubal (www.dlubal.com/FEA-Software-RFEM-integrates-with-Revit-Structure.aspx)
- Advance Design, VisualDesign, Arche, Effel and SuperSTRESS from GRAITEC (www.graitec.com/En/revit.asp)
- Scia Engineer from Nemetschek (www.scia-online.com/en/revit.html)
- GSA from Oasys Software (Arup) (www.oasys-software.com/products)
- ProDESK from Prokon Software Consultants (www.prokon.com/)
- RAM Structural System from Bentley (www.bentley.com/en-US/Products/RAM+Structural+System/)
- RISA-3D and RISAFloor from RISA Technologies (www.risatech.com/partner/revit_structure.asp)
- SOFiSTiK Structural Desktop Suite from SOFiSTiK (www.sofistik.com/revit-to-sofistik)
- SPACE GASS from SPACE GASS (www.spacegass.com/index.asp?resend=/revit.asp)
- Revit Structure STAAD.Pro interface from Structural Integrators (structuralintegrators.com/products/si_xchange.php)

The key to linking Revit Structure to other analysis applications is to set up the mapping relationship between the objects in different object models. That means the difficulty and level of the integration depends on the similarity between the two object models.

For example, during the product design process, design a table with at least the first two columns in the object mapping in the following table: one for Revit Structure API and the other for the structural analysis application, shown as follows:

Revit Structural API	Analysis Application	Import to Revit
StructuralColumn	Column	NewStructuralColumn
Property:		
…		
Location		Read-only;
Parameter:		
…		
Analyze as		Editable;
AnalyticalModel:		
…		
Profile		Read-only;
RigidLink		Read-only;
…		
Material:		
…		

Table 61: Revit and Analysis Application Object Mapping

30 Revit MEP

The Revit MEP portion of the Revit API provides read and write access to HVAC and Piping data in a Revit model including:

- Traversing ducts, pipes, fittings, and connectors in a system
- Adding, removing, and changing ducts, pipes, and other equipment
- Getting and setting system properties
- Determining if the system is well-connected

30.1 MEP Element Creation

Elements related to duct and pipe systems can be created using the following methods available in the Autodesk.Revit.Creation.Document class:

- NewDuct
- NewFlexDuct
- NewPipe
- NewFlexPipe
- NewMechanicalSystem
- NewPipingSystem
- NewCrossFitting
- NewElbowFitting
- NewTakeoffFitting
- NewTeeFitting
- NewTransitionFitting
- NewUnionFitting

30.1.1 Creating Pipes and Ducts

There are 3 ways to create new ducts, flex ducts, pipes and flex pipes. They can be created between two points, between two connectors, or between a point and a connector.

The following code creates a new pipe between two points. New flex pipes, ducts and flex ducts can all be created similarly.

Code Region 30-1: Creating a new Pipe

```
public Pipe CreateNewPipe(Document document)
{
    // find a pipe type

    FilteredElementCollector collector = new FilteredElementCollector(document);
    collector.OfClass(typeof(PipeType));
    PipeType pipeType = collector.FirstElement() as PipeType;

    Pipe pipe = null;
    if (null != pipeType)
    {
        // create pipe between 2 points
```

```
        XYZ p1 = new XYZ(0, 0, 0);
        XYZ p2 = new XYZ(10, 0, 0);

        pipe = document.Create.NewPipe(p1, p2, pipeType);
    }

    return pipe;
}
```

After creating a pipe, you might want to change the diameter. The Diameter property of Pipe is read-only. To change the diameter, get the RBS_PIPE_DIAMETER_PARAM built-in parameter.

Code Region 30-2: Changing pipe diameter

```
public void ChangePipeSize(Pipe pipe)
{
    Parameter parameter = pipe.get_Parameter(BuiltInParameter.RBS_PIPE_DIAMETER_PARAM);

    string message = "Pipe diameter: " + parameter.AsValueString();

    parameter.Set(0.5); // set to 6"

    message += "\nPipe diameter after set: " + parameter.AsValueString();

    MessageBox.Show(message, "Revit");
}
```

Another common way to create a new duct or pipe is between two existing connectors, as the following example demonstrates. In this example, it is assumed that 2 elements with connectors have been selected in Revit MEP, one being a piece of mechanical equipment and the other a duct fitting with a connector that lines up with the SupplyAir connector on the equipment.

Code Region 30-3: Adding a duct between two connectors

```
public Duct CreateDuctBetweenConnectors(UIDocument uiDocument)
{
    // prior to running this example
    // select some mechanical equipment with a supply air connector
    // and an elbow duct fitting with a connector in line with that connector
    Connector connector1 = null, connector2 = null;
    ConnectorSetIterator csi = null;
    ElementSet selection = uiDocument.Selection.Elements;
    // First find the selected equipment and get the correct connector
    foreach (Element e in selection)
    {
        if (e is FamilyInstance)
        {
            FamilyInstance fi = e as FamilyInstance;
            Family family = fi.Symbol.Family;
            if (family.FamilyCategory.Name == "Mechanical Equipment")
```

```
            {
                csi = fi.MEPModel.ConnectorManager.Connectors.ForwardIterator();
                while (csi.MoveNext())
                {
                    Connector conn = csi.Current as Connector;
                    if (conn.Direction == FlowDirectionType.Out &&
                          conn.DuctSystemType == DuctSystemType.SupplyAir)
                    {
                         connector1 = conn;
                         break;
                    }
                }
            }
        }
    }
    // next find the second selected item to connect to
    foreach(Element e in selection)
    {
        if (e is FamilyInstance)
        {
            FamilyInstance fi = e as FamilyInstance;
            Family family = fi.Symbol.Family;
            if (family.FamilyCategory.Name != "Mechanical Equipment")
            {
                csi = fi.MEPModel.ConnectorManager.Connectors.ForwardIterator();
                while (csi.MoveNext())
                {
                    if (null == connector2)
                    {
                        Connector conn = csi.Current as Connector;

                        // make sure to choose the connector in line with the first connector
                        if (Math.Abs(conn.Origin.Y - connector1.Origin.Y) < 0.001)
                        {
                            connector2 = conn;
                            break;
                        }
                    }
                }
            }
        }
    }
    Duct duct = null;
    if (null != connector1 && null != connector2)
    {
        // find a duct type
        FilteredElementCollector collector =
            new FilteredElementCollector(uiDocument.Document);
```

```
        collector.OfClass(typeof(DuctType));

        // Use Linq query to make sure it is one of the rectangular duct types
        var query = from element in collector
                    where element.Name.Contains("Mitered Elbows") == true
                    select element;
        // use extension methods to get first duct type
        DuctType ductType = collector.Cast<DuctType>().First<DuctType>();

        if (null != ductType)
        {
            duct = uiDocument.Document.Create.NewDuct(connector1, connector2, ductType);
        }
    }

    return duct;
}
```

Below is the result of running this code after selecting a VAV Unit – Parallel Fan Powered and a rectangular elbow duct fitting.

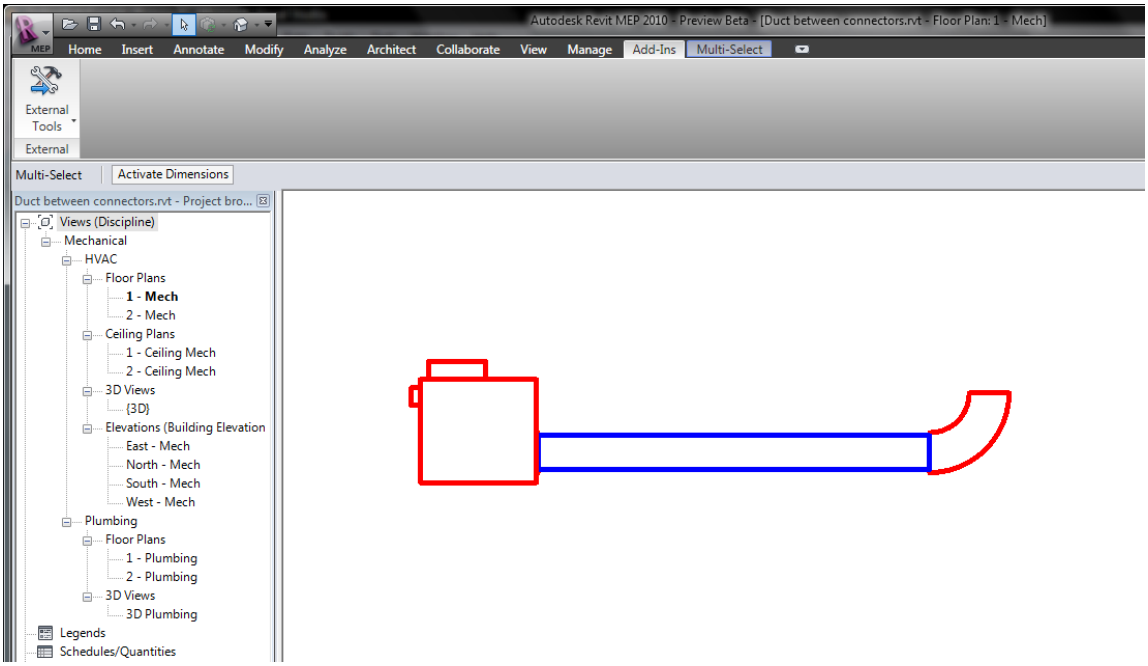


Figure 165: New duct added between selected elements

30.1.2 Creating a new system

New piping and mechanical systems can be created using the Revit API. NewPipingSystem() and NewMechanicalSystem() both take a Connector that is the base equipment connector, such as a hot water heater for a piping system, or a fan for a mechanical system. They also take a ConnectorSet of connectors that will be added to the system, such as faucets on sinks in a piping system. The last piece of information required to create a new system is either a PipeSystemType for NewPipingSystem() or a DuctSystemType for NewMechanicalSystem().

In the following sample, a new SupplyAir duct system is created from a selected piece of mechanical equipment (such as a fan) and all selected Air Terminals.

Code Region 30-4: Creating a new mechanical system

```
// create a connector set for new mechanical system
ConnectorSet connectorSet = new ConnectorSet();
// Base equipment connector
Connector baseConnector = null;

// Select a Parallel Fan Powered VAV and some Supply Diffusers
// prior to running this example
ConnectorSetIterator csi = null;
ElementSet selection = document.Selection.Elements;
foreach (Element e in selection)
{
    if (e is FamilyInstance)
    {
        FamilyInstance fi = e as FamilyInstance;
        Family family = fi.Symbol.Family;
        // Assume the selected Mechanical Equipment is the base equipment for new system
        if (family.FamilyCategory.Name == "Mechanical Equipment")
        {
            //Find the "Out" and "SupplyAir" connector on the base equipment
            if (null != fi.MEPModel)
            {
                csi = fi.MEPModel.ConnectorManager.Connectors.ForwardIterator();
                while (csi.MoveNext())
                {
                    Connector conn = csi.Current as Connector;
                    if (conn.Direction == FlowDirectionType.Out &&
                        conn.DuctSystemType == DuctSystemType.SupplyAir)
                    {
                        baseConnector = conn;
                        break;
                    }
                }
            }
        }
        else if (family.FamilyCategory.Name == "Air Terminals")
        {
            // add selected Air Terminals to connector set for new mechanical system
            csi = fi.MEPModel.ConnectorManager.Connectors.ForwardIterator();
            csi.MoveNext();
            connectorSet.Insert(csi.Current as Connector);
        }
    }
}
```

```
MechanicalSystem mechanicalSys = null;
if (null != baseConnector && connectorSet.Size > 0)
{
    // create a new SupplyAir mechanical system
    mechanicalSys = document.Create.NewMechanicalSystem(baseConnector, connectorSet,
                                                        DuctSystemType.SupplyAir);
}
```

30.2 Connectors

As shown in the previous section, new pipes and ducts can be created between two connectors. Connectors are associated with a domain – ducts, piping or electrical – which is obtained from the Domain property of a Connector. Connectors are present on mechanical equipment as well as on ducts and pipes.

To traverse a system, you can examine connectors on the base equipment of the system and determine what is attached to the connector by checking the IsConnected property and then the AllRefs property. When looking for a physical connection, it is important to check the ConnectionType of the connector. There are both physical and logical connectors in Revit, but only the physical connectors are visible in the application. The following imagine shows the two types of physical connectors – end connections and curve connectors.

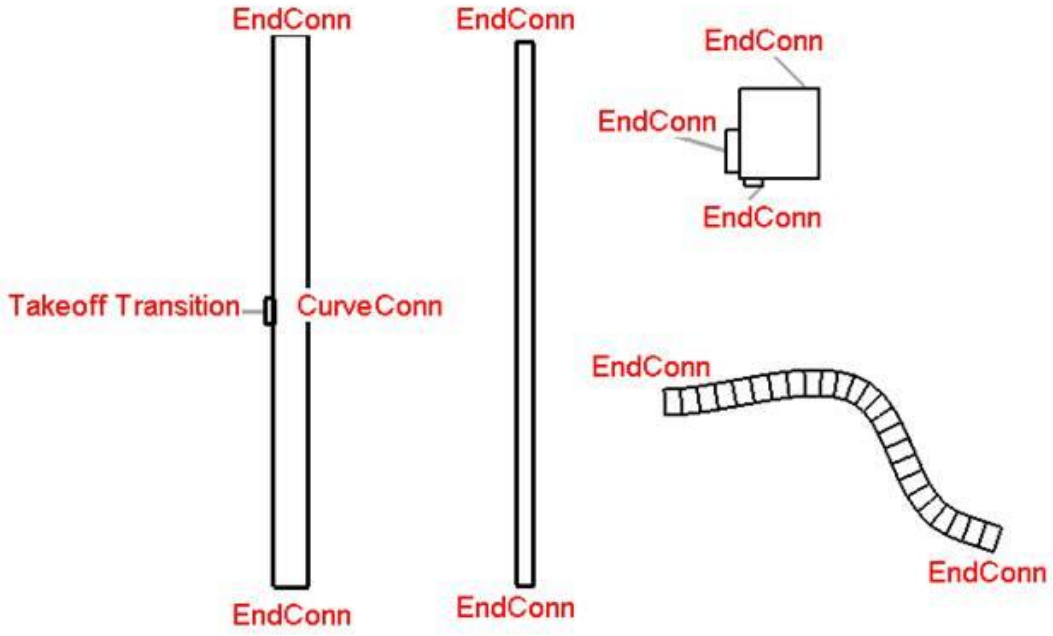


Figure 166: Physical connectors

The following sample shows how to determine the owner of a connector, and what, if anything it attaches to, along with the connection type.

Code Region 30-5: Determine what is attached to a connector

```
public void GetElementAtConnector(Connector connector)
{
```

```
    MEPSystem mepSystem = connector.MEPSystem;
    if (null != mepSystem)
    {
        string message = "Connector is owned by: " + connector.Owner.Name;

        if (connector.IsConnected == true)
        {
            ConnectorSet connectorSet = connector.AllRefs;
            ConnectorSetIterator csi = connectorSet.ForwardIterator();
            while (csi.MoveNext())
            {
                Connector connected = csi.Current as Connector;
                if (null != connected)
                {
                    // look for physical connections
                    if (connected.ConnectorType == ConnectorType.EndConn ||
                        connected.ConnectorType == ConnectorType.CurveConn ||
                        connected.ConnectorType == ConnectorType.PhysicalConn)
                    {
                        message += "\nConnector is connected to: " + connected.Owner.Name;
                        message += "\nConnection type is: " + connected.ConnectorType;
                    }
                }
            }
        }
        else
        {
            message += "\nConnector is not connected to anything.";
        }

        MessageBox.Show(message, "Revit");
    }
}
```

The following dialog box is the result of running this code example on the connector from a piece of mechanical equipment.

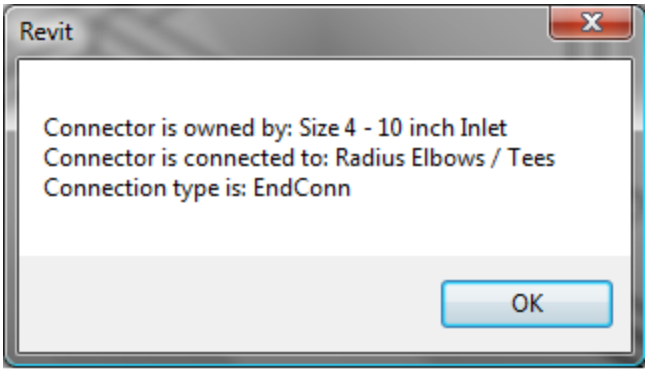


Figure 167: Connector Information

30.3 Family Creation

When creating mechanical equipment in a Revit family document, you will need to add connectors to allow the equipment to connect to a system. Duct, electrical and pipe connectors can all be added similarly, using a reference plane where the connector will be placed and a system type for the connector.

The functions provided in FamilyItemFactory for MEP Connectors are:

- NewDuctConnector
- NewElectricalConnector
- NewPipeConnector

The following code demonstrates how to add two pipe connectors to faces on an extrusion and set some properties on them.

Code Region 30-6: Adding a pipe connector

```
public void CreatePipeConnectors(UIDocument uiDocument, Extrusion extrusion)
{
    // get the faces of the extrusion
    Options geoOptions = uiDocument.Document.Application.Create.NewGeometryOptions();
    geoOptions.View = uiDocument.Document.ActiveView;
    geoOptions.ComputeReferences = true;

    List<PlanarFace> planarFaces = new List<PlanarFace>();
    Autodesk.Revit.DB.GeometryElement geoElement = extrusion.get_Geometry(geoOptions);
    foreach (GeometryObject geoObject in geoElement.Objects)
    {
        Solid geoSolid = geoObject as Solid;
        if (null != geoSolid)
        {
            foreach (Face geoFace in geoSolid.Faces)
            {
                if (geoFace is PlanarFace)
                {
                    planarFaces.Add(geoFace as PlanarFace);
                }
            }
        }
    }

    if (planarFaces.Count > 1)
    {
        // Create the Supply Hydronic pipe connector
        PipeConnector connReturn =
               uiDocument.Document.FamilyCreate.NewPipeConnector(planarFaces[1].Reference,
                                                                 PipeSystemType.ReturnHydronic);
        Parameter param = connSupply.get_Parameter(BuiltInParameter.CONNECTOR_RADIUS);
        param.Set(1.0); // 1' radius
        param = connSupply.get_Parameter(BuiltInParameter.RBS_PIPE_FLOW_DIRECTION_PARAM);
```

```
        param.Set(2);

        // Create the Return Hydronic pipe connector
        PipeConnector connReturn =
            uiDocument.Document.FamilyCreate.NewPipeConnector(planarFaces[1].Reference,
                                                              PipeSystemType.ReturnHydronic);

        param = connReturn.get_Parameter(BuiltInParameter.CONNECTOR_RADIUS);
        param.Set(0.5); // 6" radius
        param = connReturn.get_Parameter(BuiltInParameter.RBS_PIPE_FLOW_DIRECTION_PARAM);
        param.Set(1);
    }
}
```

The following illustrates the result of running this example using in a new family document created using a Mechanical Equipment template and passing in an extrusion 2'x2'x1'. Note that the connectors are placed at the centroid of the planar faces.

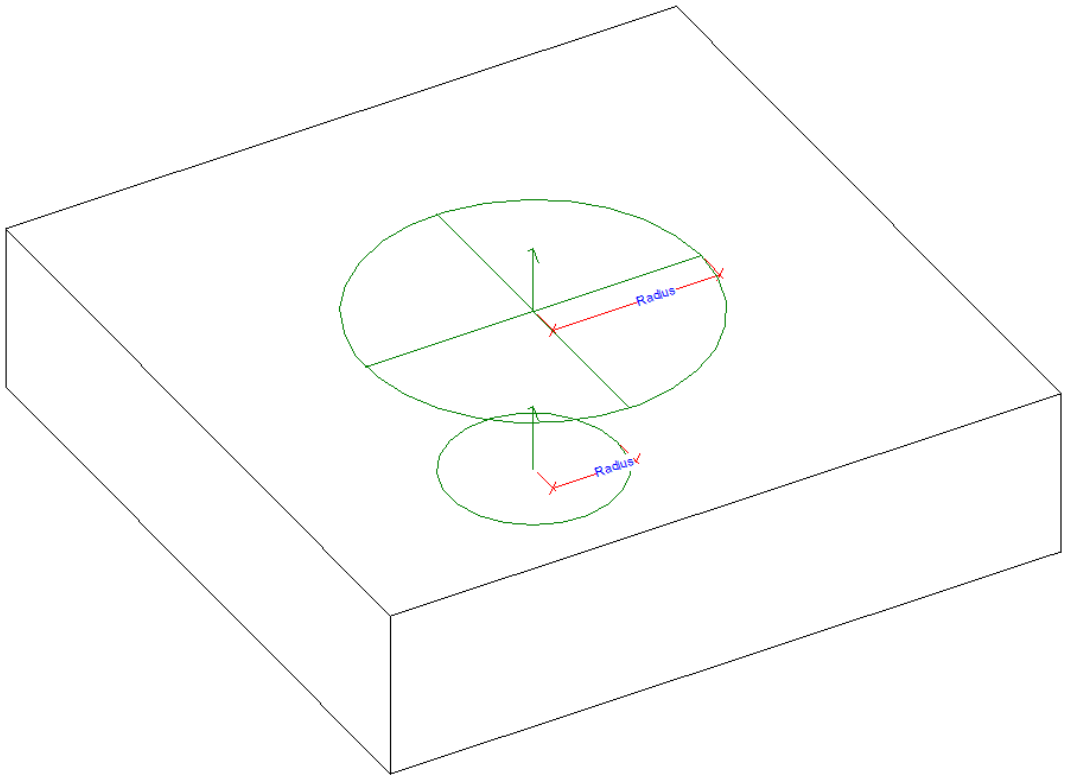


Figure 168: Two connectors created on an extrusion

A. Glossary

Array

Arrays hold a series of data elements, usually of the same size and data type. Individual elements are accessed by their position in the array. The position is provided by an index, which is also called a subscript. The index usually uses a consecutive range of integers, but the index can have any ordinal set of values.

BIM

Building Information Modeling is the creation and use of coordinated, internally consistent, computable information about a building project in design and construction. In a BIM application the graphics are derived from the information and are not the original information itself like in general CAD applications.

Class

In object-oriented programming (OOP), classes are used to group related Properties (variables) and Methods (functions) together. A typical class describes how those methods operate upon and manipulate the properties. Classes can be standalone or inherited from other classes. In the latter, a class from which others are derived is usually referred to as a Base Class.

Events

Events are messages or functions that are called when an event occurs within an application. For example when a model is saved or opened.

Iterator

An iterator is an object that allows a programmer to traverse through all elements in a collection (an array, a set, etc.), regardless of its specific implementation.

Method

A method is a function or procedure that belongs to a class and operates or accesses the class data members. In procedural programming, this is called a function.

Namespace

A namespace is an organizational unit used to group similar and/or functionally related classes together.

Overloading

Method overloading is when different methods (functions) of the same name are invoked with different types and/or numbers of parameters passed.

Properties

Properties are data members of a class accessible to the class user. In procedural programming this is called a variable. Some properties are read only (support Get() method) and some are modifiable (support Set() method).

Revit Families

A Family is a collection of objects called types. A family groups elements with a common set of parameters, identical use, and similar graphical representation. Different types in a family can have different values of some or all parameters, but the set of parameters - their names and their meaning - are the same.

Revit Parameters

There are a number of Revit parameter types.

- Shared Parameters can be thought of as user-defined variables.
- System Parameters are variables that are hard-coded in Revit.
- Family parameters are variables that are defined when a family is created or modified.

Revit Types

A Type is a member of a Family. Each Type has specific parameters that are constant for all instances of the Type that exist in your model; these are called Type Properties. Types have other parameters called Instance parameters, which can vary in your model.

Sets

A set is a collection (container) of values without a particular order and no repeated values. It corresponds with the mathematical concept of set except for the restriction that it has to be finite.

Element ID

Each element has a corresponding ID. It is identified by an integer value. It provides a way of uniquely identifying an Element within an Autodesk Revit project. It is only unique for one project, but not unique across separate Autodesk Revit projects.

Element UID

Each element has a corresponding UID. It is a string identifier that is universally unique. That means it is unique across separate Autodesk Revit projects.

B. FAQ

General Questions

Q: How do I reference an element in Revit?

A: Each element has an ID. The ID that is unique in the model is used to make sure that you are referring to the same element across multiple sessions of Revit.

Q: Can a model only use one shared parameter file?

A: Shared parameter files are used to hold bits of information about the parameter. The most important piece of information is the GUID (Globally Unique Identifier) that is used to insure the uniqueness of a parameter in a single file and across multiple models.

Revit can work with multiple shared parameter files but you can only read parameters from one file at a time. It is then up to you to choose the same shared parameter file for all models or a different one for each model.

In addition, your API application should avoid interfering with the user's parameter file. Ship your application with its own parameter file containing your parameters. To load the parameter(s) into a Revit file:

- The application must remember the user parameter file name.
- Switch to the application's parameter file and load the parameter.
- Then switch back to the user's file.

Q: Do I need to distribute the shared parameters file with the model so other programs can use the shared parameters?

A: No. The shared parameters file is only used to load shared parameters. After they are loaded the file is no longer needed for that model.

Q: Are shared parameter values copied when the corresponding element is copied?

A: Yes. If you have a shared parameter that holds the unique ID for an element in your database, append the Revit element Unique ID or add another shared parameter with the Revit element unique ID. Do this so that you can check it and make sure you are working with the original element ID and not a copy.

Q: Are element Unique IDs (UID) universally unique and can they ever change?

A: The element UIDs are universally unique, but element IDs are only unique within a model. For example, if you copy a wall from one Revit project to another one, the UID of the wall is certain to change to maintain universal uniqueness, but the ID of the wall may not change.

Q: Revit takes a long time to update when my application sends data back to the model. What do I need to do to speed it up?

A: You can try using the SuspendUpdating command. See the FrameBuilder example in the SDK.

Q: What do I do if I want to add shared parameters to elements that do not have the ability to have shared parameters bound to them? For example, Grids or Materials.

A: If an element type does not have the ability to add shared parameters, you need to add a project parameter. This does make it a bit more complicated when it is time to access the shared parameter associated with the element because it does not show up as part of the element's parameter list. By using tricks like making the project shared parameter a string and including the element ID in the shared parameter you can associate the data with an element by first parsing the string.

Q: How do I access the saved models and content BMP?

A: The Preview.dll is a shell plug-in which is an object that implements the IExtractImage interface. IExtractImage is an interface used by the Windows Shell Folders to extract the images for a known file type.

For more information, review the information at http://windowssdk.msdn.microsoft.com/en-us/library/ms645964.aspx

CRevitPreviewExtractor implements standard API functions:

```
STDMETHOD(GetLocation)(LPWSTR pszPathBuffer,
                       DWORD cchMax,
                       DWORD *pdwPriority,
                       const SIZE *prgSize,
                       DWORD dwRecClrDepth,
                       DWORD *pdwFlags);
STDMETHOD(Extract)(HBITMAP*);
```

It registers itself in the registry.

Revit Structure Questions

Q: Sometimes the default end releases of structural elements render the model unstable. How can I deal with this situation?

A: The Analytical Model Check feature introduced in Revit Structure R3 can find some of these issues. When importing the analytical model, you are asked if you want to retain the release conditions from RST (Revit Structure) or if you want to set all beams and columns to fixed. When re-importing the model to RST, always update the end releases and do not overwrite the end releases on subsequent export to analysis programs.

Q: I am rotating the beam orientation so they are rotated in the weak direction. For example, the I of a W14X30 is rotated to look like an H by a 90 degree rotation. How is that rotation angle accessed in the API?

Because the location is a LocationCurve not a LocationPoint I do not have access to the Rotation value so what is it I need to check? I have a FamilyInstance element to check so what do I do with it?

A: Take a look at the RotateFramingObject example in the SDK. It has examples of how to get and change the beam braces and columns rotation angle.

Q: How do I add new concrete beam and column sizes to a model?

A: Take a look at the FrameBuilder sample code in the SDK

Q: How do I view the true deck layer?

A: There is an example in the SDK called DeckProperties that provides information about how to get the layer information for the deck. The deck information is reported in exactly the same way as it is in the UI. The deck dimension parameters are shown as follows.

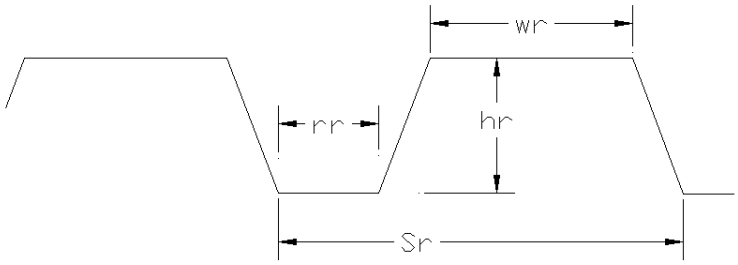


Figure 169: Deck dimension parameters

Q: How do I tell when I have a beam with a cantilever?

A: There is no direct way in the Revit database to tell if a beam has a cantilever. However, one or more of the following options can give you a good guess at whether a section is a cantilever:

4. There are two parameters called Moment Connection Start and Moment Connection End. If the value set for these two is not None then you should look and see if there is a beam that is co-linear and also has the value set to something other than None. Also ask the user to make sure to select Cantilever Moment option rather than Moment Frame option.

Trace the connectivity back beyond the element approximately one or two elements.

Look at element release conditions.

Q: When exporting a model containing groups to an external program, the user receives the following error at the end of the export:

"Changes to group "Group 1" are allowed only in group edit mode. Use the Edit Group command to make the change to all instances of the group. You may use the "Ungroup" option to proceed with this change by ungrouping the changed group instances."

A: Currently the API does not permit changes to group members. You can programmatically ungroup, make the change, regroup and then swap the other instances of the old group to the new group to get the same effect.

C. Registering Add-Ins Using Revit.ini

Prior to Revit 2011, external commands and applications were always registered using the Revit.ini. However, using manifest files is now the preferred method and the ini file method of registering add-ins will be deprecated in Revit 2012.

The process for registering external commands and external applications via the Revit.ini file is similar. Entries are added to the Revit.ini file under either the [ExternalCommands] or [ExternalApplications] section and those entries are read from the Revit.ini when Revit starts.

Note: The Revit.ini is not read at any other point during application execution.

[ExternalCommands]

In the Revit.ini [ExternalCommands] section, you can specify external tools that are loaded on start-up. There can only be one [ExternalCommands] in the revit.ini file.

Example:

Code Region 30-7: Revit.ini ExternalCommands Section

```
[ExternalCommands]
ECCount=1
ECClassName1=Project1.Class1
ECAssembly1=C:\Project1\Project1.dll
ECName1="My Tool"
ECDescription1="Implementation of My Tool within Revit"
```

The following table describes the entries in the Revit.ini section:

Tag	Description
ECCount	The total number of external commands available. Should be updated when new command entries are added.
ECClassName#	Provides the name of the class that implements the IExternalCommand interface. The command object is the full class name (with the namespace), such as MyNamespace.MyClass. The # is the command number, such as ECClassName1.
ECAssembly#	This property identifies the path to the Assembly containing the .NET command object. The path can be relative to the Revit installation directory of fully qualified. The # identifies the command number, such as ECAssembly1. The command number must match the command number used for the ECClassName property. In the previous example, the fully qualified path for the assembly is C:\Project1\Project1.dll.
ECName#	Provides a short name for the external command in the External Tools menu-button. The # identifies the command number, such as ECName1. The command number must match the command number used for the ECClassName property. In the previous example, the menu item for command #1 is "My Tool".
ECDescription#	An optional string that appears in the Revit status bar when the mouse moves over the menu item. The help string is a one-line description of the external command. In the previous example, the message "Implementation of My Tool within Revit" appears in the status bar when the mouse moves over the My Tool

	menu item.

[ExternalApplications]

In the Revit.ini, you list external applications in the [ExternalApplications] section.

You can load and run external applications using the Add-in Manager utility provided with the Revit Platfor SDK instead of manually editing the Revit.ini file.

Revit-style Task Dialogs

A TaskDialog is a Revit-style alternative to a simple Windows MessageBox. It can be used to display information and receive simple input from the user. It has a common set of controls that are arranged in a standard order to assure consistent look and feel with the rest of Revit.

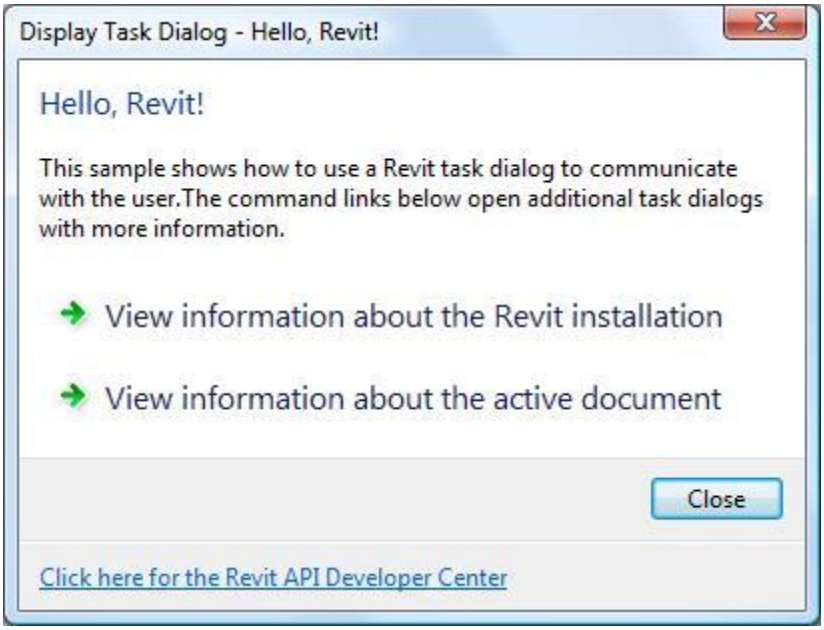


Figure 18: Revit-style Task Dialog

There are two ways to create and show a task dialog to the user. The first option is to construct the TaskDialog, set its properties individually, and use the instance method Show() to show it to the user. The second is to use one of the static Show() methods to construct and show the dialog in one step. When you use the static methods only a subset of the options can be specified.

Please see the Task Dialog section of the API User Interface Guidelines appendix for information on developing a task dialog that is compliant with the standards used by Autodesk.

The following example shows how to create and display the task dialog shown above.

Code Region 3-27: Displaying Revit-style TaskDialog

```
[Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.Automatic)]
[Autodesk.Revit.Attributes.Regeneration(Autodesk.Revit.Attributes.RegenerationOption.Manual)]
class TaskDialogExample : IExternalCommand
{
    public Autodesk.Revit.UI.Result Execute(ExternalCommandData commandData,
        ref string message, Autodesk.Revit.DB.ElementSet elements)
    {
        // Get the application and document from external command data.
```

```
        Application app = commandData.Application.Application;
        Document activeDoc = commandData.Application.ActiveUIDocument.Document;

        // Creates a Revit task dialog to communicate information to the user.
        TaskDialog mainDialog = new TaskDialog("Hello, Revit!");
        mainDialog.MainInstruction = "Hello, Revit!";
        mainDialog.MainContent =
            "This sample shows how to use a Revit task dialog to communicate with the user."
            + "The command links below open additional task dialogs with more information.";

        // Add commmandLink options to task dialog
        mainDialog.AddCommandLink(TaskDialogCommandLinkId.CommandLink1,
                                  "View information about the Revit installation");
        mainDialog.AddCommandLink(TaskDialogCommandLinkId.CommandLink2,
                                  "View information about the active document");

        // Set common buttons and default button. If no CommonButton or CommandLink is added,
        // task dialog will show a Close button by default
        mainDialog.CommonButtons = TaskDialogCommonButtons.Close;
        mainDialog.DefaultButton = TaskDialogResult.Close;

        // Set footer text. Footer text is usually used to link to the help document.
        mainDialog.FooterText =
          "<a href=\"http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=2484975 \">"
            + "Click here for the Revit API Developer Center</a>";

        TaskDialogResult tResult = mainDialog.Show();

        // If the user clicks the first command link, a simple Task Dialog
        // with only a Close button shows information about the Revit installation.
        if (TaskDialogResult.CommandLink1 == tResult)
        {
            TaskDialog dialog_CommandLink1 = new TaskDialog("Revit Build Information");
            dialog_CommandLink1.MainInstruction =
                "Revit Version Name is: " + app.VersionName + "\n"
                + "Revit Version Number is: " + app.VersionNumber + "\n"
                + "Revit Version Build is: " + app.VersionBuild;

            dialog_CommandLink1.Show();
        }

        // If the user clicks the second command link, a simple Task Dialog
        // created by static method shows information about the active document
        else if (TaskDialogResult.CommandLink2 == tResult)
        {
            TaskDialog.Show("Active Document Inforamtion",
                "Active document: " + activeDoc.Title + "\n"
                + "Active view name: " + activeDoc.ActiveView.Name);
```

```
        }

        return Autodesk.Revit.UI.Result.Succeeded;
    }
}
```

for more information.

The following code example illustrates how to specify an external application:

Code Region 30-8: Revit.ini ExternalApplications section

```
[ExternalApplications]
EACount=1
EAClassName1=EA1.Class1
EAAssembly1=D:\EA1\bin\Debug\EA1.dll
```

The following table describes the entries in the Revit.ini section:

Tag	Description
EACount	The total number of external applications available. Should be updated when new application entries are added.
EAClassName#	Provides the name of the class that implements the IExternalApplication interface. For .NET objects it is the full class name (with the namespace), such as MyNamespace.MyClass. The # is the command number, such as EAClassName1.
EAAssembly#	This property identifies the path to the Assembly containing the .NET application object. The path can be relative to the Revit installation directory of fully qualified. The # identifies the application number, such as EAAssembly1. The application number must match the application number used for the EAClassName property. In the previous example, the Assembly has the fully qualified path D:\EA1\bin\Debug\EA1.dll.

Add-in Manager for the Revit.ini

The Revit Platform SDK includes a utility called the Add-in Manager, which makes it possible to load and run external commands and applications without having to edit the Revit.ini file. The installer for this utility is located in the SDK, in the Add-in Manager directory. Read the Add-In Manager Read Me.doc file, located in this directory, for more information about installing and using the Add-in Manager.

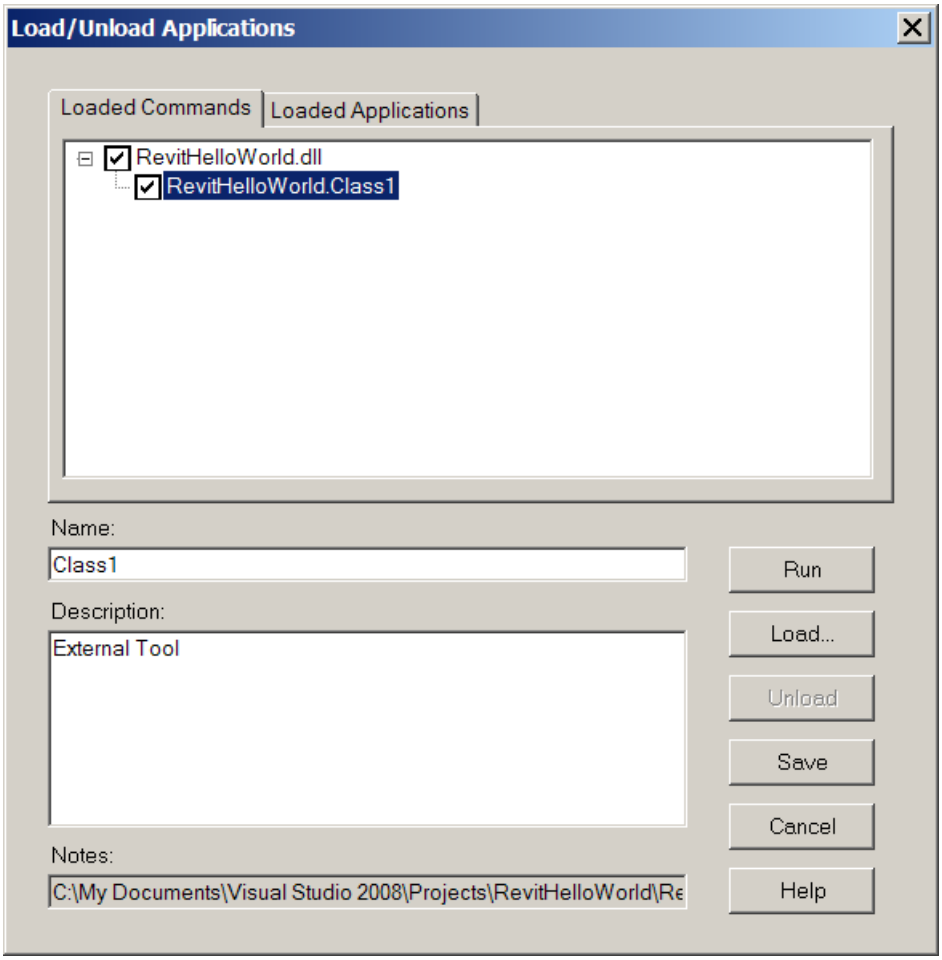


Figure 170: The Add-in Manager

D. Hello World for VB.NET

Directions for creating the sample application for Visual Basic .NET are available in the following sections. The sample application was created using Microsoft Visual Studio.

Create a New Project

The first step in writing a VB.NET program with Visual Studio is to choose a project type and create a new project.

1. From the File menu, select New> Project….
2. In the Project types frame, click Visual Basic.
3. In the Templates frame, click Class Library. The application assumes that your project location is: D:\Sample.
4. In the Name field, type HelloWorld as the project name.
5. Click OK.

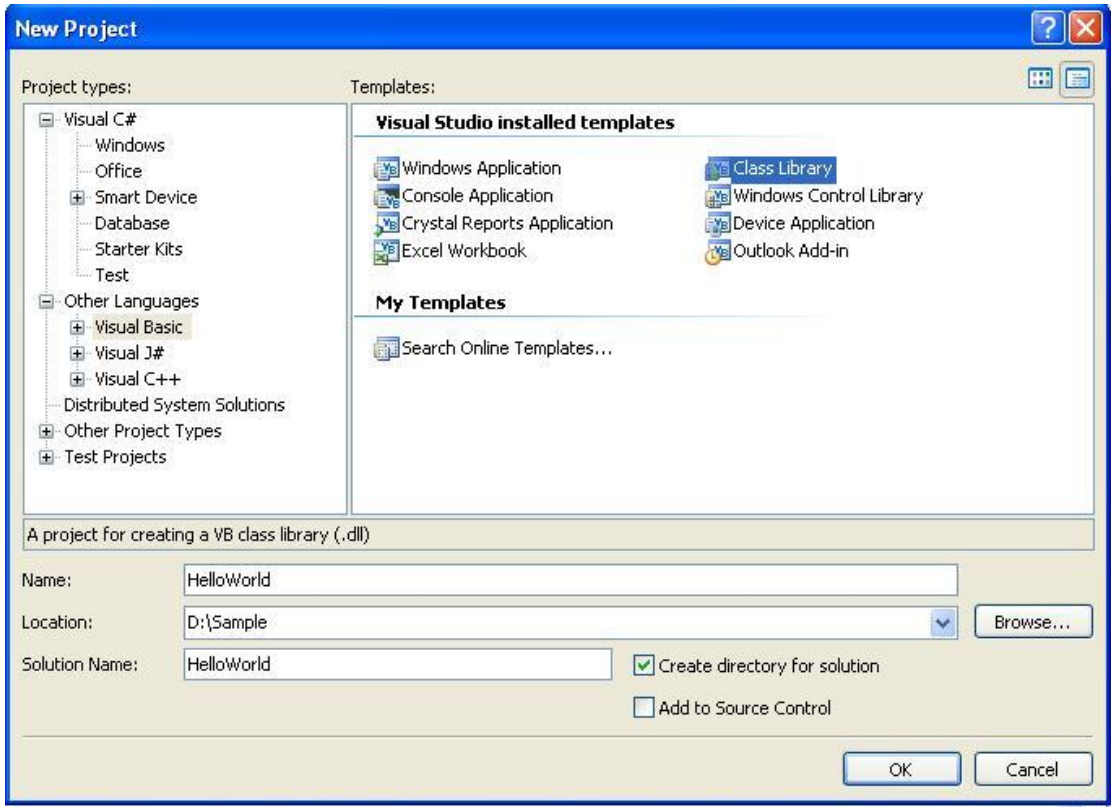


Figure 171: New Project dialog box

Add Reference and Namespace

VB.NET uses a process similar to C#. After you create the Hello World project, complete the following steps:

1. Right-click the project name in the Solution Explorer to display a context menu.
2. From the context menu, select Properties to open the Properties dialog box.
3. In the Properties dialog box, click the References tab. A list of references and namespaces appears.

4. Click the Add button to open the Add Reference dialog box.
5. In the Add Reference dialog box, click the Browse tab. Locate the folder where Revit is installed and click the RevitAPI.dll. For example the installed folder location might be C:\Program Files\Autodesk\Revit Architecture 2011\Program\RevitAPI.dll.
6. Click OK to add the reference and close the dialog box.
7. Repeat steps above to add a reference to RevitAPIUI.dll, which is in the same folder as Revit API.dll.

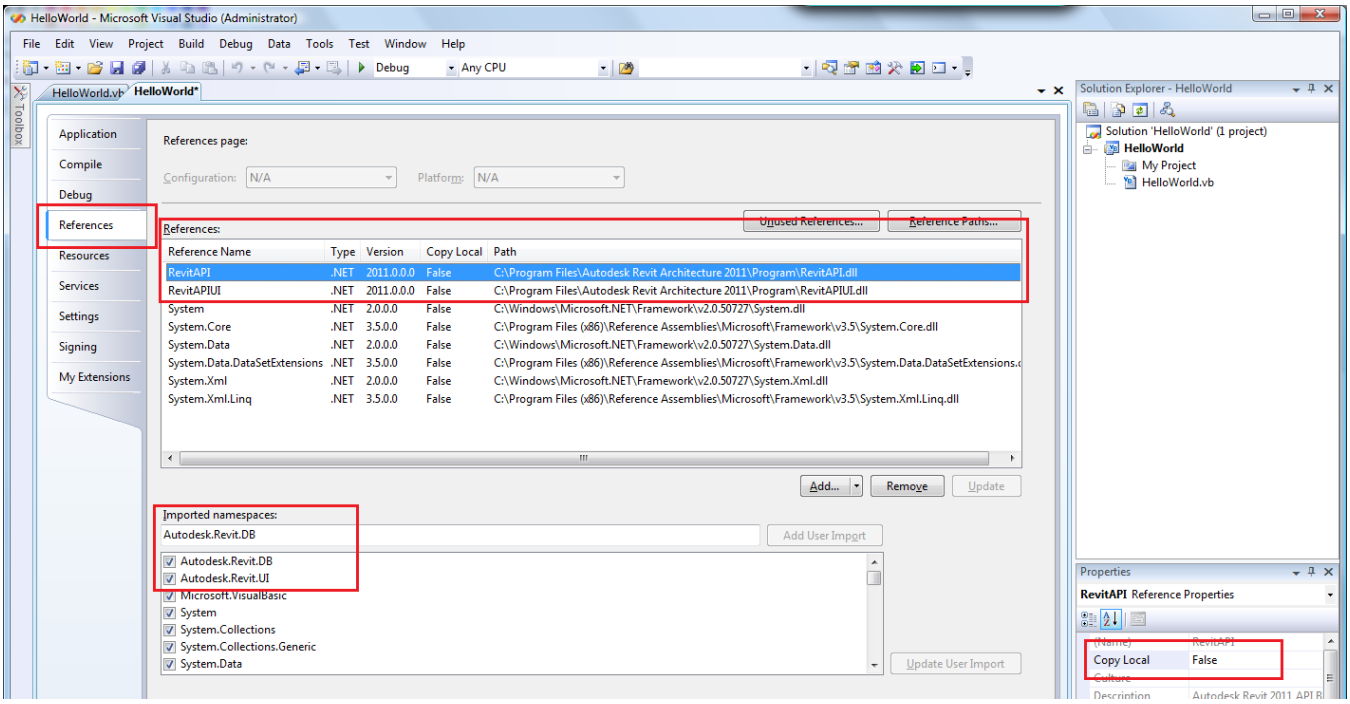


Figure 172: Add references and import Namespaces

8. After adding the reference, you must import the namespaces used in the project. For this example, import the Autodesk.Revit.DB and Autodesk.Revit.UI namespaces.
9. To complete the process, click RevitAPI in the Reference frame to highlight it. Set Copy Local to False in the property frame. Repeat for the RevitAPIUI.dll.

Change the Class Name

To change the class name, complete the following steps:

1. In the Solution Explorer, right-click Class1.vb to display a context menu.
2. From the context menu, select Rename. Rename the file HelloWorld.vb.
3. In the Solution Explorer, double-click HelloWorld.vb to open it for editing.

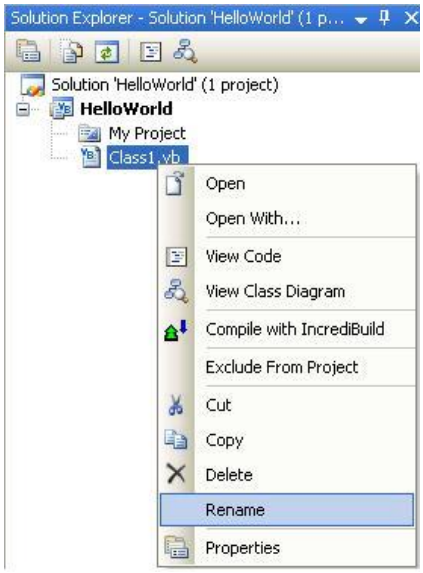


Figure 173: Change the class name

Add Code

When writing the code in VB.NET, you must pay attention to key letter capitalization.

Code Region 30-9: Hello World in VB.NET

```
Imports System

Imports Autodesk.Revit.UI
Imports Autodesk.Revit.DB

<Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.Automatic)>
_
<Autodesk.Revit.Attributes.Regeneration(Autodesk.Revit.Attributes.RegenerationOption.Manual)>
_
Public Class Class1
Implements IExternalCommand
    Public Function Execute(ByVal revit As ExternalCommandData, ByRef message As String, _
                            ByVal elements As ElementSet) As Autodesk.Revit.UI.Result _
                            Implements IExternalCommand.Execute

        TaskDialog.Show("Revit", "Hello World")
        Return Autodesk.Revit.UI.Result.Succeeded

    End Function
End Class
```

Create a .addin manifest file

The HelloWorld.dll file appears in the project output directory. If you want to invoke the application in Revit, create a manifest file to register it into Revit.

1. To create a manifest file, create a new text file in Notepad.
2. Add the following text:

Code Region 30-10: Creating a .addin manifest file for an external command

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
  <AddIn Type="Command">
    <Assembly>D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll</Assembly>
    <AddInId>239BD853-36E4-461f-9171-C5ACEDA4E721</AddInId>
    <FullClassName>Class1</FullClassName>
    <Text>HelloWorld</Text>
  </AddIn>
</RevitAddIns>
```

3. Save the file as HelloWorld.addin and put it in the following location:

- For Windows XP - C:\Documents and Settings\All Users\Application Data\Autodesk\Revit\Addins\2011\
- For Vista/Windows 7 - C:\ProgramData\Autodesk\Revit\Addins\2011\

Refer to the Add-in Integration chapter for more details using manifest files.

Build the Program

After completing the code, you must build the file. From the Build menu, click Build Solution. Output from the build appears in the Output window indicating that the project compiled without errors.

Debug the Program

Running a program in Debug mode uses breakpoints to pause the program so that you can examine the state of variables and objects. If there is an error, you can check the variables as the program runs to deduce why the value is not what you might expect.

1. In the Solution Explorer window, right-click the HelloWorld project to display a context menu.
2. From the context menu, click Properties. The Properties window appears.
3. Click the Debug tab.
4. In the Debug window Start Action section, click Start external program and browse to the Revit.exe file. By default, the file is located at the following path, C:\Program Files\Autodesk\Revit Structure 2011\Program\Revit.exe.

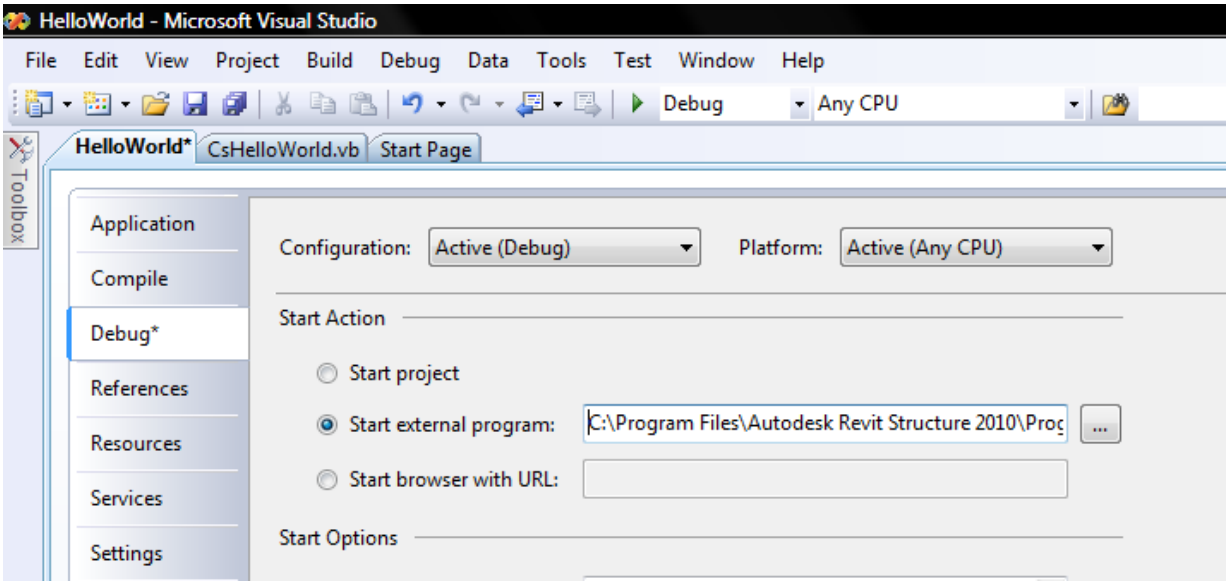


Figure 174: Set Debug environment

5. From the **Debug** menu, select **Toggle Breakpoint** (or press F9) to set a breakpoint on the following line.

Code Region 30-11: TaskDialog

```
TaskDialog.Show("Revit", "Hello World")
```

6. Press F5 to start the debug procedure.
7. Test the debugging
   - On the Add-Ins tab, HelloWorld appears in the External Tools menu-button.

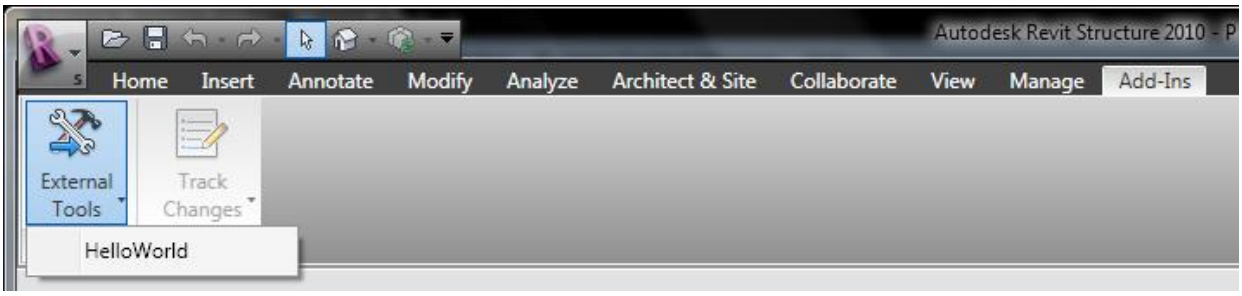


Figure 175: HelloWorld External Tools command

- Click HelloWorld to execute the program, activating the breakpoint.
- Press F5 to continue executing the program. The following system message appears.

Figure 176: TaskDialog message

E. Material Properties Internal Units

Parameter Name in Dialog	API Parameter Name	Variable Type	Internal Database Units	Description
Steel / Generic				
Behavior	Behavior	bool	Isotropic, Orthotropic	Currently exposed for Generic Only
Young Modulus X	YoungModulusX	double	UT_Stress	
Young Modulus Y	YoungModulusY	double	UT_Stress	
Young Modulus Z	YoungModulusZ	double	UT_Stress	
Poisson ratio X	PoissonModulusX	double	UT_Number	
Poisson ratio Y	PoissonModulusY	double	UT_Number	
Poisson ratio Z	PoissonModulusZ	double	UT_Number	
Shear modulus X	ShearModulusX	double	UT_Stress	
Shear modulus Y	ShearModulusY	double	UT_Stress	
Shear modulus Z	ShearModulusZ	double	UT_Stress	
Thermal Expansion coefficient X	ThermalExpansionCoefficientX	double	UT_TemperalExp	
Thermal Expansion coefficient Y	ThermalExpansionCoefficientY	double	UT_TemperalExp	
Thermal Expansion coefficient Z	ThermalExpansionCoefficientZ	double	UT_TemperalExp	
Unit Weight	UnitWeight	double	UT_UnitWeight	
Damping ratio	DampingRatio	double	UT_Number	
Minimum Yield Stress	MinimumYieldStress	double	UT_Stress	Fy in US codes. Also can be considered as the compression stress capacity
Minimum tensile stress	MinimumTensileStrength	double	UT_Stress	Only used for steel
Reduction factor for shear	ReductionFactor	double	UT_Number	Reduction of Minimum Yield Stress for Shear. Shear Yield Stress = MinimumYieldStress / ReductionFactor
Concrete				
Behavior	Behavior	bool	Isotropic, Orthotropic	Currently exposed for Generic Only

Young Modulus X	YoungModulusX	double	UT_Stress	
Young Modulus Y	YoungModulusY	double	UT_Stress	
Young Modulus Z	YoungModulusZ	double	UT_Stress	
Poisson ratio X	PoissonModulusX	double	UT_Number	
Poisson ratio Y	PoissonModulusY	double	UT_Number	
Poisson ratio Z	PoissonModulusZ	double	UT_Number	
Shear modulus X	ShearModulusX	double	UT_Stress	
Shear modulus Y	ShearModulusY	double	UT_Stress	
Shear modulus Z	ShearModulusZ	double	UT_Stress	
Thermal Expansion coefficient X	ThermalExpansionCoefficientX	double	UT_TemperalExp	
Thermal Expansion coefficient Y	ThermalExpansionCoefficientY	double	UT_TemperalExp	
Thermal Expansion coefficient Z	ThermalExpansionCoefficientZ	double	UT_TemperalExp	
Unit Weight	UnitWeight	double	UT_UnitWeight	
Damping ratio	DampingRatio	double	UT_Number	
Concrete compression	ConcreteCompression	double	UT_Stress	Concrete compression stress capacity. F’c for US codes.
Lightweight	LightWeight	Bool		If true then lightweight concrete is defined.
Shear strength modification	ShearStrengthReduction	double	UT_Number	When Lightweight = True then this value is available. It is the reduction of the concrete compression capacity for shear. Concrete Shear stress Capacity = ConcreteCompression / ShearStrengthReduction
Wood				
Young Modulus	PoissonModulus	double	UT_Stress	
Poisson ratio	ShearModulus	double	UT_Number	
Shear modulus	ShearModulus	double	UT_Stress	
Thermal Expansion coefficient	ThermalExpansionCoefficient	double	UT_TemperalExp	

Unit Weight	UnitWeight	double	UT_UnitWeight	
Species	Species	AString		
Grade	Grade	AString		
Bending	Bending	double	UT_Stress	
Compression parallel to grain	CompressionParallel	double	UT_Stress	
Compression perpendicular to grain	CompressionPerpendicular	double	UT_Stress	
Shear parallel to grain	ShearParallel	double	UT_Stress	
Tension perpendicular to grain	ShearPerpendicular	double	UT_Stress	

#	Unit	Dimension	Internal Representation
1.	UT_Number	No dimension	Simple number.
2.	UT_Stress	$(Mass)x(Length^{-1})x(Time^{-2})$	Kg(mass)/(Foot*Sec²)
3.	UT_TemperalExp	$(Temperature^{-1})$	(1/°C)
4.	UT_UnitWeight	$(Mass)x(Length^{-2})x(Time^{-2})$	Kg(mass)/(Foot²*Sec²)

F. Concrete Section Definitions

Concrete-Rectangular Beam

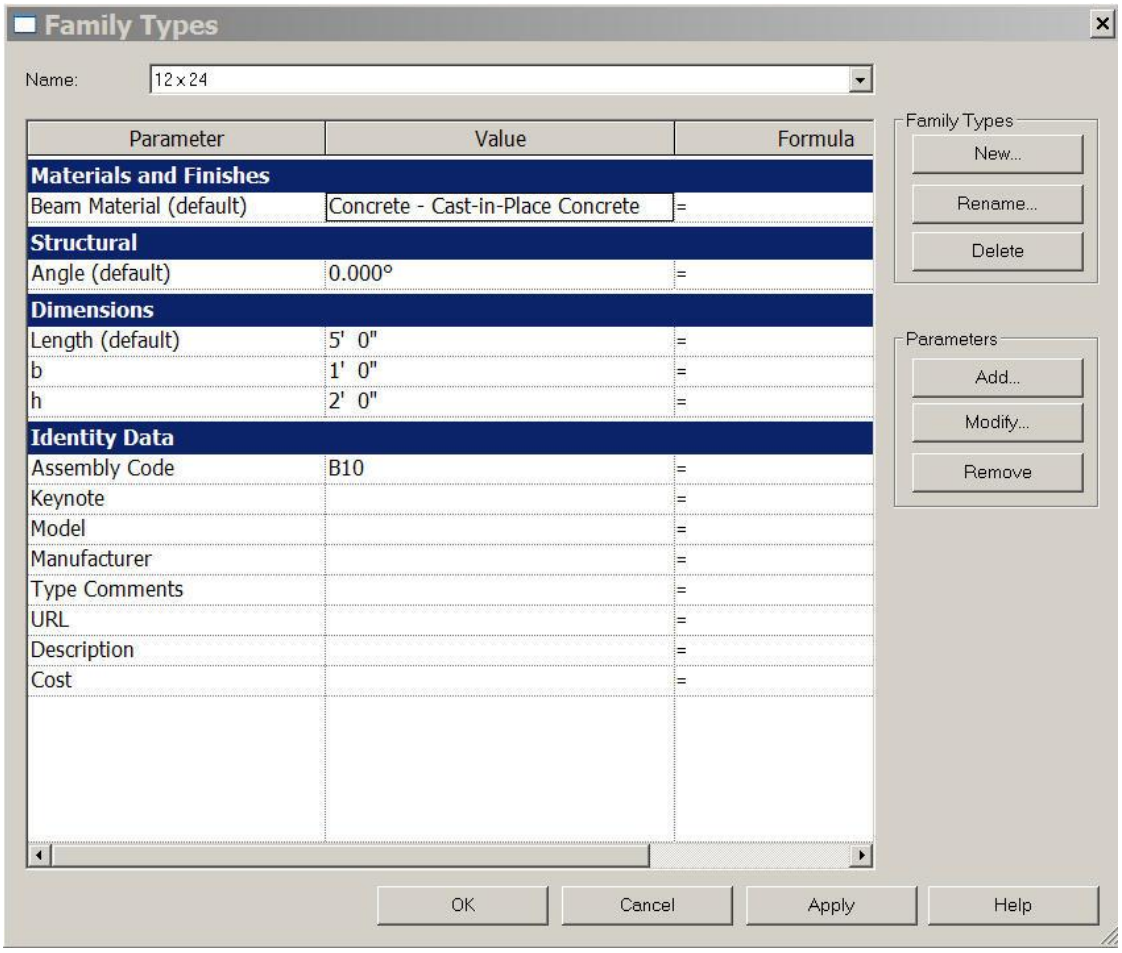


Figure 177: Concrete-Regangular Beam Parameters

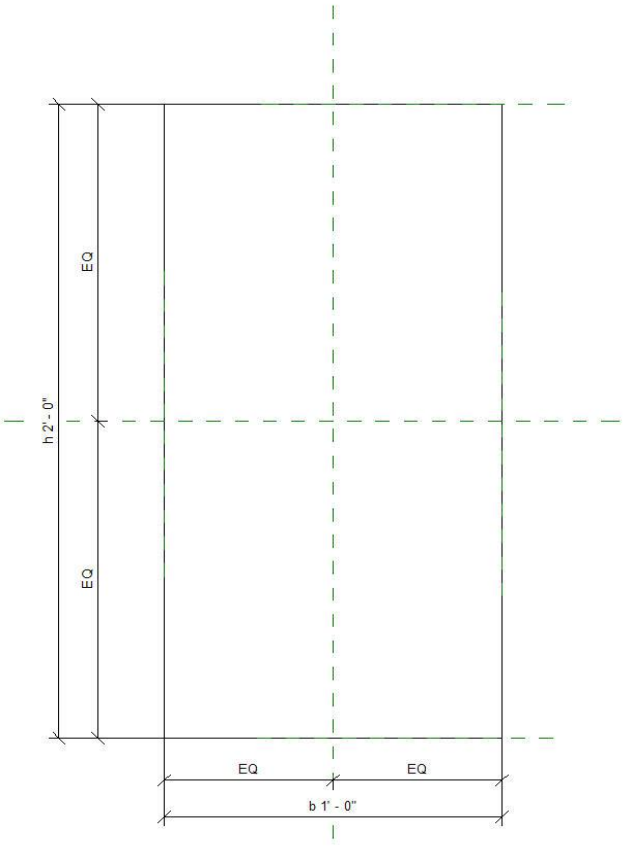


Figure 178: Concrete-Rectangular Beam Cross Section

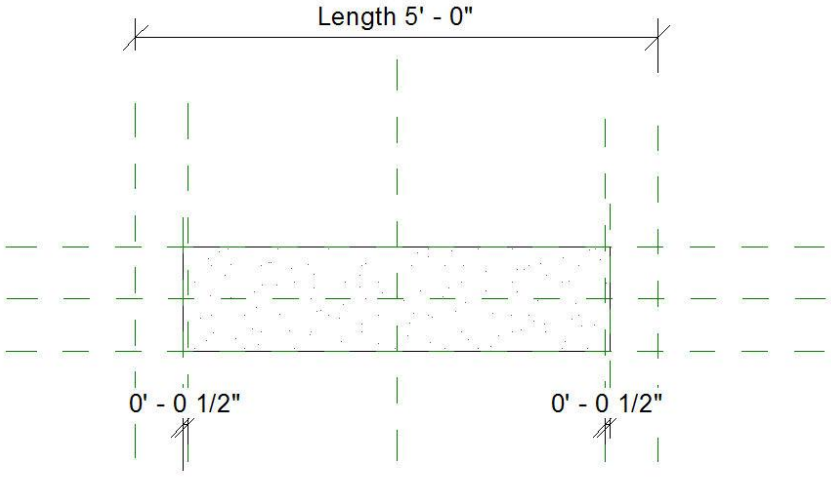


Figure 179: Concrete-Rectangular Beam

Precast-Rectangular Beam

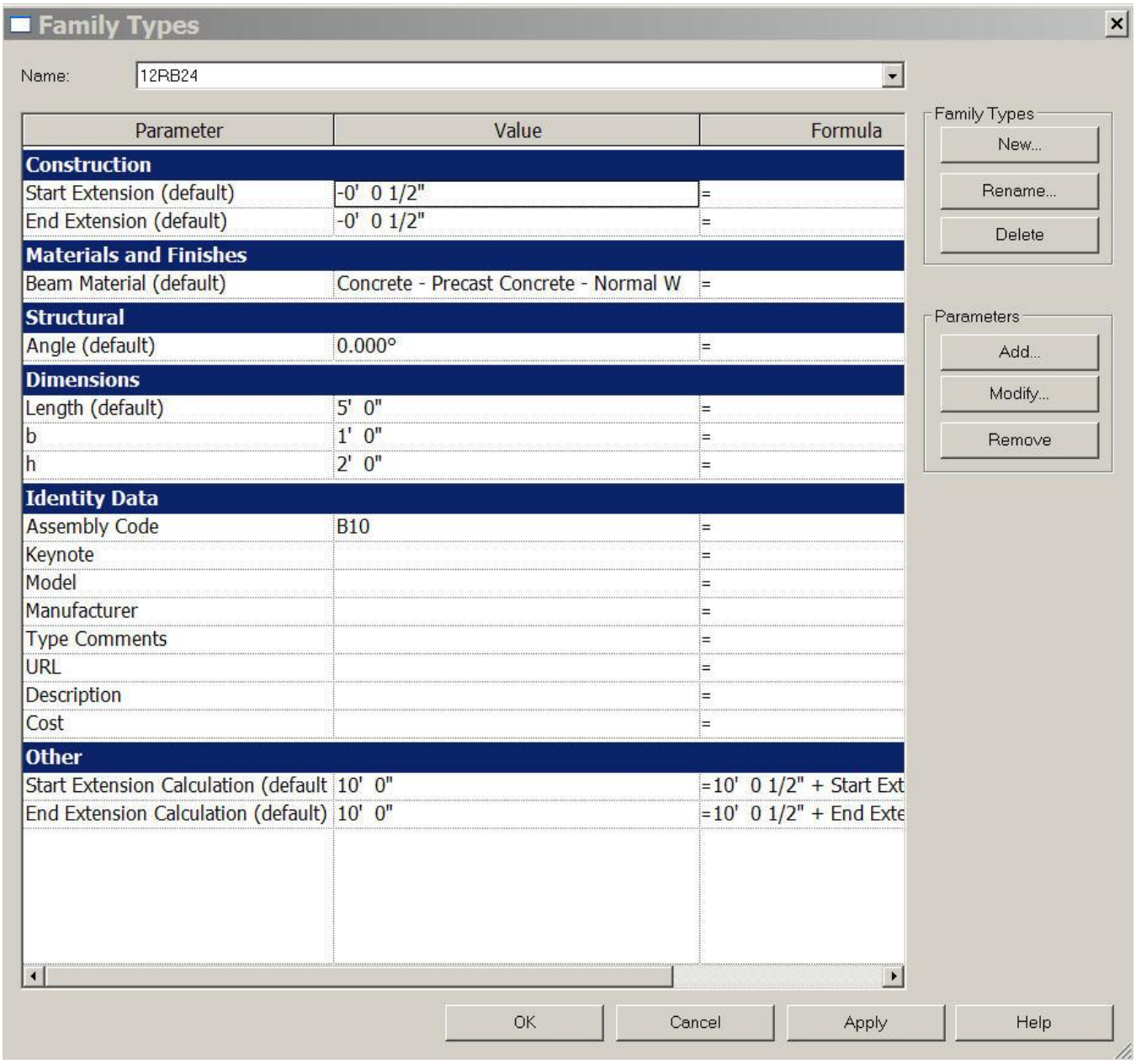


Figure 180: Precast-Rectangular Beam Properties

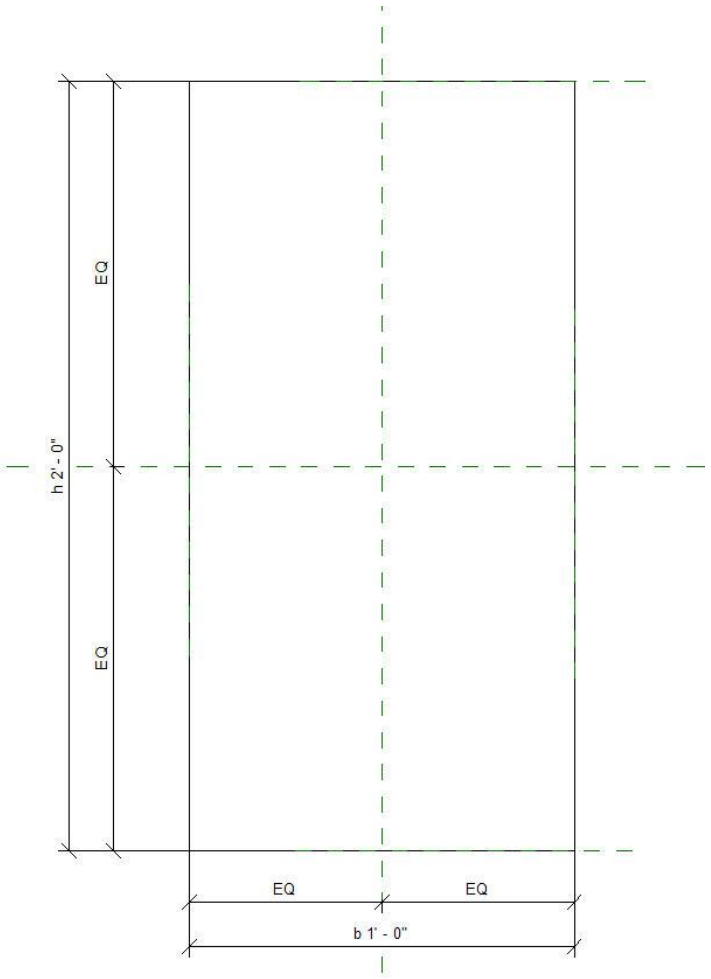


Figure 181: Precast-Rectangular Beam Cross Section

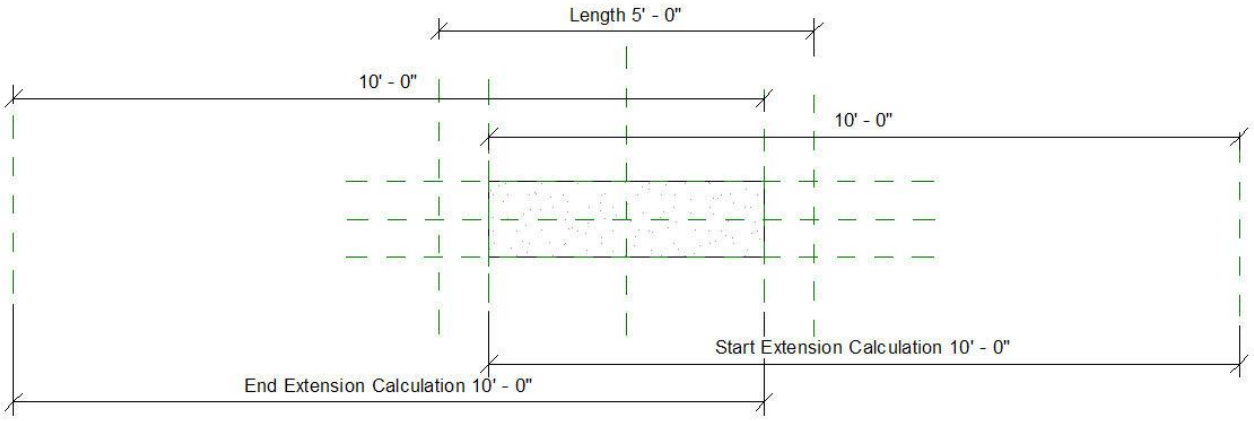


Figure 182: Precast-Rectangular Beam

Precast-L Shaped Beam

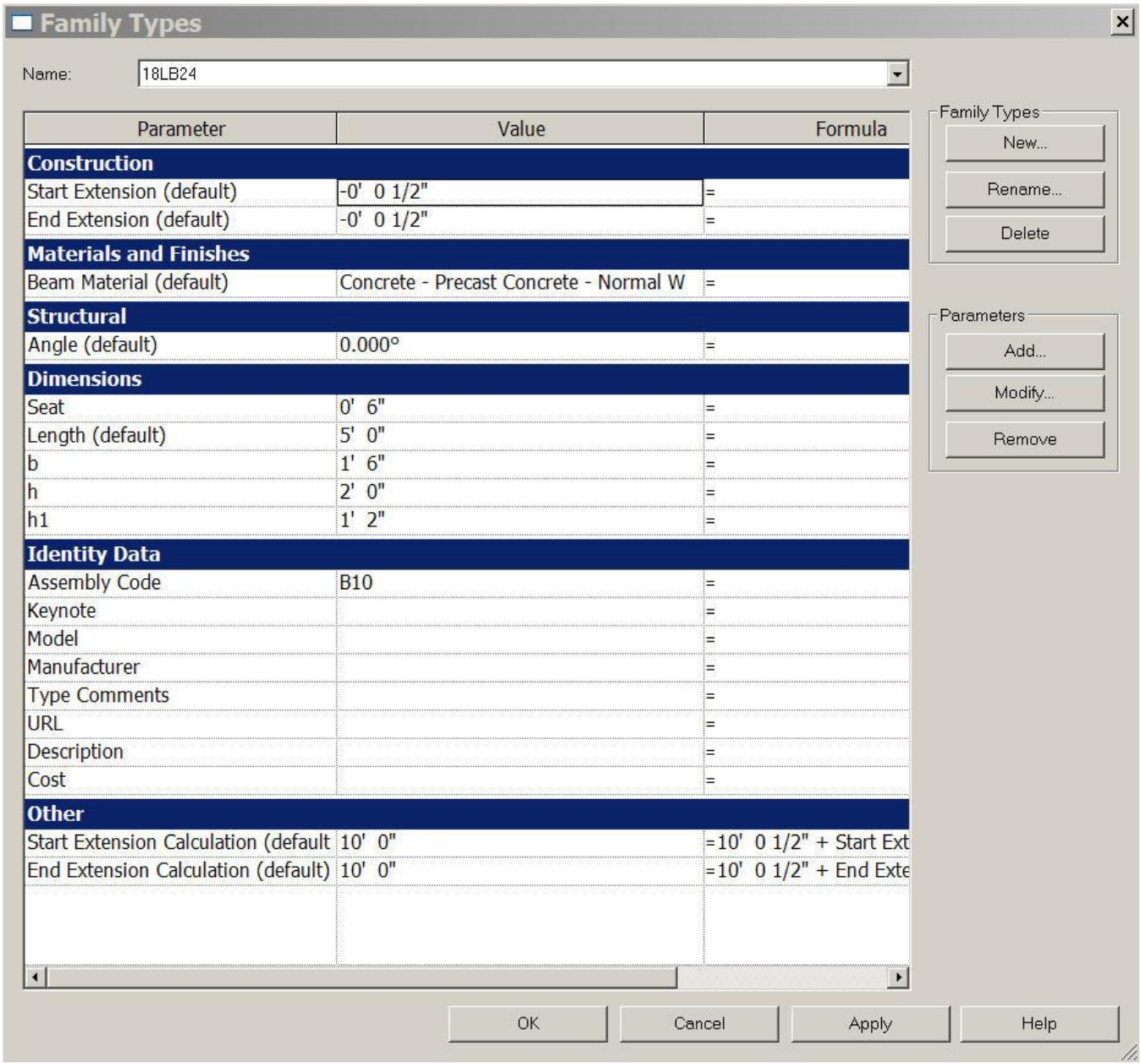


Figure 183: Precast-L Shaped Beam Properties

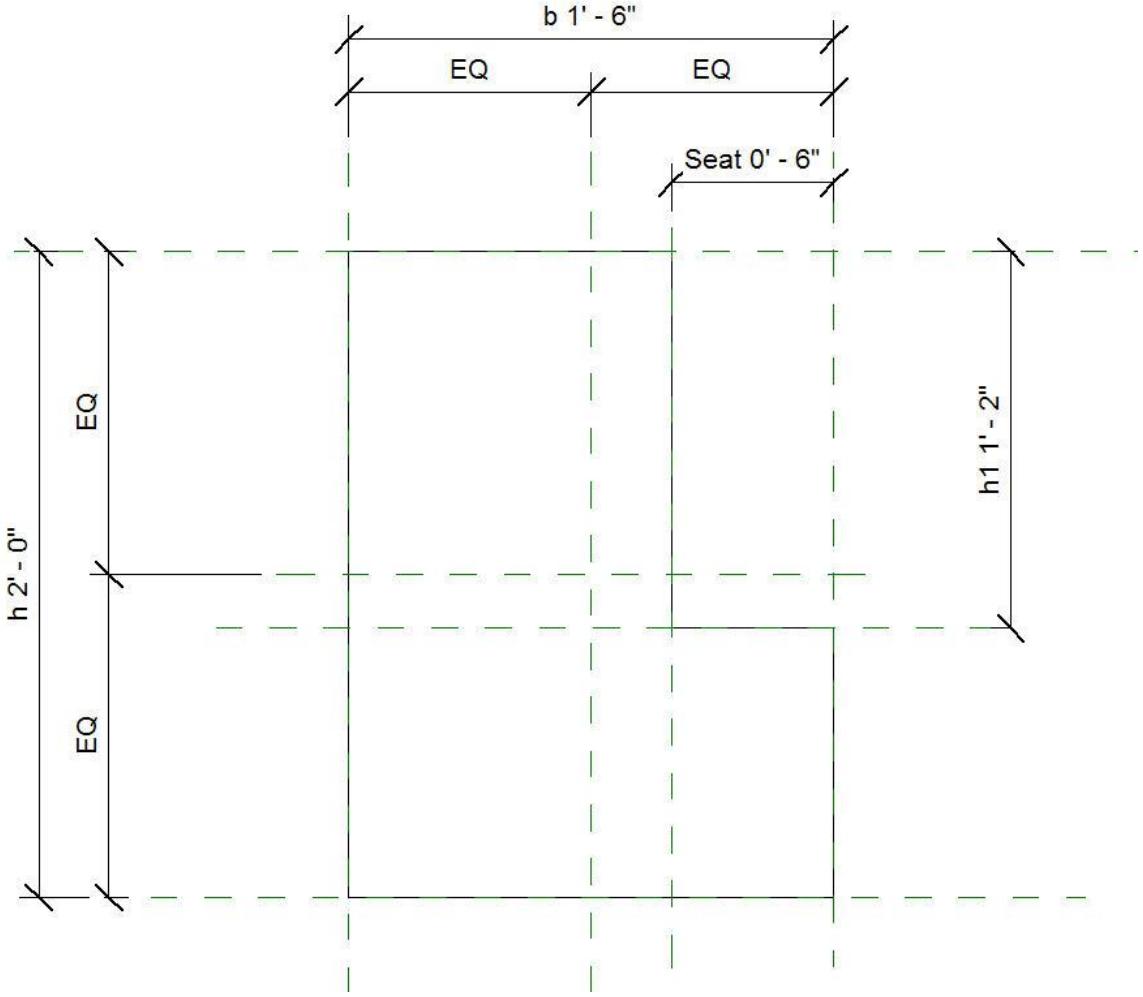


Figure 184: Precast-L Shaped Beam Cross Section

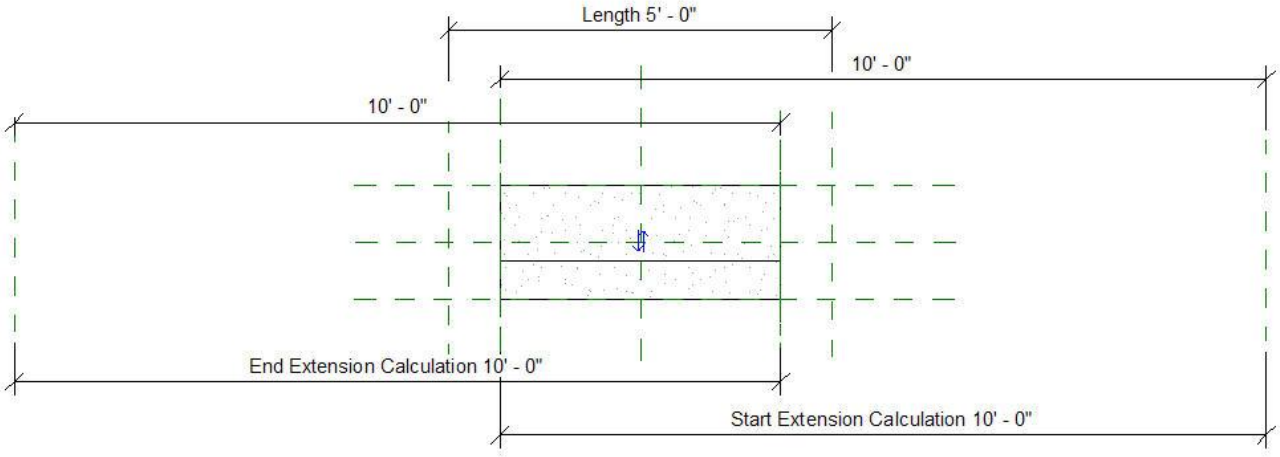


Figure 185: Precast-L Shaped Beam

Precast-Single Tee

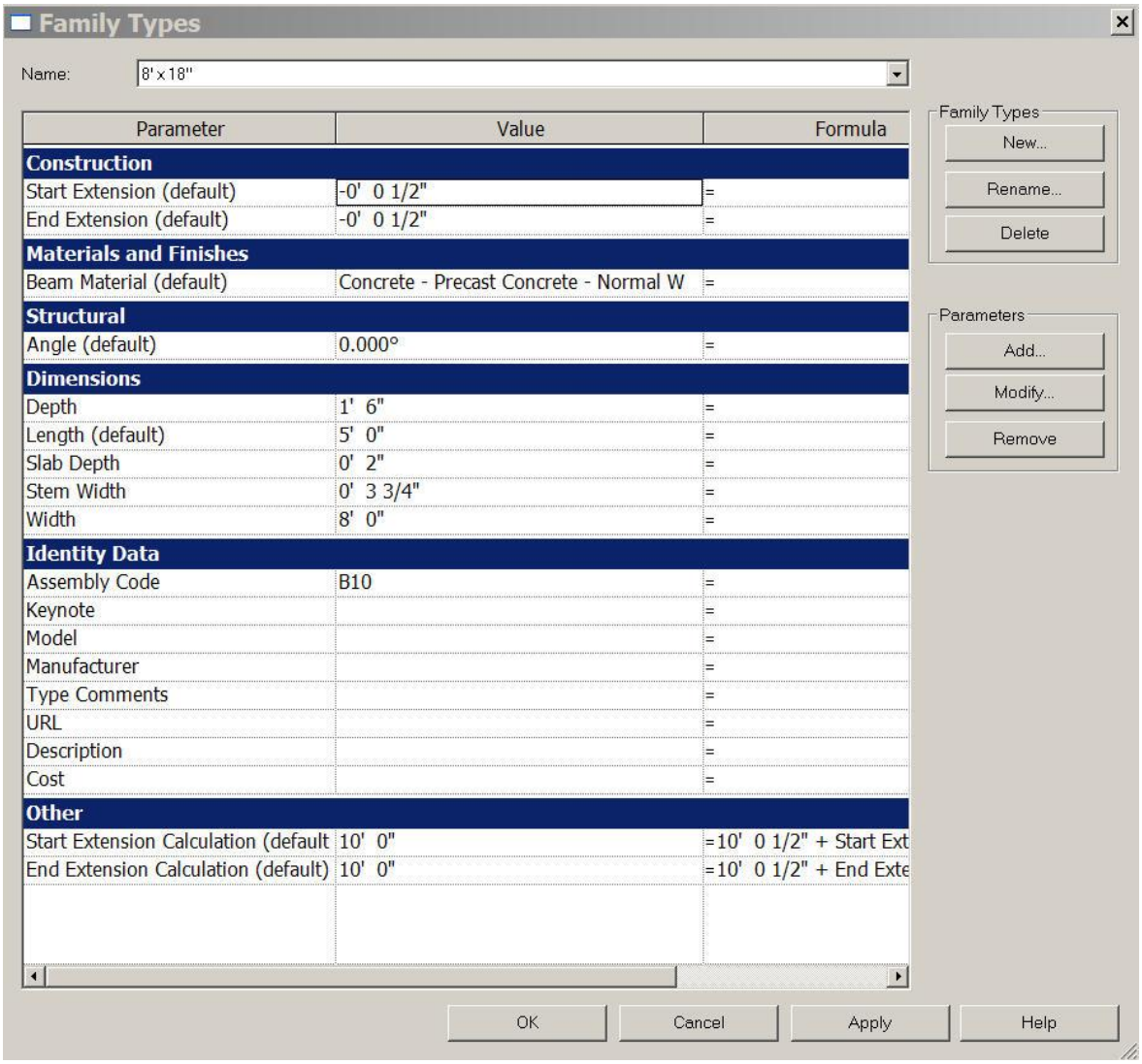


Figure 186: Precast-Single Tee Properties

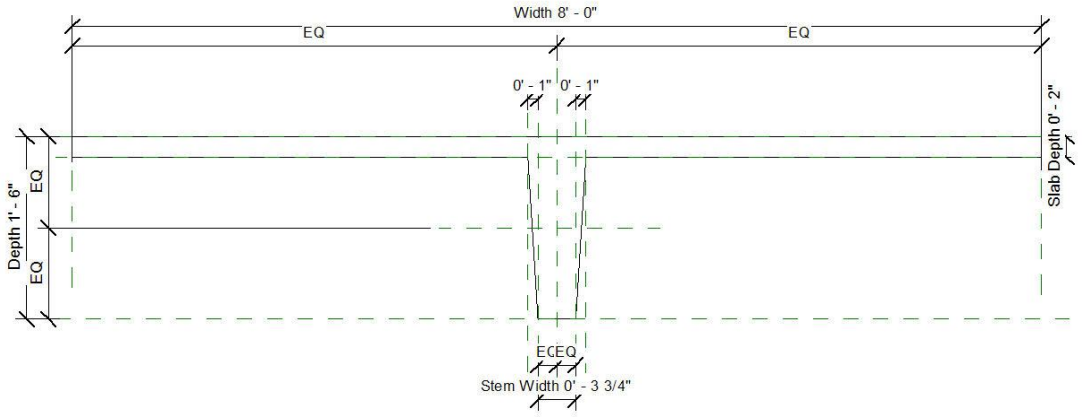


Figure 187: Precast-Single Tee Cross Section

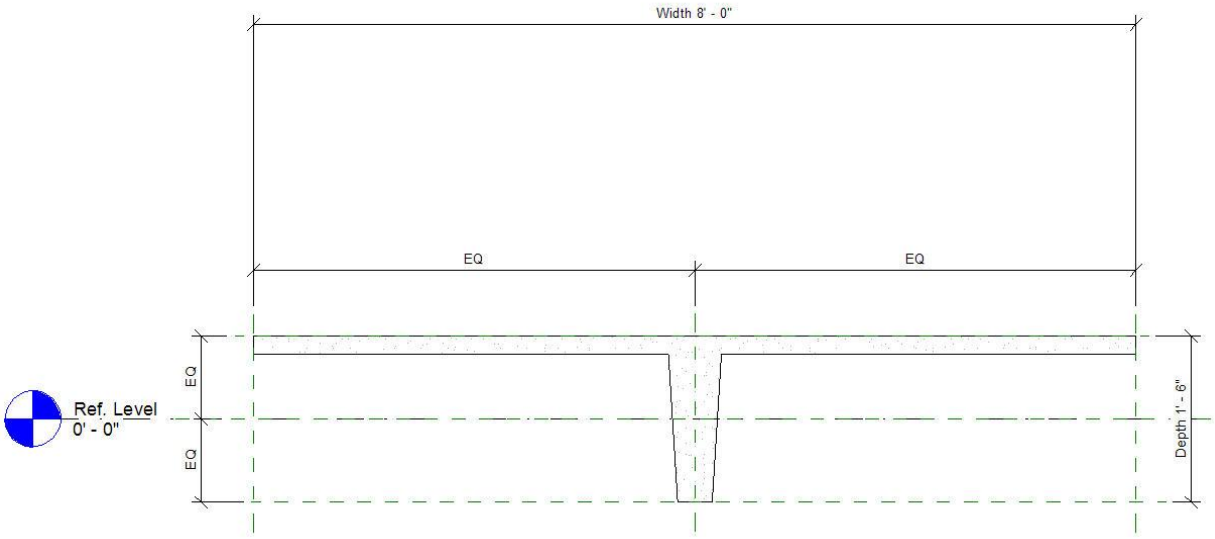


Figure 188: Precast-Single Tee Cross Section 2

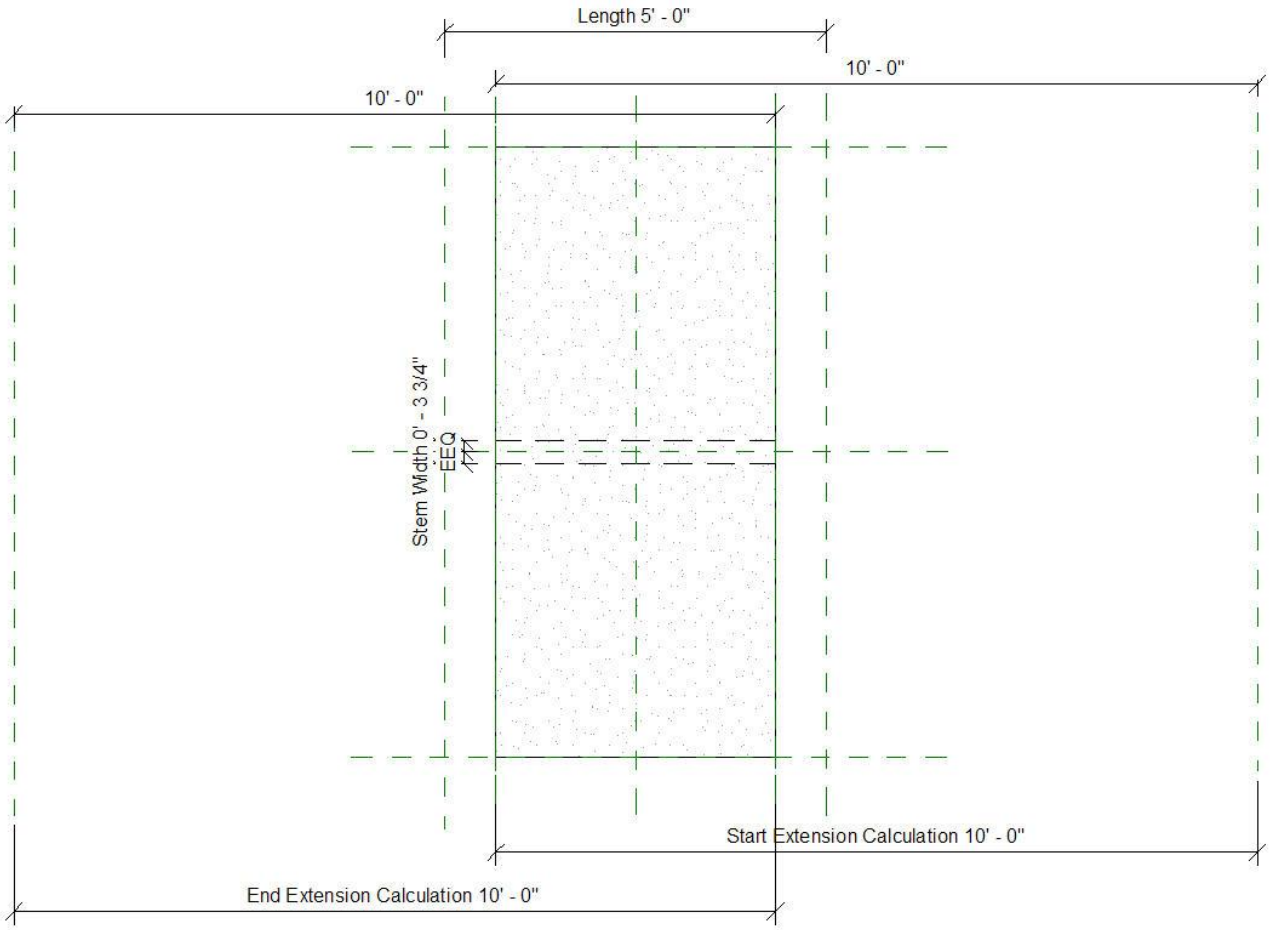


Figure 189: Precast-Single Tee

Precast-Inverted Tee

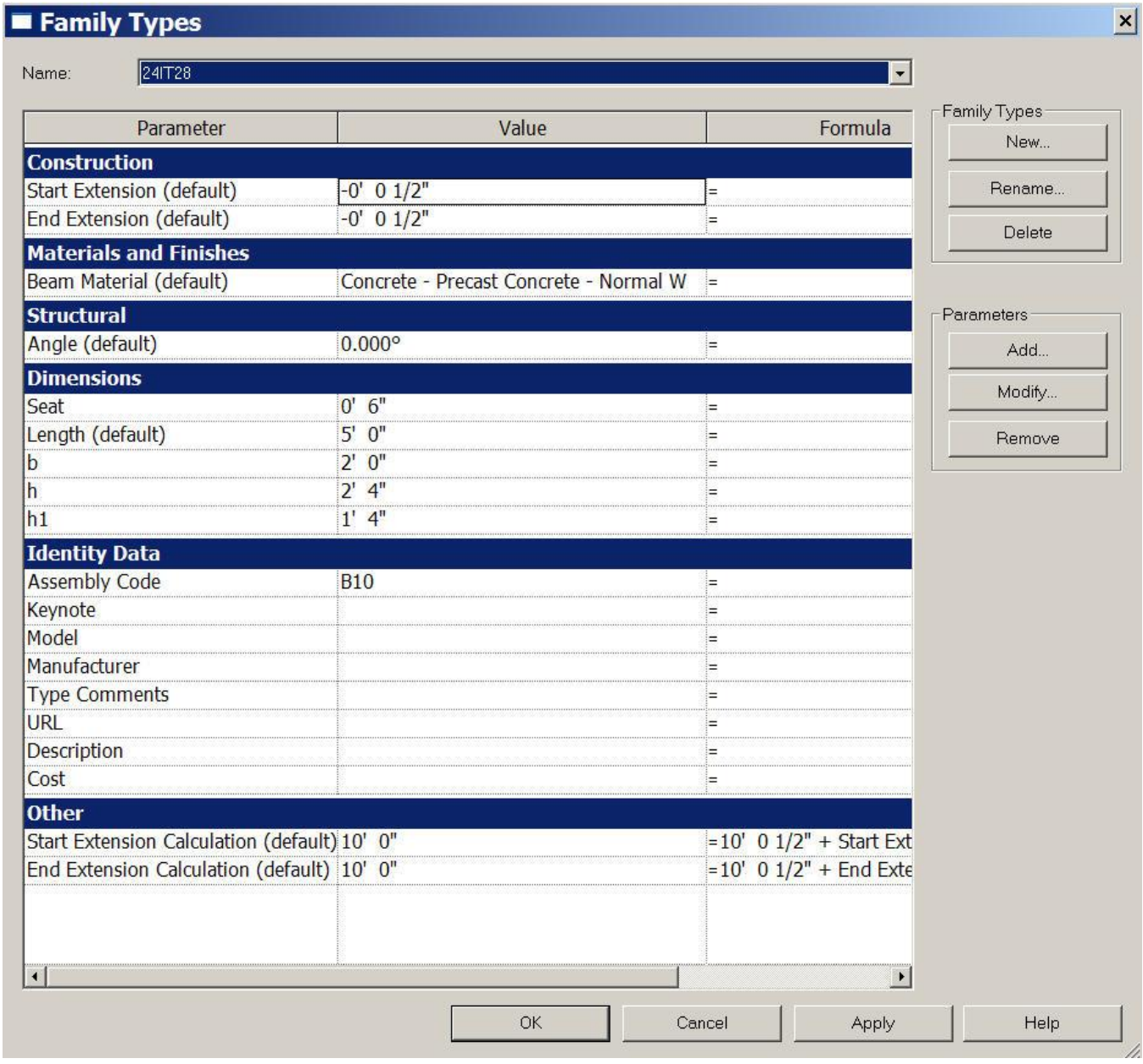


Figure 190: Precast-Inverted Tee Properties

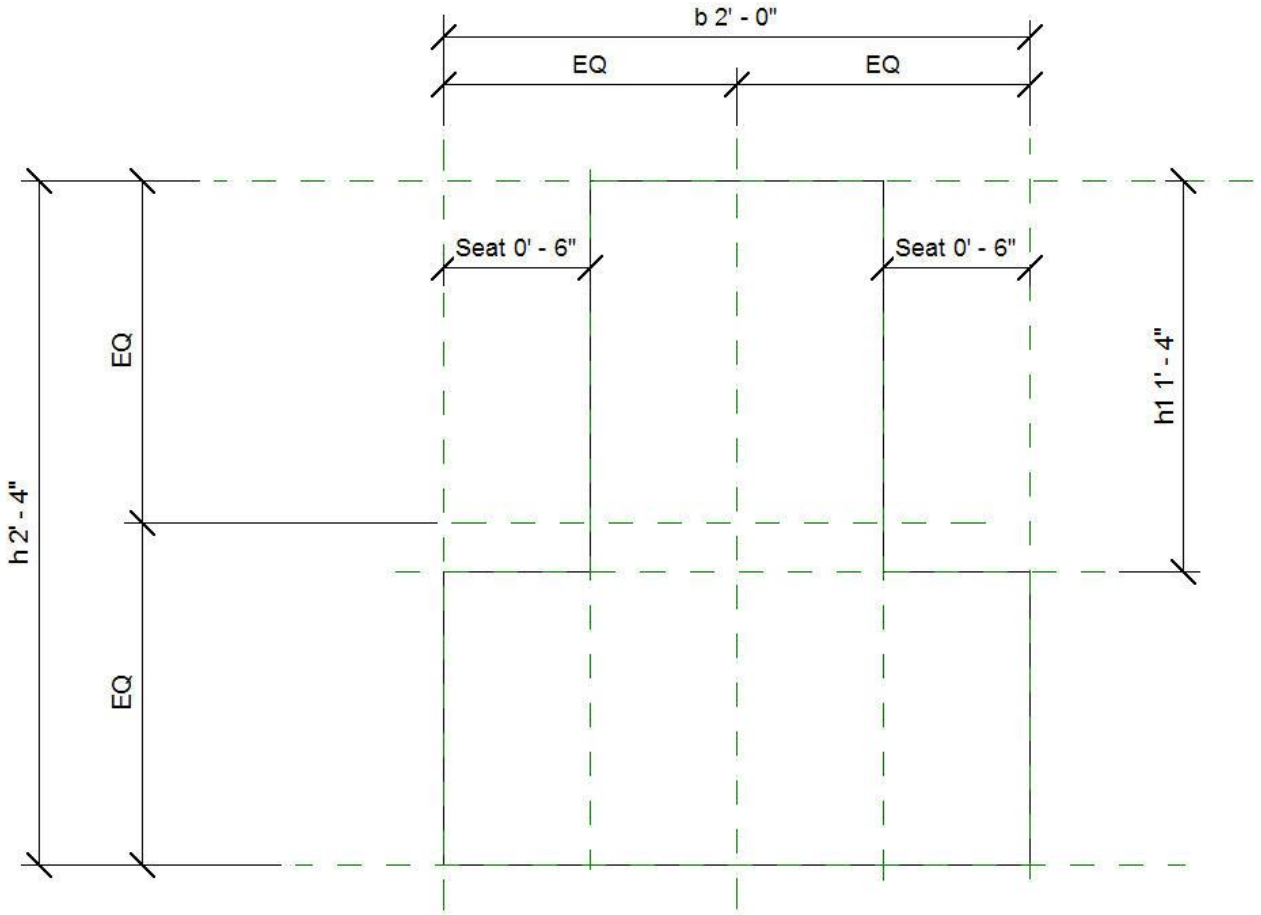


Figure 191: : Precast-Inverted Tee Cross Section

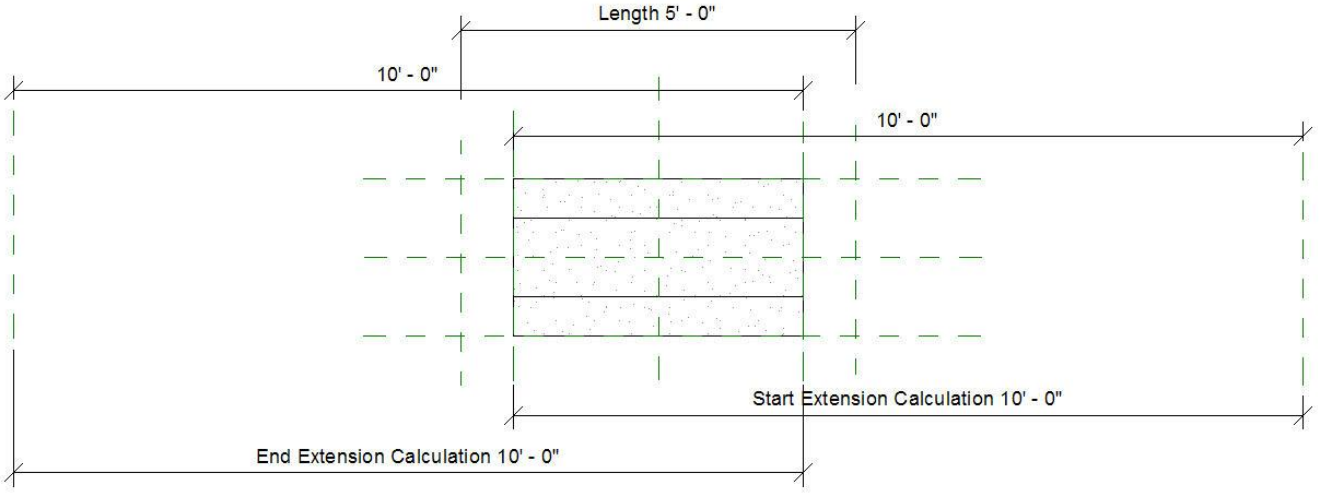


Figure 192: Precast-Inverted Tee

Precast-Double Tee

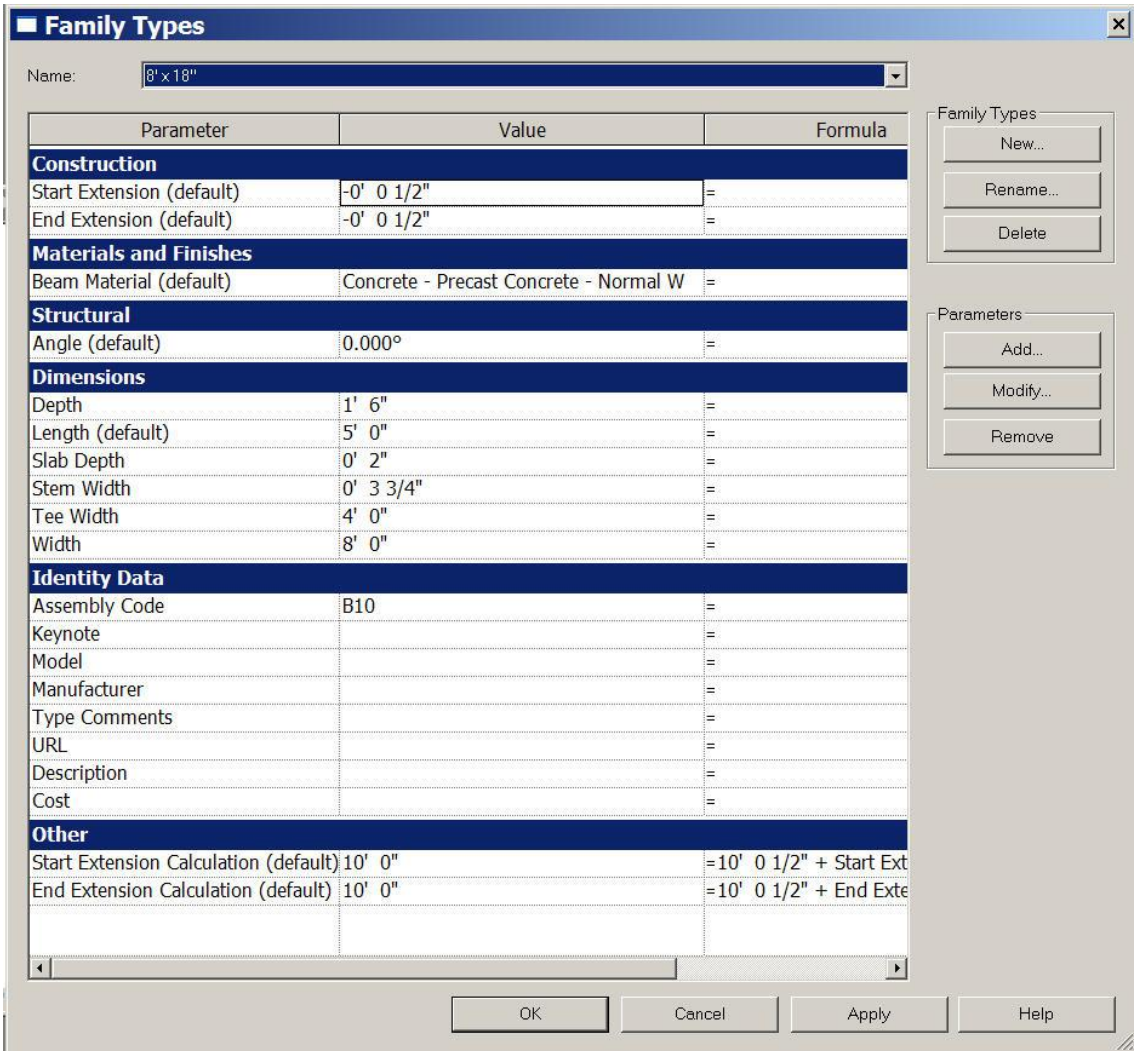


Figure 193: Precast-Double Tee

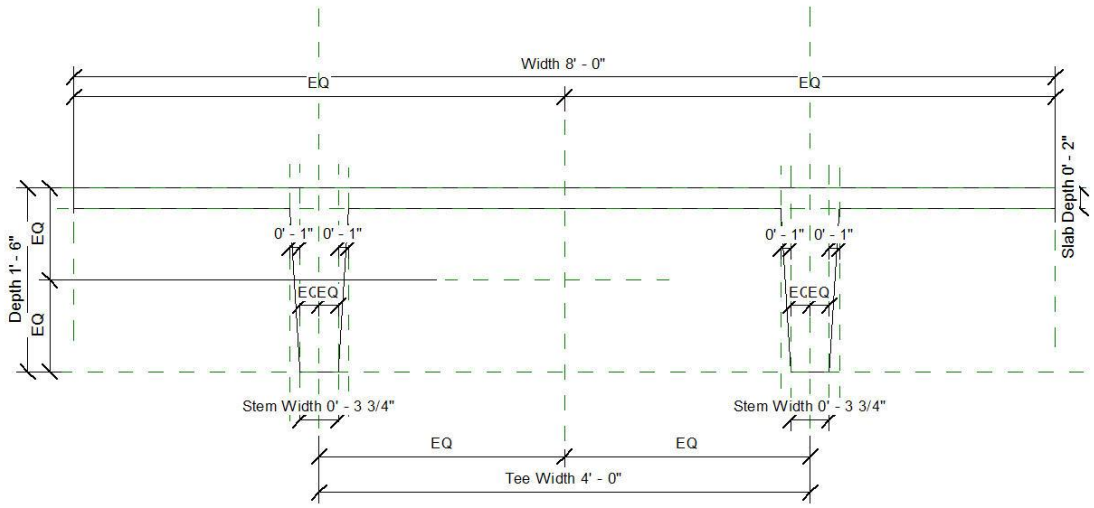


Figure 194: Precast-Double Tee Cross Section

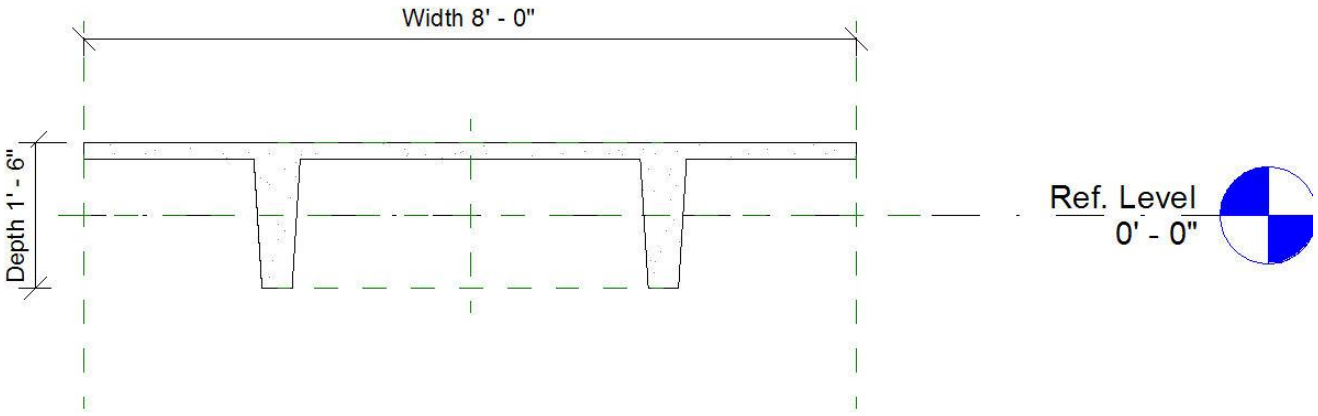


Figure 195: Precast-Double Tee Cross Section 2

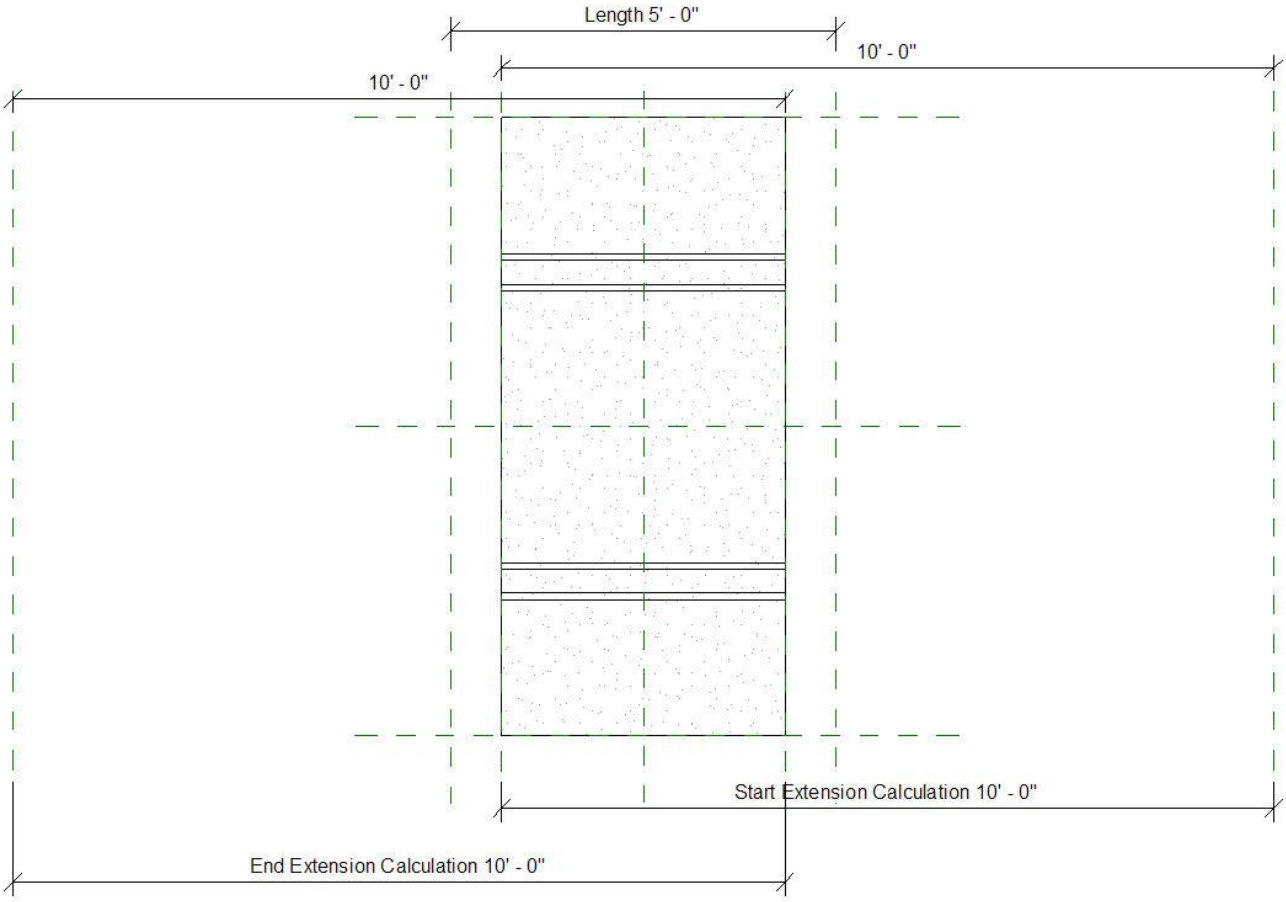


Figure 196: Precast-Double Tee

G. API User Interface Guidelines

Introduction

This section is intended to provide guidance and best practices for designing user interfaces for applications based on the Revit API. The following principles are followed by the Autodesk design and development staff when developing Revit and will provide a good starting point to the rest of this document.

Consistency

It is important for applications based on the API to provide a user experience that is consistent with Revit. This will allow the users of your application to leverage knowledge they developed while learning Revit. Consistency can be obtained by re-using certain dialog types, instead of creating or re-creating dialogs. See the dialog types section for examples of dialogs you can leverage when creating your application.

Speak the Users' Language

Understanding and communicating within the user's language is always critical, but particularly within a user interface. Users will need to provide as well as receive feedback from the application.

The tone used in user interface language should be informal, helpful, and consultative in tone. The user interface should politely provide information that is clear and informative to the user, and that can be acted upon accordingly with confidence.

Good Layout

A well-balanced layout of information can be achieved by following Gestalt Principles:

- PROXIMITY: items placed close together are perceived as being closely associated
- SIMILARITY: items that share similar appearance are perceived as being closely associated
- CONTINUITY: humans tend to prefer simple, unbroken contours over more complex, yet similarly plausible forms

Good Defaults

When a user needs to edit data or change a setting, the lack of any or obvious default can lead to errors and force users to re-edit and re-enter information. Remember to:

- Set the control value with a reasonable default value for the current context by consulting usage data or using the previously entered values by the user. The appropriate default may be blank.
- Where appropriate, remember the last used setting of the user instead of always presenting the same system default

A common example is pre-setting certain settings or options which would be the most often selected.

Progressive Disclosure

As an application's complexity increases, it becomes harder for the user to find what they need. To accommodate the complexity of a user interface, it is common to separate the data or options into groupings. The concept of Progressive Disclosure shows the needed information as necessary. Progressive Disclosure may be user-initiated, system initiated, or a hybrid of the two.

User initiated action

Examples here include a Show More button for launching a child dialog, TABS for chunking interface elements into logical chunks, and a COLLECTION SEARCH / FILTER for selectively displaying items in a collection by certain criteria.

System initiated action

These can either be:

- Event based: An event within the product initiates the disclosure of more information, such as with a **TASK DIALOG**.
- Time based: More information is disclosed after specified amount of time passes, such as in an automatic slide show.

Hybrid (user and time initiated)

An example of a hybrid is Progressive TOOLTIPS, where the user initiates the initial tooltip by hovering the mouse over the control, but after a set amount of time a more detailed tooltip appears.

Localization of the User Interface

If you plan to localize the user interface into languages other than English, be aware of the space requirements.

The English language is very compact, so translated text usually ends up taking up more space (30% on average for longer strings, 100% or more on short strings (a word or short phrase)). This can present problems if translated text is inserted into dialog boxes that were designed for an English product, because there is not usually sufficient space in available to fit the translated text. The common solution to this problem is to resize the dialog box so that the translated text fits properly, but most times this isn't the best solution.

Instead, by careful design of the dialog box by the developer, the same dialog box resource can be used for most if not all languages without the need for costly and time-consuming re-engineering. This paper tells you how to design 'global' dialog boxes.

These following design rules must be adhered to at all times to prevent globalization and localization problems.

- The English dialog must be at least 30% smaller than the minimum screen size specified by the product.
- A dialog must be designed with the following amounts of expansion in mind. This amount of extra space should look good in English and avoid resizing for most localization.

CHARACTERS	PERCENTAGE
1-5 characters	100%
6-10 characters	40%
11-100 characters	30%
100 characters or greater	20%

- Make the invisible control frame around text controls, frames etc. as large as possible to allow for longer translated text. These frames should be at least 30% larger than the English text.

Refer to the USER INTERFACE TEXT GUIDELINES FOR MICROSOFT WINDOWS USER EXPERIENCE GUIDELINES for additional information regarding localizing text.

Dialog Guidelines

Introduction

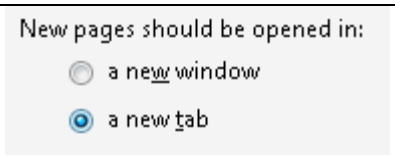
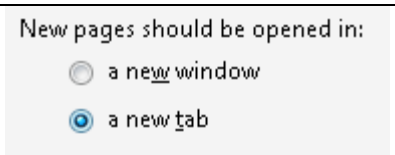
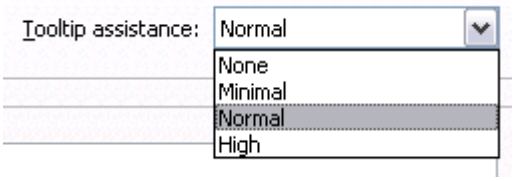
A dialog is a separate controllable area of the application that contains information and/or controls for editing information. Dialogs are the primary vehicle for obtaining input from and presenting information to the user. Use the following guidelines when deciding which dialog to use.

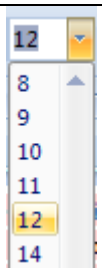
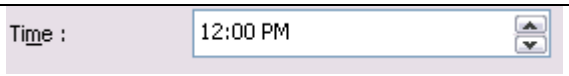
Dialog Type	Definition	Use When
Modal	• Halts the rest of the application and waits for user input	• Task(s) in the dialog are infrequent • It is acceptable to halt the system while the user enters data
Modeless	• User can switch between the dialog and the rest of the application without closing the dialog	• Task(s) in the dialog are frequent • Halting the system would interrupt the user workflow

Behavior Rules

- A dialog can either be user initiated (prompted by clicking a control) or system initiated (a warning triggered by a system event)
- The initial dialog location typically should be centered on the screen, but this rule may vary based on variation or on a particular context of use, but should have an immediate focus
- Dialogs should be resizable when the content is dynamic, please see DYNAMIC LAYOUT section for more on this topic

Dialog Controls

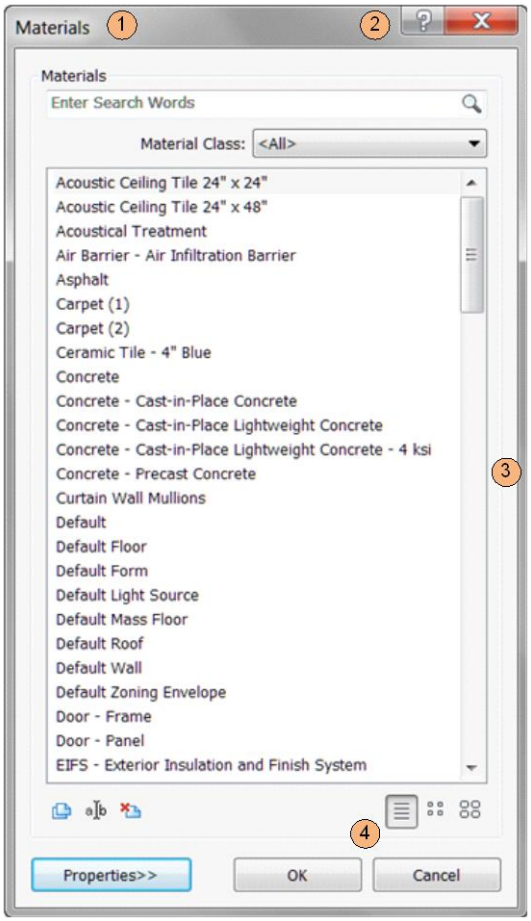
Control	Use When	Example
CHECK BOX	The choice being made (and it's opposite) can be clearly expressed to the user with a single label	 The opposite of enable is disable
RADIO BUTTON	There are between 2 - 5 mutually exclusive but related choices and the user can only make one selection per choice	
TEXT BOX	This choice requires manually entering a numerical text value	
DROP DOWN LIST	Select from a list of mutually exclusive choices It is appropriate to hide the rest of the choices and only show the	

	default selection Also, use this instead of radio buttons when there are more than four choices or if real estate is limited	
COMBO BOX	Similar to a drop-down list box, but allows the user to enter information not pre-populated in the drop-down	
SLIDER	Use when the expected input or existing data will be in a specific range. Sliders can also be combined with text boxes to give additional level of user control and give feedback as the slider is moved	
SPINNER	Use this option if the data can be entered sequentially and has a logical limit. This can be used in conjunction with an editable text box	

Laying Out a Dialog

Basic Elements

Every dialog contains the following elements:

	Element	Requirements	Illustration
1	**Title bar**	Title bars text describes the contents of the window.	
2	**Help button**	A button in the title bar next to the close button. • Help button is optional - use only when a relevant section is available in the documentation • Note that many legacy dialogs in Revit still have the Help button located in the bottom right of the dialog	
3	**Controls**	The bulk of a dialog consists of controls used to change settings and/or interact with data within an application. The layout of the controls should follow the LAYOUT FLOW, SPACING AND MARGINS, **Grouping**, and DIALOG CONTROLS sections outlined below. When an action control interacts with another control, such as a text box with a Browse button, denote the relationship by placing the *RELATED* controls in one of three places: • To the right of and top-aligned with the other control • Below and left-aligned with the other control. See CONTENT EDITOR • Vertically centered between related controls. See LIST BUILDER	
4	**Commit Buttons**	See the COMMITTING CHANGES section. The most frequently-used Commit buttons include: • OK • Cancel • Yes • No • Retry • Close	

Layout Flow

When viewing a dialog within a user interface, the user has multiple tasks to accomplish. How the information is designed and laid out must support the user in accomplishing their task(s). Keeping this in mind, it is important to remember users:

- Scan (not read) an interface and then stop when they get to what they were looking for and ignore anything beyond it
- Focus on items that are different
- Not scroll unless they need to

Lay out the window in such a way that suggests and prompts a "path" with a beginning, middle, and end. This path should be designed to be easily scanned. The information and controls in the "middle" of the path must be designed to be well balanced and clearly delineate the relationship between controls. Of course, not all users follow a strictly linear path when completing a task. The path is intended for a typical task flow.

Variation A: Top-Down

Place UI items that:

1. Initiate a task in the upper-left corner or upper-center
2. User must interact with to complete the task(s) in the middle
3. Complete the task(s) in the lower-right corner

In this example (Materials dialog), the user does the following:

1. Searches/filters the list
2. Selects an item
3. Commits or Cancels

Variation B: Left-Right

Place UI items that:

1. Initiate a task or change between modes in the far left (see TABS section)
2. User must interact with to complete the task(s) in the middle. This may be separated into separate distinct sections
3. Complete the task(s) in the lower-right corner

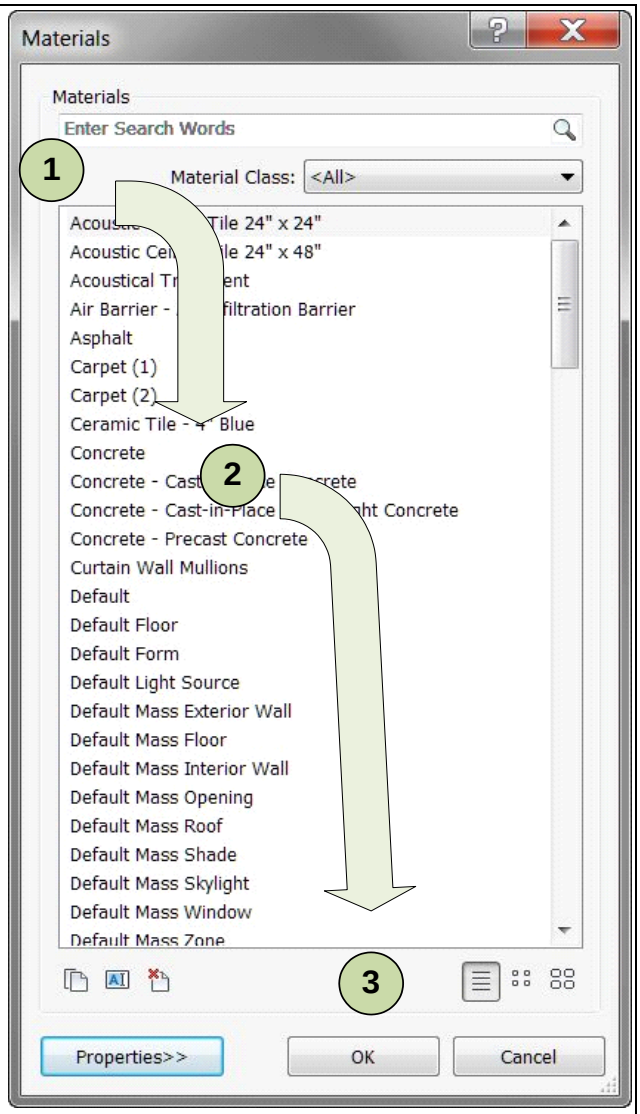


Figure 197 – Materials dialog

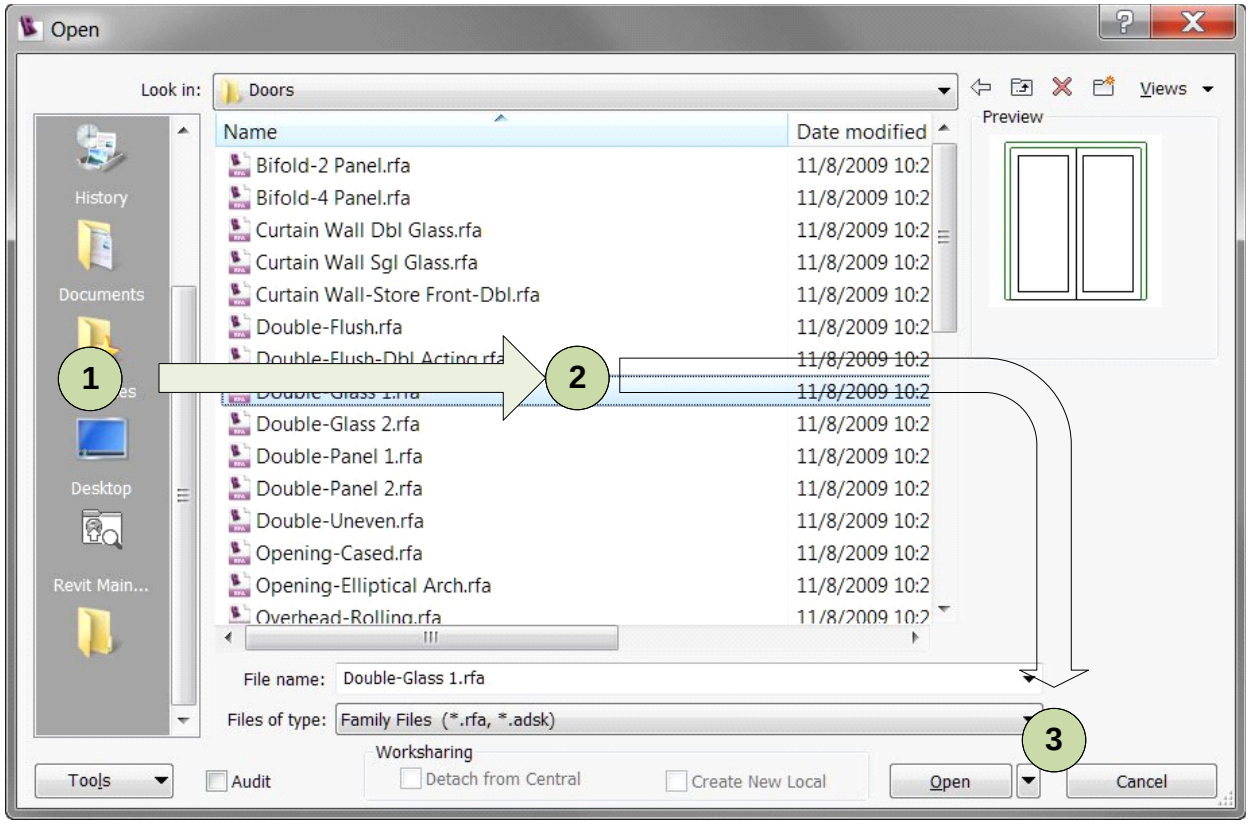


Figure 198 – Revit File Open dialog

Note: A top-down flow can also be used within this dialog, if they user is browsing the file hierarchy instead of the shortcuts on the far left.

Variation C: Hybrid

As seen in Variation B, many windows that are laid out left to right are actually a hybrid of left-right and top-down.

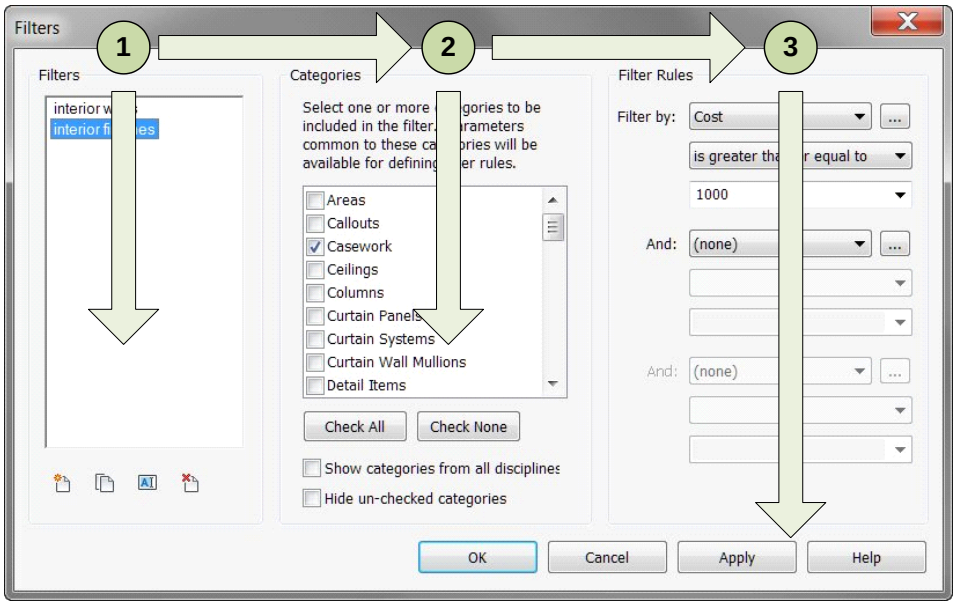


Figure 199 - Revit Filters dialog

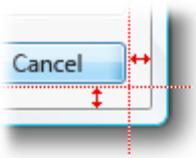
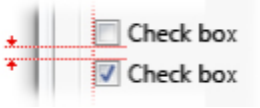
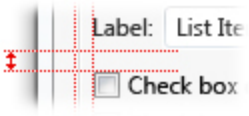
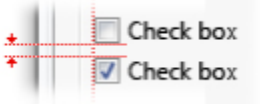
In this example (Revit Filters), the three primary tasks are grouped into columns, delineated by the group boxes: Filters, Categories and Filter Rules. Each of these columns contains a top-down flow that involves selecting from a collection of items or modifying a control.

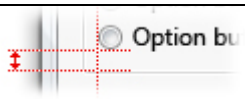
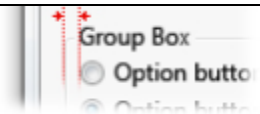
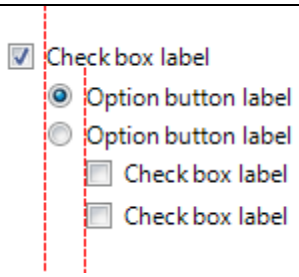
Spacing and Margins

The following table, taken from the LAYOUT SECTION OF THE MICROSOFT WINDOWS USER EXPERIENCE GUIDELINES lists the recommended spacing between common UI elements (for 9 pt. Segoe UI at 96 dpi). For a definition of the difference between dialog units (DLU) and relative pixels, see the LAYOUT METRICS section from the Microsoft guidelines.

TIP: Visual Studio makes it easy to follow these guidelines using a combination of Margin and Padding properties and the Snap lines feature. For more information, see WALKTHROUGH: LAYING OUT WINDOWS FORMS CONTROLS WITH PADDING, MARGINS, AND THE AUTO SIZE PROPERTY.

Spacing and Margins Table

Element	Placement	Dialog units	Relative pixels
	Dialog box margins	7 on all sides	11 on all sides
	Between text labels and their associated controls (for example, text boxes and list boxes)	3	5
	Between related controls	4	7
	Between unrelated controls	7	11
	First control in a group box	11 down from the top of the group box; align vertically to the group box title	16 down from the top of the group box; align vertically to the group box title
	Between controls in a group box	4	7
	Between horizontally or vertically arranged buttons	4	7

	Last control in a group box	7 above the bottom of the group box	11 above the bottom of the group box
	From the left edge of a group box	6	9
	Text label beside a control	3 down from the top of the control	5 down from the top of the control
	Between paragraphs of text	7	11
	Smallest space between interactive controls	3 or no space	5 or no space
	Smallest space between a non-interactive control and any other control	2	3
	When a control is dependent on another control, it should be indented 12 DLUS or 18 relative pixels, which by design is the distance between check boxes and radio buttons from their labels.	12	18

Grouping

Group boxes are the most common solution used to explicitly group related controls together in a dialog and give them a common grouping.

Figure 200 - Group box within a standard Print dialog

A group box should include:

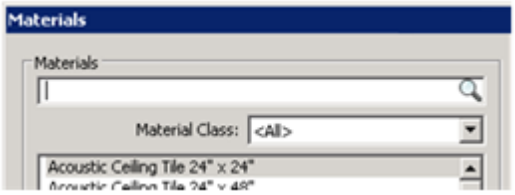
- Two or more related controls
- Exists with at least one other group box
- A label that:
  - describes the group
  - follows sentence style
  - is in the form of a noun or noun phrase
  - does not use ending punctuation
- A Spacing and Margins section describes spacing rules

Poor Examples- What Not to Use

The following are examples of what **should not** be done:

Group boxes without a label or a group box with only one control.

One group box in a dialog.



The (Materials) single group box title is redundant with the dialog title and can be removed.

Figure 201 – Group box with no label and group box with one control and Group box with title that is redundant with dialog title

Avoid "nesting" two of more group boxes within one another and placing Commit buttons inside a group box.

Horizontal Separator

An alternative to the traditional group box is a horizontal separator. Use this only when the groups are stacked vertically in a single column.

The following example is from Microsoft Outlook 2007:

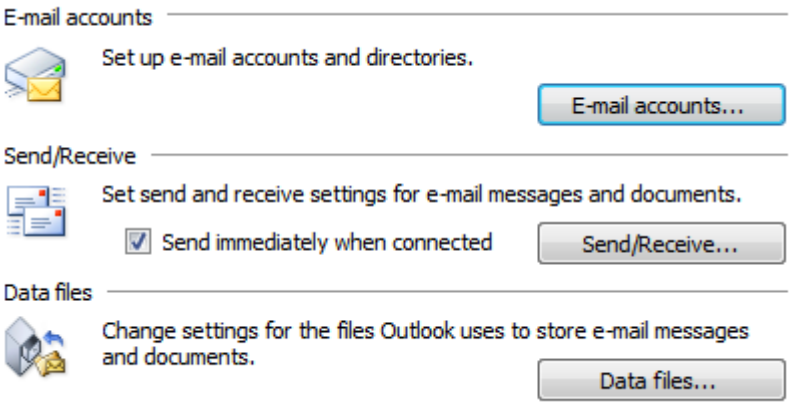


Figure 202 - Horizontal separators in Microsoft Outlook 2007

Spacing between the last control in the previous group and the next grouping line should be 12 DLUs (18 relative pixels).

Dynamic Layout

Content that is presented on different types or sizes of display devices usually requires the ability to adapt to the form that it is displayed in. Using a dynamic layout can help when environment

changes such as localizing to other languages, changing the font size of content, and for allowing user to manually expand the window to see more information.

To create a dynamic layout:

- Treat the content of the window as dynamic and expand to fill the shape of its container unless constrained.
- Add a resizing grip to the bottom right corner of the dialog.
- The dialog should not be resizable to a size smaller than the default size.
- The user defined size should be remembered within and between application sessions.
- Elements in the dialog should maintain alignment during resizing based on the quadrant they are described in the following table:

Home Quadrant	Alignment
1	Left and Top
2	Right and Top
3	Left and Bottom
4	Right and Bottom
Multiple	If control is located in multiple quadrants, it should anchor to quadrant 1 and/or 3 (to the left) and expand/contract to the right to maintain alignments

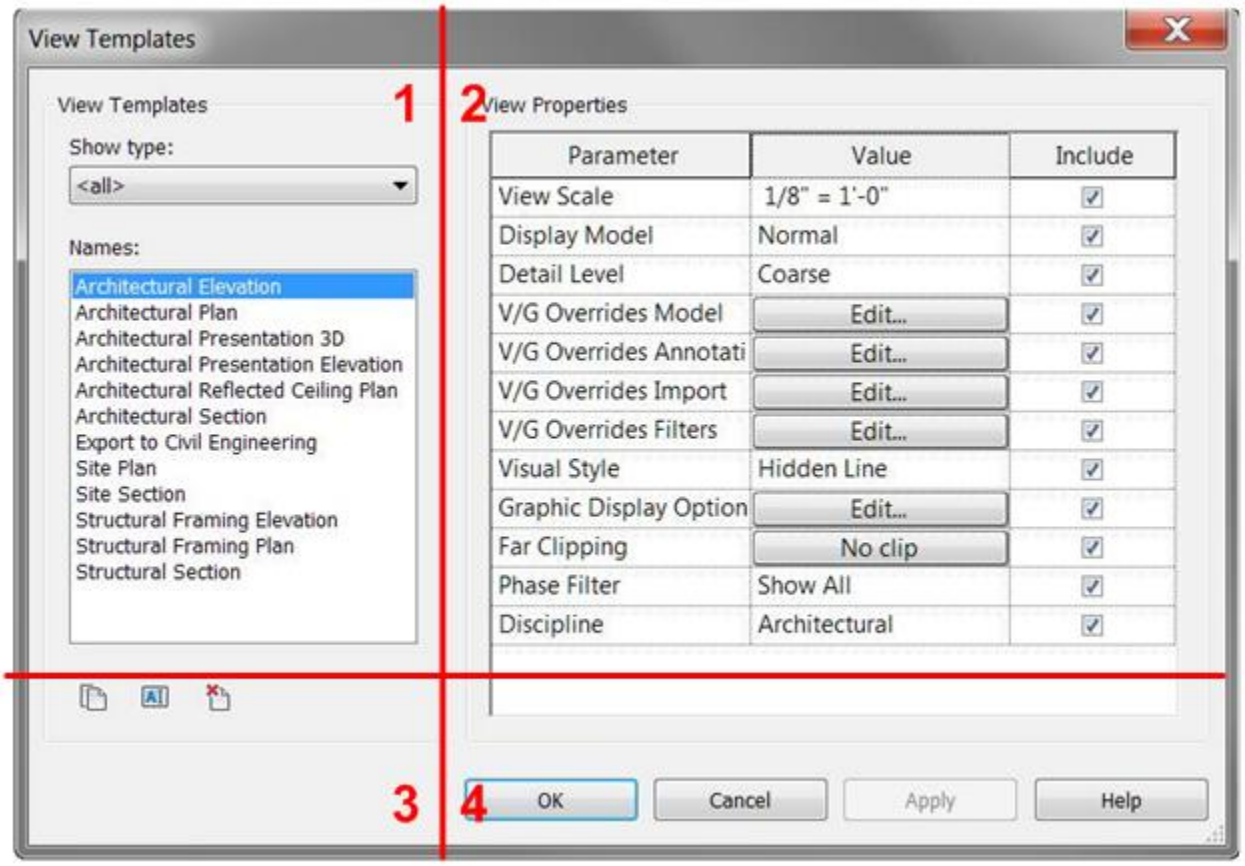


Figure 203 - Four square grid applied to Revit View Templates dialog to demonstrate how it should be resized

In this example, the list box is located in all four quadrants. So, it is anchored to the top-left and expands to the right and to the bottom.

Implementation Notes

Here are the some steps to consider when implementing a dynamic layout:

- Break the design on sections based on the structure of the content and flow you would like to achieve when container's size changes.
- Define the minimum, maximum and other size constrains for the various sections and control used. This is usually driven by the purpose of the data type we are presenting, images, supplemental information and controls.
- Consider alignment, that is, how the content will flow when re-sized.
- Consider which items should be static and which dynamic and how they are expandable – which should usually be for left-to-right languages, and dialogs should flow to the right and bottom, being anchored and aligned to the top and left.

For guidelines on how different controls should handle resizing, see the table below:

Control	Content	Re-sizable	Moveable
Button	Static	No	Yes
Link	Static	No	Yes
Radio Button	Static	No	Yes
Spin Control	Static	No	Yes, based on the element to which it is attached
Slider	Static	X Direction	Yes
Scroll Bar	Static	X Direction	Yes
Tab	Dynamic	X and Y Direction	Yes, but not smaller then the biggest control contained
Progressive Disclosure	Dynamic	X and Y Direction	Yes, but not smaller then the biggest control contained
Check Box	Static	No	Yes
Drop-Down List	Dynamic	X Direction	Yes but not smaller then the biggest text contained.
Combo Box	Dynamic	X and Y Direction	Yes, but not smaller then the biggest text contained
List View	Dynamic	X and Y Direction	Yes, but not smaller then the biggest text contained
Text Box	Dynamic	X and Y if multi-line	Yes
Date Time Box	Dynamic	X Direction	Yes, but not smaller then the biggest text contained
Tree View	Dynamic	X and Y Direction	Yes, but not smaller then the biggest text contained
Canvas	Dynamic	X and Y Direction	Yes
Group Box	Dynamic	X and Y Direction	Yes, but not smaller then the biggest control contained
Progress Bar	Static	X	Yes
Status Bar	Dynamic	X	Yes

Table or data grid	Dynamic	X and Y	Yes, table columns should grow proportionally in the X direction

***TIP**: For more detail on using a FlowLayoutPanel to build a resizable dialog, see:*
Walkthrough: Arranging Controls on Windows Forms Using a FlowLayoutPanel

Dialog Types

There are a handful of dialog types that persist throughout the Revit products. Utilizing these standard types helps to drive consistency and leverage users' existing learning and knowledge patterns.

Standard input dialog

This is the most basic dialog type. This should be used when the user needs to make a number of choices and then perform a discrete operation based on those choices. Controls should follow rules for

Grouping, SPACING AND MARGINS, and LAYOUT FLOW.

The Revit Export 2D DWF Options dialog is a good example of a Standard Input dialog.

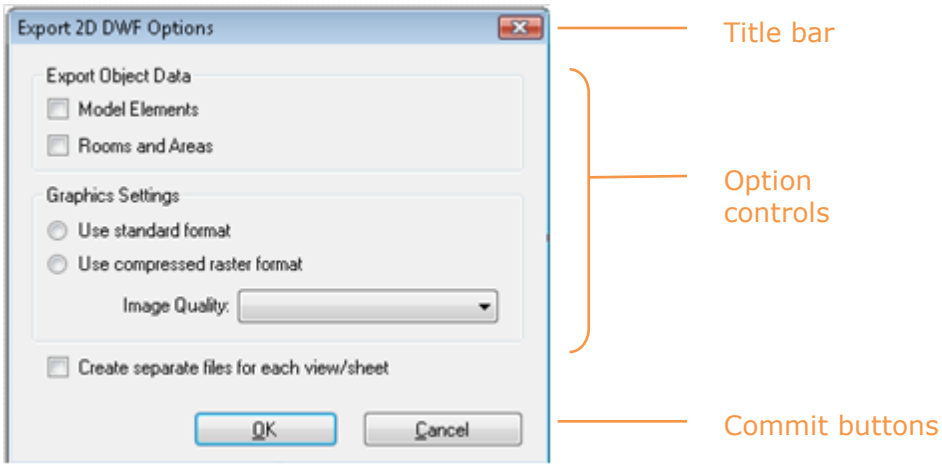


Figure 204 - The Export 2D DWF Options dialog

Property Editor

Use when an item's properties need to be modified by the user. To create a property editor, provide a TABLE VIEW that presents a name/property pair. The property field can be modified by a Text Box, Check Box, Command Button, Drop-Down List, or even a slider.

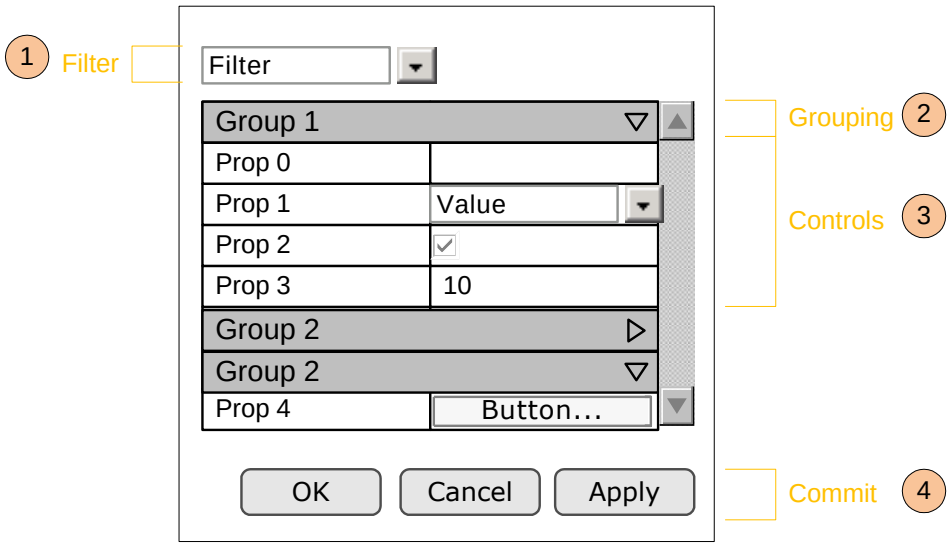


Figure 205 - Property Grid

Supported Behaviors

ID	Behavior	Description	Required
1	Filter	Filters the list of properties based on specified criteria	No
2	Grouping	Grouping the properties makes them easier to scan	No
3	Controls (Edit properties of an item)	Each value cell can contain a control that can be edited (or disabled) depending on the context	Yes
4	Commit (Buttons)	Optional, only use if within a modal dialog	No

Content Editor

If multiple action controls interoperate with the same content control (such as a list box), vertically stack them to the right of and top-aligned with the content control, or horizontally place them left-aligned under the content control. This layout decision is at the developer's discretion.

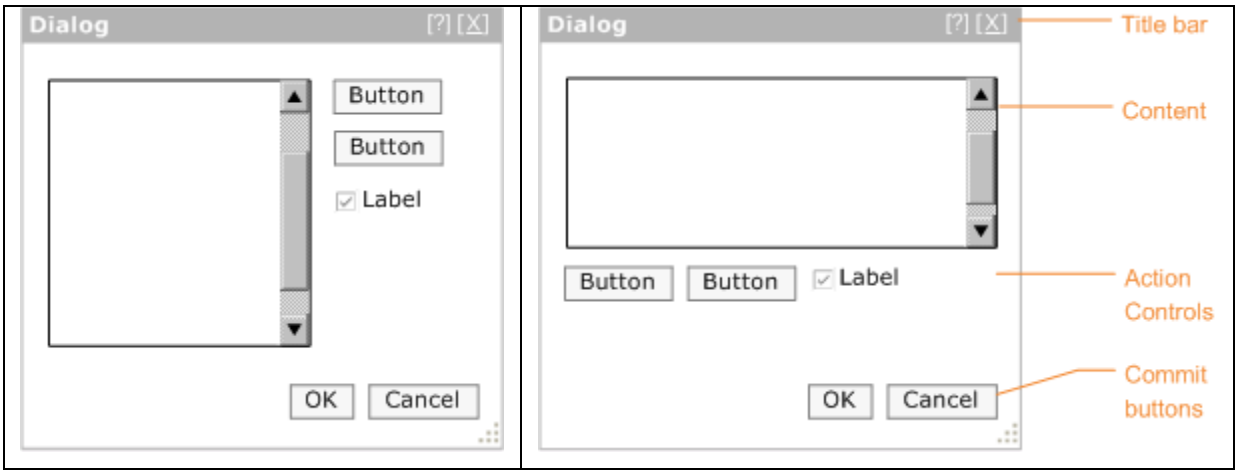


Figure 206 – Action controls to the right of content and action controls below content

Collection Viewer

In applications such as Revit, the user must view and manage collections of items to complete their tasks. The Collection View provides the user a variety of ways of viewing the items (browsing, searching and filtering) in the collection. Provide a way for a user to easily browse through a collection of items for the purpose of selecting an item to view or edit (see COLLECTION EDITOR). Optionally provide the ability to search and/or filter the collection.

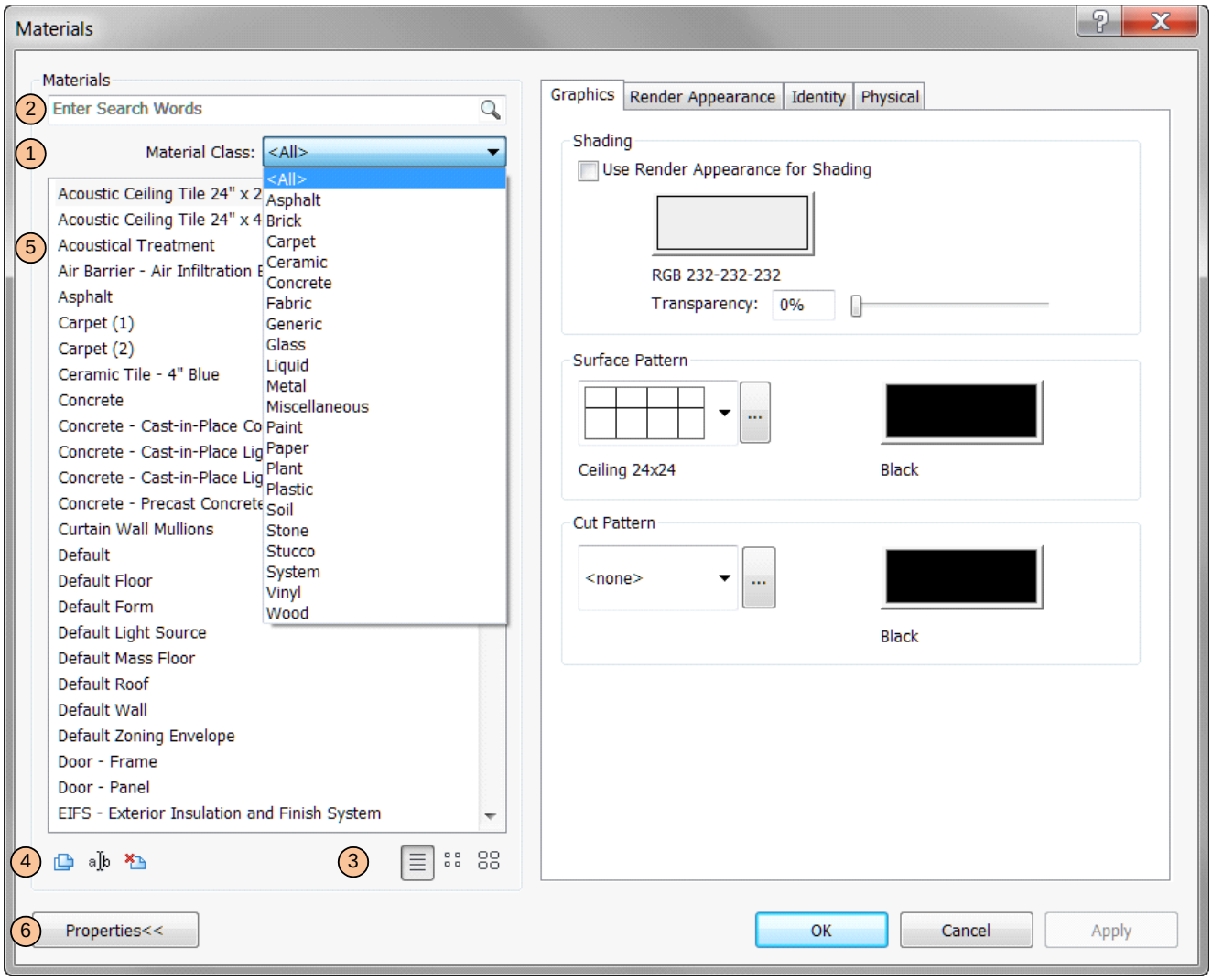


Figure 207 - Materials dialog from Revit

The example above shows Collection Viewer represented as a List. Table View and Tree View are also options for displaying the collection items.

Supported Behaviors

	Action	Description	Required
1	Filter	This provides options for filtering the list view. This can be represented as a: • Drop down - default criteria must be selected and should include "All" as an option • Check boxes - check box ON would refine the list. Default set by designer • Radio buttons - choosing between radio buttons refines the	No

		list. Default set by designer Once a choice is selected, the collection will automatically update based on the selected criteria. Controls must follow the MICROSOFT WINDOWS USER EXPERIENCE GUIDELINES	
2	Search Box	A search box allows users to perform a keyword search on a collection. The search box must follow MICROSOFT WINDOWS USER EXPERIENCE GUIDELINES	No
3	Change viewing mode	If the collection is viewed as list, the items can be optionally displayed with small or large icons instead of text	No
4	Collection Manager	• A separate UI will be provided to edit, rename, delete or add items to the collection. See Collection Editor • This is only displayed if managing the collection is user-editable	No
5	View the collection	The collection itself can be viewed in the following ways: List View, Table View, Tree View, or as a Tree Table	Yes
6	Show More	This button hides/shows the additional data associated with the currently selected item. See Show More **Button**	No

List View

When the user needs to view and browse and optionally select, sort, group, filter, or search a flat collection of items. If the list is hierarchical, use TREE VIEW or TREE TABLE and if the data is grouped into two or more columns, use TABLE VIEW instead.

Figure 208 - List View, showing different ways of presenting the data

There are four basic variations of this pattern that includes the following:

Drop down list

Use a drop down list box when you only need to present a flat list of names, only a single selection is required and space is limited. The MICROSOFT USER EXPERIENCE GUIDELINES should be followed when using a drop-down list.

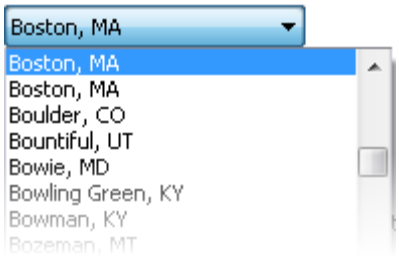


Figure 209 - Drop-down list

Combo box

Use a combo box when you want the functionality of the drop-down list box but also need to the ability to edit the drop-down list box. The MICROSOFT USER EXPERIENCE GUIDELINES should be followed when using this option, including for how to decide between drop-down and combo box.

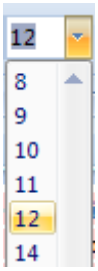


Figure 210 – The font selector in Microsoft Office 2007 is an example of a combo box

List box

Use when you only need to present a - list of names, it benefits the user to see all items and when there is sufficient room on the UI to display list box. Use also if selecting more than one option is required. The MICROSOFT USER EXPERIENCE GUIDELINES should be followed when using a list box.

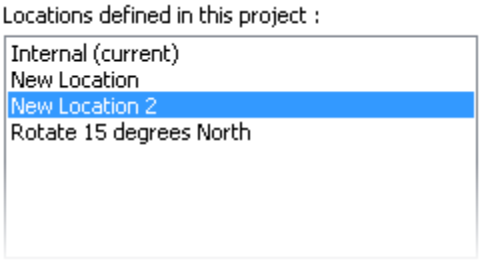


Figure 211 - List box

List View

Use when the data includes the option of list, details and/or graphical thumbnail previews, such as a Windows Open Dialog.

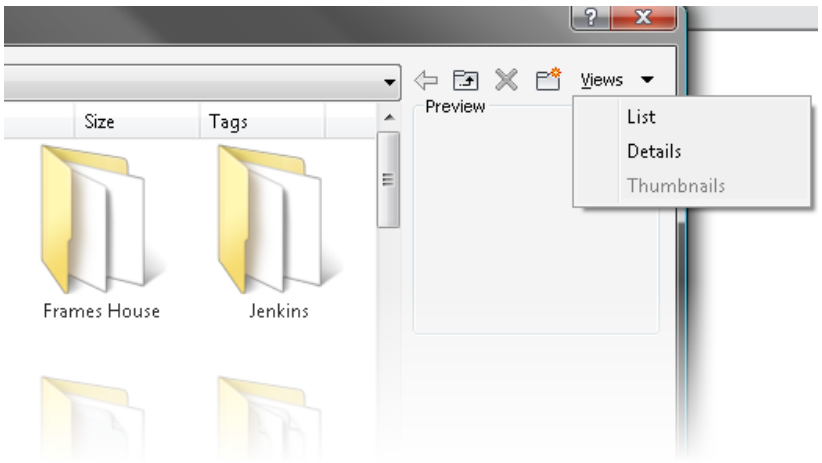


Figure 212 - List view

Table View

Users often need to view the contents of a collection so that they can easily comprehend and compare the attributes of many items at once. To accommodate this, present the user with a table that is formatted in such a way that is conducive to scanning.

Examples

Good Example - The following is an example of a well-formatted table.

Note the header cells are differentiated from the data cells and the alignment differs to makes it easier to scan the data.

Annual Design Conditions				
Threshold (%)	Cooling		Heating	
	Dry Build (°F)	MCWE (°F)	Dry Build (°F)	MCWE (°F)
0.1%	92.3	77.5	-6.0	-7.8
0.2%	90.3	72.3	-3.6	-5.3
0.4%	88.9	75.4	-1.3	-1.9
1.0%	88.5	73.5	2.7	-0.2
0.4%	88.9	75.4	-1.3	-1.9

Figure 213 - Good table example

Poor Example - The following is an example of a poorly formatted table.

Note the header cells are not differentiated from the data cells and the alignment makes it difficult to scan the data.

Annual Design Conditions				
Threshold (%)	Cooling		Heating	
	Dry Build (°F)	MCWE (°F)	Dry Build (°F)	MCWE (°F)
0.1%	102.3	77.5	-6.0	-7.8
0.2%	90.3	72.3	-3.6	-5.3
0.4%	88.9	75.4	-1.3	-1.9
11.0%	115..5	73.5	2.7	-0.2

Figure 214 - Bad table example

Table title and header cells

- Highlight and bold the table title to differentiate it from the data cells and the header cells
- For columns that are sort able, clicking the header sorts the collection. To differentiate table rows from each other, a different shade is used as background color for every second row. Keep the difference between the two colors to a minimum to preserve a gentle feeling. The colors should be similar in value and low in saturation - the one should be slightly darker or lighter than the other. It is often seen that one of the two colors is the background color of the page itself
- Ensure that the colors are different than header and title rows
- Title and header cells should be in title case

Columns containing numeric data

- Right align the column heading**s** for the data column
- Right (decimal) align data in numeric columns
- Format the values in percentage columns with percentage signs immediately to the right of the values to ensure that users are aware that the values are percentages

  NOTE: People can easily forget that they are looking at percentages so the redundancy is important here, especially for tables with many values

Columns containing numerical data

- Right align the column headings for the data column
- Right (decimal) align data in financial columns

Columns with a mix of positive and negative numeric data

Align the data so that decimals align

-1.23	(1.23)
0.03	0.03
-111.23	(111.47)

Figure 215 - Properly aligned numeric data

Columns containing only single letter or control (such as check box)

- Center the data or check symbol in this column
- Center the heading for the column

Columns with text that does not express numbers or dates

- Left align the column header of the number column
- Left align the text data
- Left align data that are not used as numbers like product IDs or registration numbers

Columns containing dates (treat dates as text)

- Left align the column header of a date column
- Left align the dates
- Include a date format in the column header if you are presenting to an international audience to avoid confusion

Column Sorter

Use a column sorter when users are viewing a collection (such as a large table), possibly spanning multiple pages, that they must scan for interesting values.

There are several meaningful possibilities for sorting the table and users may be more effective if they can dynamically change the column that is used for sorting the values on.

- Allow users to sort a collection of items by clicking on a column header
- As users click on the column label, the table is sorted by that column
- Another click reverses the order, which should be visualized using an up or down-wards pointing arrow
- Make sure it is visible which columns can be clicked on and which one is active now

Column header 1 ▼	Column header 2 ▲
a	3
b	2
c	1

Figure 216 - Column sorter

Tree View

Often a user may need to understand complex relationships within a hierarchy of items and this can often best displayed within a 'tree view.' The user may also need to select one or more of the items. If the collection is a flat list, use the LIST VIEW and if the data is grouped into two or more columns, use Table VIEW or TREE TABLE instead.

A tree UI follows the principle of user initiated PROGRESSIVE DISCLOSURE. Using a tree allows complex hierarchical data to be presented in a simple, yet progressively complex manner. If the data becomes too broad or deep, a search box should be considered.

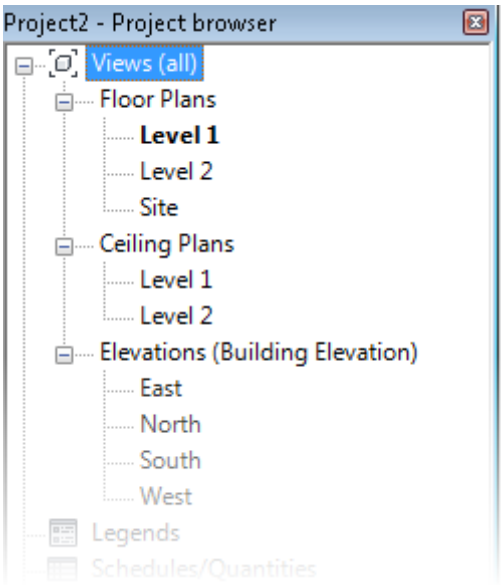


Figure 217 - The Revit Project Browser is a good example of a tree view

Tree Table

As with a Tree View, the user may need to view and browse a hierarchically organized collection of items with the intent of selecting one or more of the items. However, the user also needs to see more properties of the item than just the name. To accommodate this, present the user with a tree embedded within a table. Each row presents additional attributes of the item. Expanding a node exposes another row.

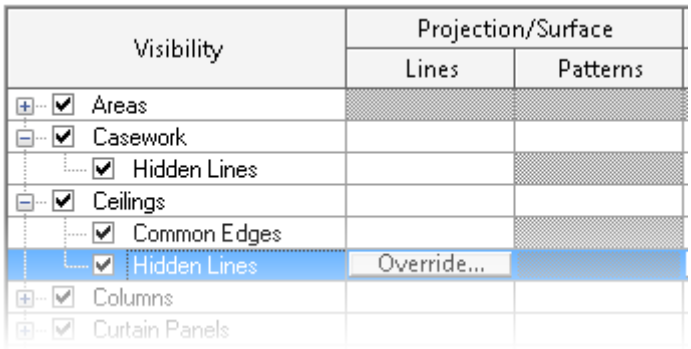


Figure 218 - The Revit Visibility/Graphics dialog is a good example of a Tree Table

Collection Search / Filter

When the user is viewing a collection with many items, they may need to filter the number of items. To accomplish this, provide a way for the user choose between either a system-provided list of filter criteria and/or user-creatable criteria. Selecting the criteria automatically filters the collection. The two most common ways are demonstrated in the Revit Materials dialog.

- A search box allows the list to be filtered based on a keyword
- A drop-down allows the list to be filtered based on a set of defined criteria

Collection Editor

In addition to viewing a collection of items, a user will also typically want to edit the collection. This can be accomplished by associating a toolbar for editing, creating, duplicating, renaming, and deleting items.

The Edit Bar

The buttons should be ordered left to right in the following order and with the following as tooltip labels: *Edit*, *New*, *Duplicate*, *Delete*, *Rename*. If a feature does not utilize one or more buttons, the rest move to the left.

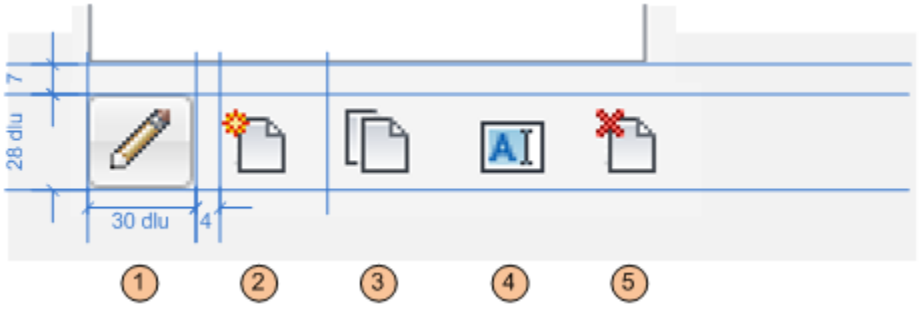


Figure 219 - The Edit Bar

	Action	Context
1	Edit	Use If an item can be edited. Editing an item may launch a separate dialog if there are less than three properties to edit. See the Collection Viewer section for more

		details on displaying collection items
2	New	Use New if the application is creating a new item
3	Duplicate	Use Duplicate if the feature can only duplicate existing items
4	Rename	Use Rename if the feature allows items to be renamed
5	Delete	Use Delete to remove the feature

To ensure that the primary UI element is placed first in the layout path, the following rules should be followed when placing the manage controls:

- Navigating list is primary task: place at the bottom-left of the list control
- Managing list is primary task: place at top left of list control
- When the main collection being managed is represented as a combo box: place to the right of the combo box

Add/Remove

A slight variation on the Edit Bar is the use of Add and Remove buttons, denoted by plus and minus icons, as shown below. Add and Remove is used when data is being added to an existing item in the model.

The following is a good example of a Collection Editor dialog that uses both forms of the Edit Bar. The Add (+) and Remove (-) buttons are used to add values to an already existing demand factor type.

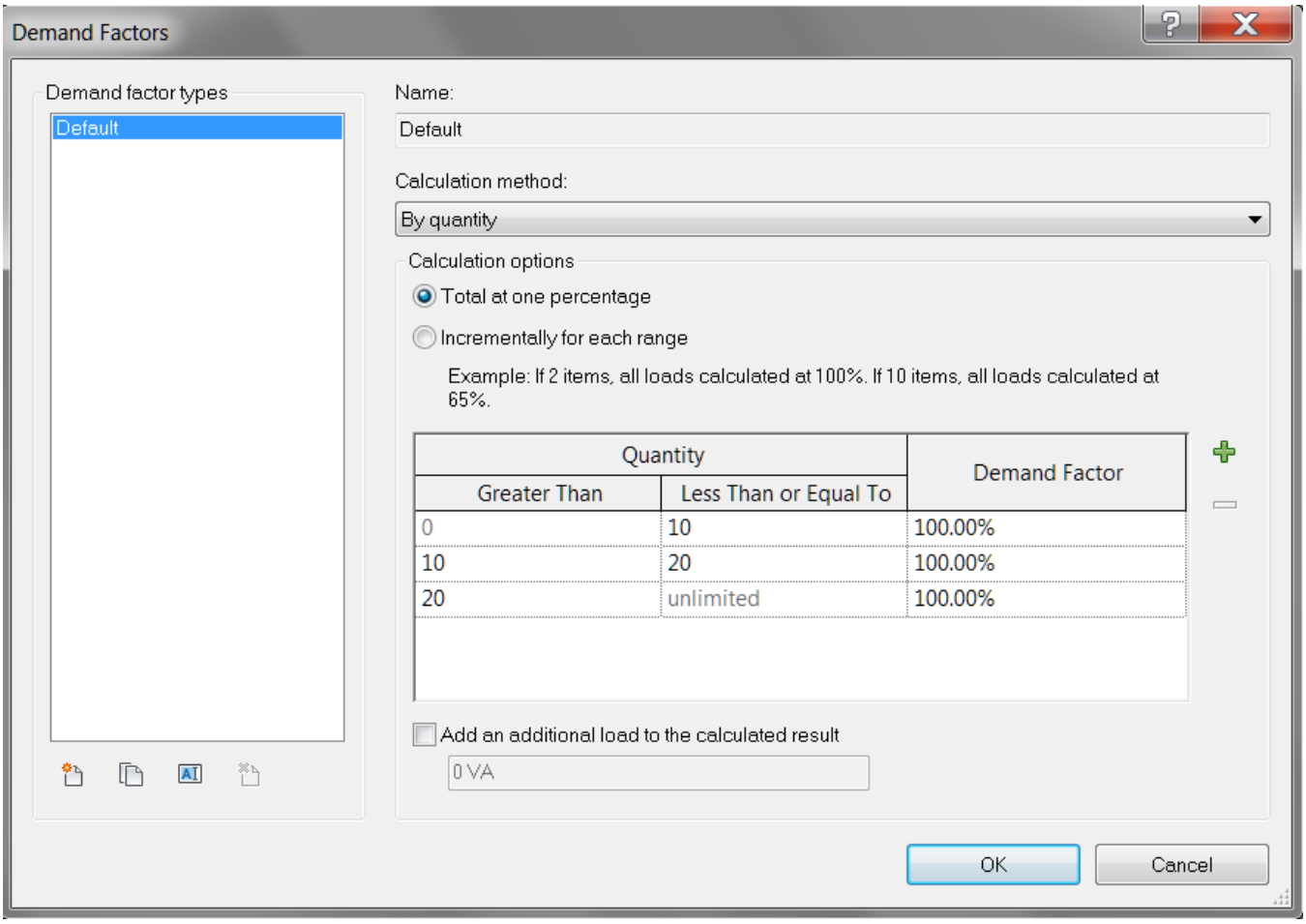


Figure 220 - Demand Factors dialog in Revit MEP 2011

List Builder

Use when there is a number of items that the user has to add/remove from one list to another. This is typically used when there is a list (located on the right) that needs to have items added to it from an existing list (located on the left.) Provide a List Box View of the two lists with button controls between, one for adding and the other for removing items.

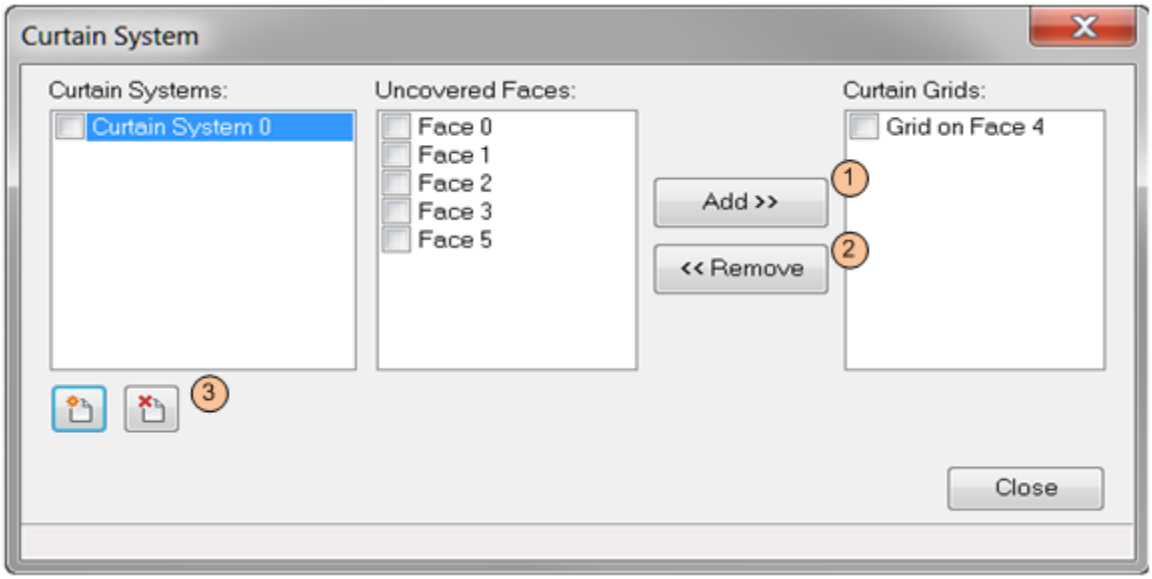


Figure 221 - The Curtain System SDK sample

Supported Behaviors

	Action	Description	Required
1	Add (to list)	Takes an item from list A and adds it to list B	Yes
2	Remove (from list)	Removes item from List B	Yes
3	Collection Editor	If List A can be managed in this context, use Collection Manager	No

Depending on the feature, the List Builder can act in one of two ways:

- Item can only be added once items from List A can only be added to List B once. In this case, the Add button should be disabled when a previously added item is selected in List A.
- Item can be added multiple times. In this case, the Add button is not disabled, and the user can add an item from List A to List B multiple times (the amount determined by the feature.) See Edit Label example below.

If the user needs to arbitrarily move an item up or down in a collection, provide up/down buttons next to the list.

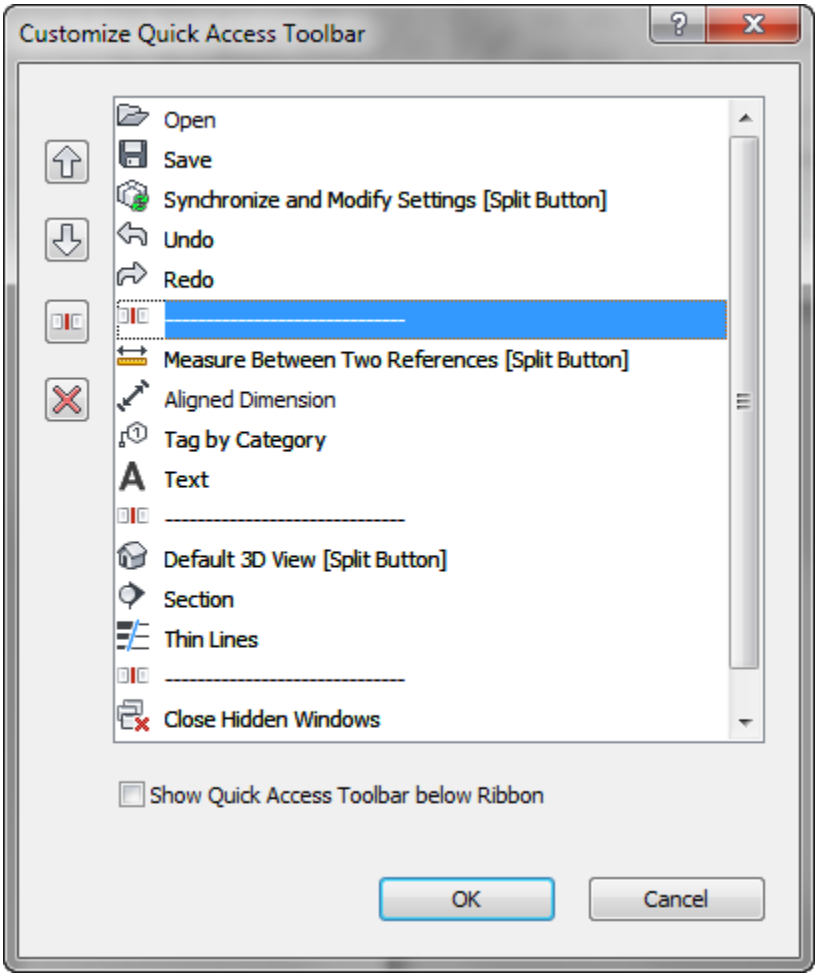


Figure 222 - Up/down buttons

Task Dialog

Task dialogs are a type of modal dialog. They have a common set of controls that are arranged in a standard order to assure consistent look and feel.

A task dialog is used when the system needs to:

- provide users with information
- ask users a question
- or allow users to select options to perform a command or task

The image below shows a mockup of a task dialog with all possible controls enabled. Most of the controls shown are optional and one would never create a task dialog that had everything on. The mockup below simple illustrates all the parts of a task dialog that could be utilized in one image.

Figure 223 - A task dialog with all components visible

Note that this particular task dialog would never happen in a real implementation. Only a small subset would ever be used at one time. Task dialogs cannot display other controls such as, text inputs, list boxes, combo boxes, check boxes, etc. They also only accommodate single step, single action operations; meaning a user may make a single choice and complete the task dialog operation. As a result any dialog that requires such additional controls or multiple steps operations (as with a wizard) are not task dialog candidates. They would need to be implemented as custom dialogs using .NET controls to have a similar look & feel to Task Dialogs.

The sections to follow explain when, where and how each task dialog component should be used to be consistent with others in Autodesk products.

General Design Principles

These are a few guiding principles that can be applied globally to task dialogs.

When reviewing the contents of a task dialog ask:

- Does it provide all the information needed to take informed action?
- Is the information too technical or jargon filled to be understood by the target user?

The following points apply to English language versions of product releases:

- Text should be written in sentence format - normal capitalization and punctuation. Titles and command button text are the exceptions, which are written in title format.
- Use a **single space** after punctuation. For example, DO NOT put two spaces after a period at the end of a sentence.
- Avoid the use of parentheses to address plurals. Instead, recast sentences. For example:

  Write "At least one referenced drawing contains one or more objects that were created in…" instead of "The referenced drawing(s) contains object(s) that were created in …"

- Include a copyright symbol © after any third party application called out in a task dialog.

Title (required)

All task dialogs require a title. Titles of task dialogs should be descriptive and as unique as possible. Task dialog titles should take the format of the following:

<featureName> - <shortTitle>

- Where <featureName> is the module from which the task dialog was triggered
- And <shortTitle> is the action that resulted in the task dialog being shown
- Examples:
  - **Reference Edit – Version Conflict**
  - **Layer – Delete**
  - **BOM – Edit Formula**

Where possible, use verbs on the second <shortTitle> part of the title such as Create, Delete, Rename, Select, etc.

In cases where there is no obviously applicable feature name (or several) applying a short title alone is sufficient.

A task dialog title should never be the product name, such as AutoCAD Architecture.

Title bar Icon

The icon appearing in the far left to the title bar should be that of the host application – this includes third party plug-ins. Task dialogs may contain plug-in names in the title to specify the source of the message, but the visual branding of all task dialogs should match the host application; such as Revit Structure, Inventor, AutoCAD Electrical, etc.

Main Instructions (required)

This is the large primary text that appears at the top of a task dialog.

- Every task dialog should have main instructions of some kind
- Text should not exceed three lines
- **[English Language Versions]** Main instructions should be written in sentence format - normal capitalization and punctuation
- **[English Language Versions]** Address the user directly as "you"
- **[English Language Versions]** When presented with multiple command link options the standard final line for the main instructions should be, "What do you want to do?"

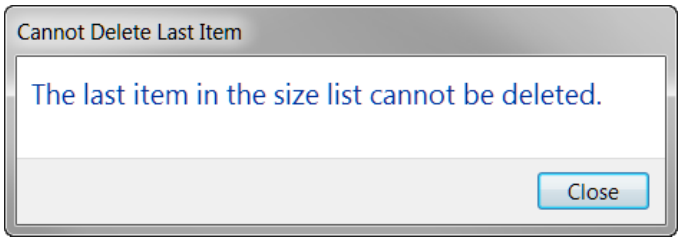


Figure 224 - A very simple task dialog with only main instructions for text

Main Content (optional – commonly used)

This is the smaller text that appears just below the main instructions.

- Main content is optional. It's primarily used when all the required instructions for a task dialog will not fit in the main instruction area
- Main content should not simply restate the main instructions in a different way, it should contain additional information that builds upon or reinforces the main instructions
- **[English Language Versions]** Main instructions should be written in sentence format (normal capitalization and punctuation)
- **[English Language Versions]** Address the user directly as "you" when needed

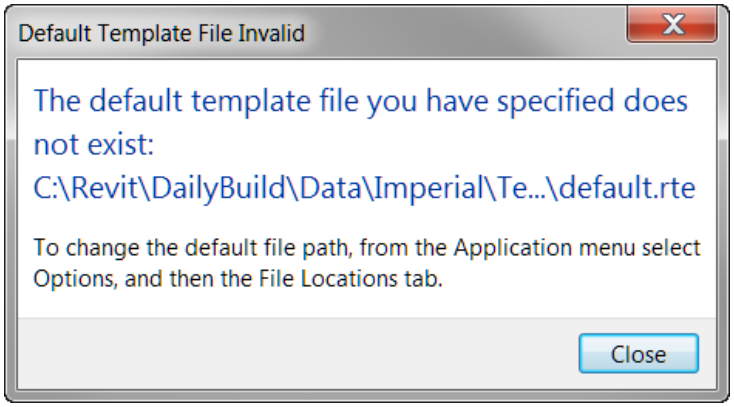


Figure 225 - A task dialog that uses both main instructions and main content

Expanded Content (optional – rarely used)

This text is hidden by default and will display at the bottom of the task dialog when the "Show" button is pressed.

- Expanded content is optional, and should be rarely used. It is used for information that is not essential (advance or additional information), or that doesn't apply to most situations
- **[English Language Versions]** Expanded content should be written in sentence format (normal capitalization and punctuation)
- **[English Language Versions]** Address the user directly as "you" when needed

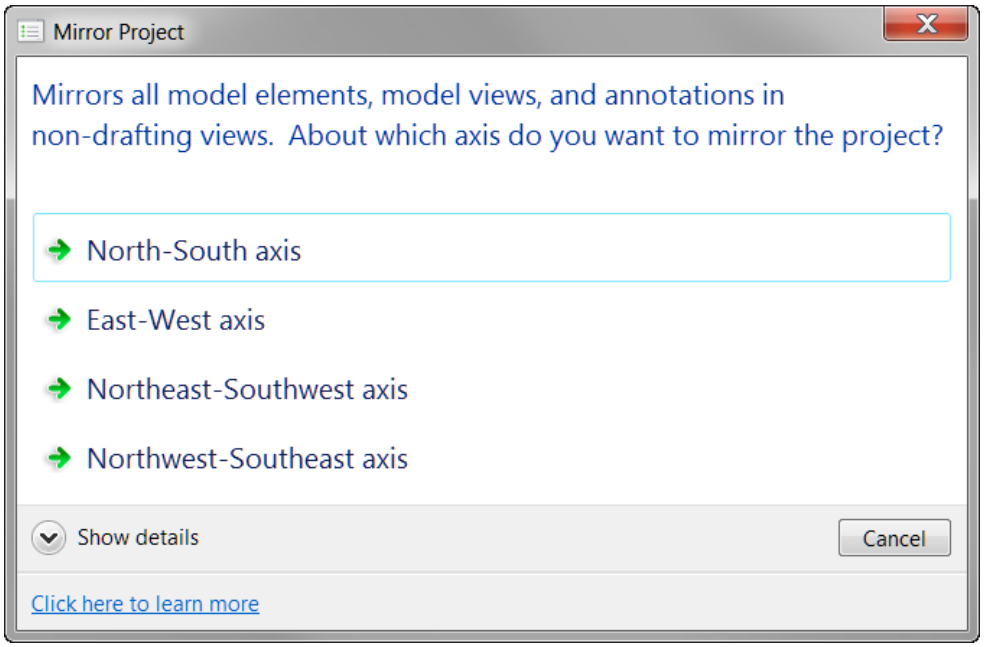


Figure 226 - The Show Details button displays additional Main Content text

Main Image (optional - low usage)

Task dialogs support the inclusion of an image to the left of the main instructions. Prior to task dialogs it has been common for most dialogs to have some sort of icon to show that the information it contained was informative, a warning, and error, etc.

Because images were used all the time the value of any image in a dialog was low.

For Autodesk products the warning icon (exclamation point in a yellow triangle) should only be used in situations where a possible action will be destructive in some way and likely cause loss of data or significant loss of time in rework.

A few examples include:

- Overwriting a file
- Saving to an older or different format where data may be lost
- Permanently deleting data
- Breaking associations between files through moving or renaming

This is only a partial list. With the exception of such situations **usage of a main image should be avoided**. See Figure 228 for an example of a Task Dialog with a warning icon.

"Do not show me again" (DNSM) Checkbox (optional)

Task dialogs support a "Do not show me again" checkbox that can be enabled on dialogs that users can opt to not see in the future. The standard wording for the label on this checkbox for English language versions is:

"Do not show me this message again"

Do not is not contracted to "Don't" and there is no punctuation at the end of the line.

For the single action the working should be "Always <action>" – for example

- If the action is "Save current drawing" the checkbox label would read "Always save current drawing"
- If the action is "Convert objects to linework" the checkbox label would read "Always convert objects to linework"

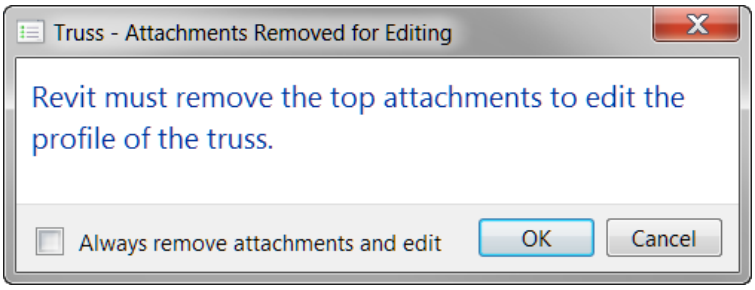


Figure 227 - Example of a task dialog using the DNSMA checkbox as an "Always..." checkbox for one choice

Where multiple are choices possible:

- The generic wording "**Always perform my current choice**" should be used
- Command links should be used to show the available choices. If buttons are used and a Cancel button is included it looks as though "cancel" is an option that could always be performed in future

Footer Text (optional)

Footer text is used to link to help. It replaces the Help or "?" button found on previous dialogs, and will link to the same location as an existing help link. On English language versions the text in the footer should read:

"Click here to learn more"

The text should be written as a statement in sentence format, but with no final punctuation. See Figure 226 for an example of a Task Dialog with footer text.

Progress Bar (optional – rarely used)

In instances where a task dialog is showing progress, or handling of an available option may take several seconds or more a progress bar can be used.

Command Links (optional – commonly used)

In task dialogs there are two ways a user can select an action – command links and commit buttons.

Command links are used in the following situations:

- More than one option is available (**avoid situations where only one command link is shown**)
- And a short amount of text would be useful in helping a user determine the best choice

Command links handle scenarios such as:

- Do A, B, or C
- Do A or B or A and B
- Do A or do not do A
- Etc

Command link text has two parts:

1. **Main content**: This is required for any command link. It is one line, written as a statement. For English language versions it is in sentence format without final punctuation.
2. **Supplemental content**: This is optional additional text to clarify the main content. For English language versions it is written in normal sentence format with final punctuation.

The first command link (one at the top) is the default action for the task dialog. It should be the most common action or the least potentially damaging action if no choice is substantially more likely than the other for the common use case.

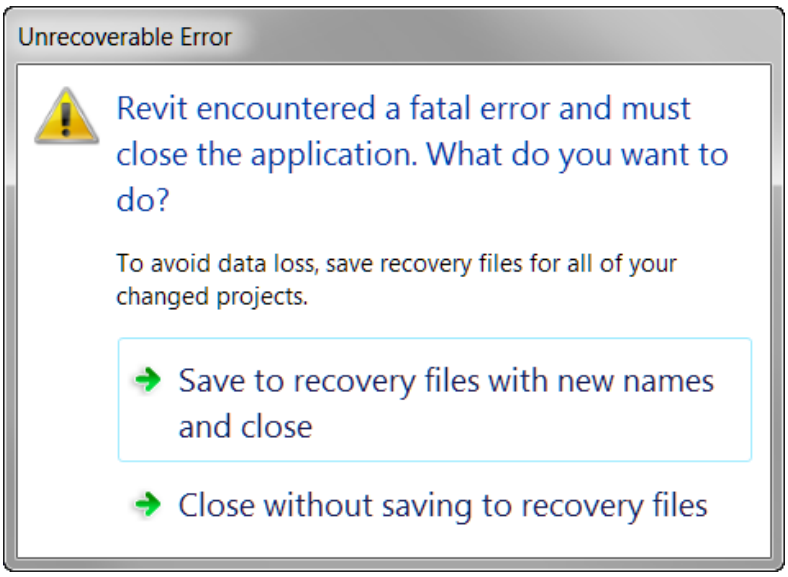


Figure 228 - A task dialog with two command links

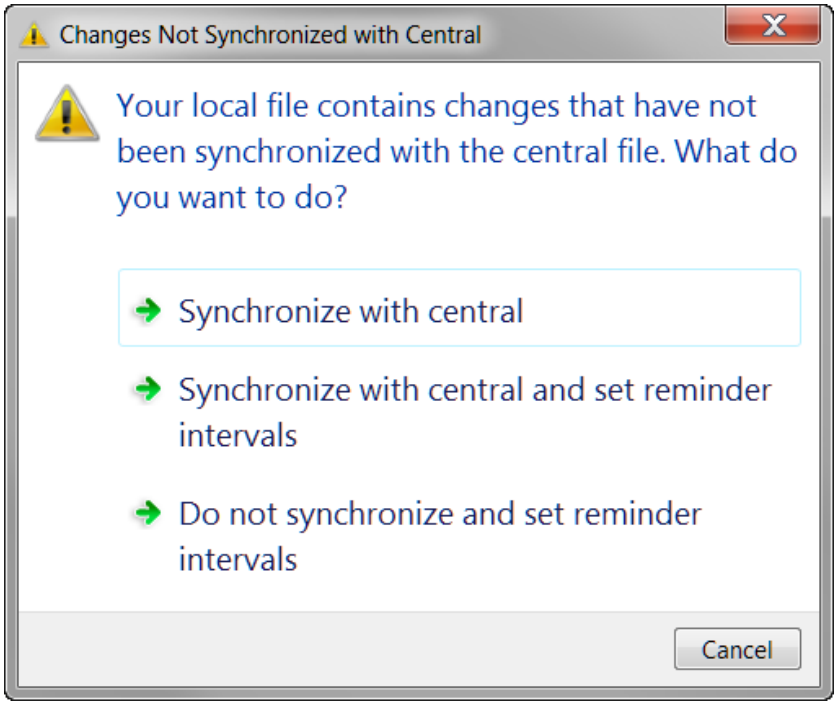


Figure 229 - Task Dialog with command links and a command button

Commit Buttons (optional – commonly used)

Commit buttons are simple buttons in the footer of the task dialog. Standard (English) terms include:

- OK
- Cancel
- Yes
- No
- Retry
- Close

It is possible to use custom text on commit buttons, but that is not recommended.

Notes on proper usage of each of the primary button types:

- The OK button should only be used in situations where a task dialog poses a question that can be answered by OK.
- The Cancel button should only be used when a task can truly be canceled, meaning the action that triggered the task dialog will be aborted and no change will be committed. It can be used in combination with other commit buttons or command links.
- The Yes & No button(s) should always be used in combination, and the text in the main instructions and / or main content should end in a yes / no question.
- The Retry button must appear with at least a Cancel button as an alternate option, so a user can choose not to retry.
- The Close button is used on any purely informational task dialog; i.e. where the user has no action to choose, but can just read the text and close the dialog.
  Previously the OK button was often used on such dialogs. It **should not** be used in task dialogs for this purpose.

The following are some examples of how commit buttons should be used:

- See Figure 226 for an example of a Cancel button with command links
- See Figure 224 for an example of a purely informative task dialog with a close button
- See Figure 227 for an example of a task dialog with OK and Cancel buttons

Default button or link

All tasks dialogs should have a default button or link explicitly assigned. If the task dialog contains an OK button, it should be the default.

Note: The exception is custom task dialogs with command links, which have actions that are equally viable, with none being “better” than the other, should not get assigned a default choice. All dialogs using only commit buttons must be assigned a default button.

Navigation

Tabs

Use when there are loosely related, yet distinct "chunks" of information need to exist within the same UI, but there is not enough room to display it all in a clear manner.

Separate the UI into distinct modal zones, each one represented by a "tab" with a descriptive label. The entire dialog should be treated as a single window with a single set of Commit buttons.

- All of the tabs should be visible at the same time
- Never have a single tab in a static UI such as dialog. Instead, use the tab's title for the page or dialog. Exception: if the number of tabs grows dynamically and the default is one. e.g. Excel's open workbook tabs
- Tabs are for navigation only. Selecting a tab should not perform any other action (such as a commit) besides simply switching to that page in the window
- Avoid nesting tabs within tabbed windows. In this case consider launching a child window
- Do not change the label on a tab dynamically based on interaction within the parent window

Variations

Variation A: Horizontal Tabs

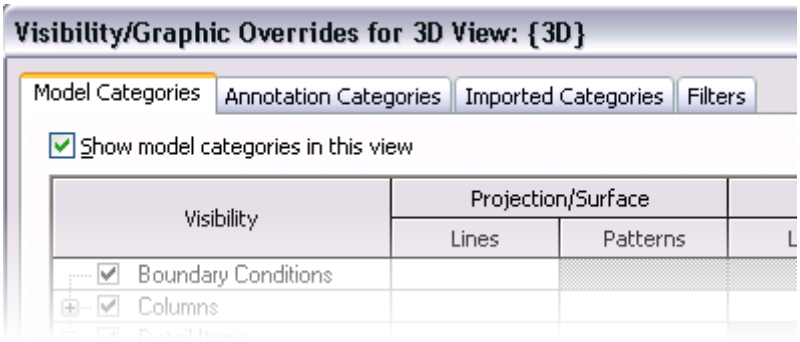


Figure 230 - Horizontal Tabs

Avoid more than one row of horizontal tabs. If a second row is needed, consider a vertical tab.

Variation B: Vertical Tabs

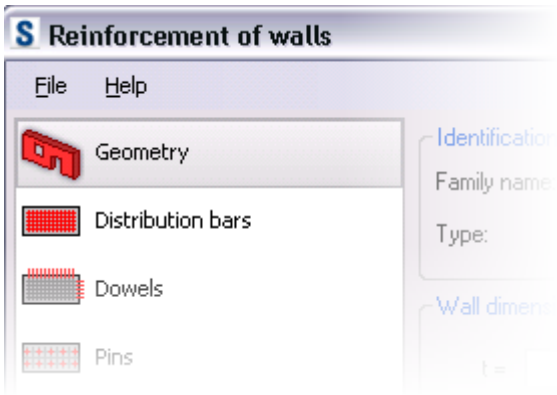


Figure 231 - Vertical Tabs

Vertical tabs are useful:

- In the Left-to-right Layout Flow
- If there are enough tabs that would force a second row in a horizontal layout

Keyboard Accessibility

Tab Order

Pressing the tab key cycles the focus between each editable control in the dialog. The general rule is left-to-right, top-to-bottom.

1. The default tab stop is at the control at the topmost, leftmost position in the dialog
2. Move right until there are no more controls in the current row
3. Move to the next row and start from the left-most control, moving right
4. Repeat step 2 until there are no more rows. Always end with the OK/Cancel/Apply row

- Right and left arrow, down and up arrows, Tab and Shift-tab all have the same behavior, respectively. Except for when the focus is on the following:
    - List control or combo box: The right/left, down/up arrows move the cursor down/up the list, respectively
    - Grid control: The right/left move the cursor right/left across columns, And down/up arrows move cursor down/up the list, respectively
    - Slider: The right/left, down/up arrows move the slider right/left, respectively
    - Spinner: The right/left, down/up arrows move the spinner down/up, respectively

- Treat each Group conceptually as a nested dialog, following the above rules within each Group FIRST and moving from the top-left Group, moving to the right until no more groups are encountered and then moving to the next row of Groups.

- If dialog is tabbed, default tab stop should be the default tab.

> ***TIP: Visual Studio can assist with the creation and editing of tab order by toggling the Tab Order visual helper (accessed from the View > Tab Order menu.)***

Access Keys

- Each editable control on a dialog should get a unique access key letter (which is represented by an underlined letter in the control's label)
- The user presses Alt key plus the assigned key and that control is activated as if it was clicked
- The default button does not require an access key since Enter is mapped to it
- The Cancel or Close button also does not need access key since Esc is mapped to it. See COMMITTING CHANGES for more detail

Show More Button

Following the principle of PROGRESSIVE DISCLOSURE, users may need a way of showing more data than is presented as a default in the user interface. The Show More button is typically implemented in one of two ways:

Expander Button: Provide a button with a label such as " << Preview " or "Show more >>" The double brackets >> should point towards where the new information pane will be presented. When opened, the double brackets should switch to indicate how the additional pane will be "closed."

See FIGURE 207 - MATERIALS DIALOG FROM REVIT for an example.

Dialog Launcher: A button with ellipses (…) that launches a separate dialog. This is typically used to provide a separate UI for editing a selected item.

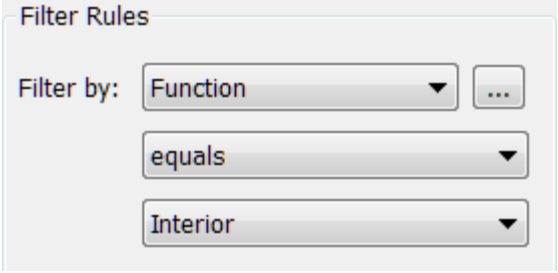


Figure 232 –Dialog launcher button, as implemented in the Revit View Filters dialog

Committing Changes

Modal dialogs are used to make changes to data within the project file. Use when there is an editor a series of edits have been queued up in a modal dialog or form and need to be committed at once. If the dialog is purely informational in nature, use a TASK DIALOG, which has its own committing rules.

Each modal dialog or web-form must have a set of commit buttons for committing the changes and/or canceling the task and/or closing the dialog.

Sizing

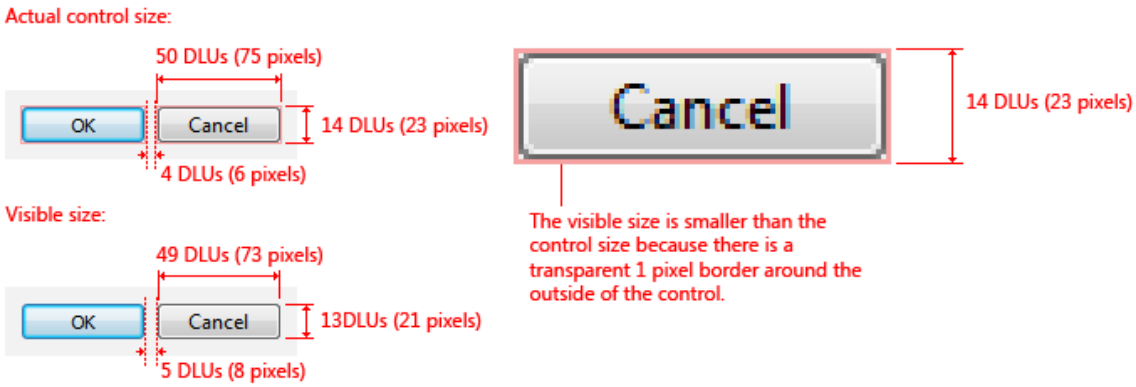


Figure 233 - Commit Button sizes (taken from Microsoft Windows User Experience Guidelines)

Layout

A summary of commit button styles for different window types

Pattern	Commit Button style
Modal Dialog	OK/Cancel or [Action]/Cancel
Modeless dialog	Close button on dialog box and title bar
Progress Indicator	Use Cancel if returns the environment to its previous state (leaving no side effect); otherwise, use Stop

Commit buttons should follow this layout pattern.

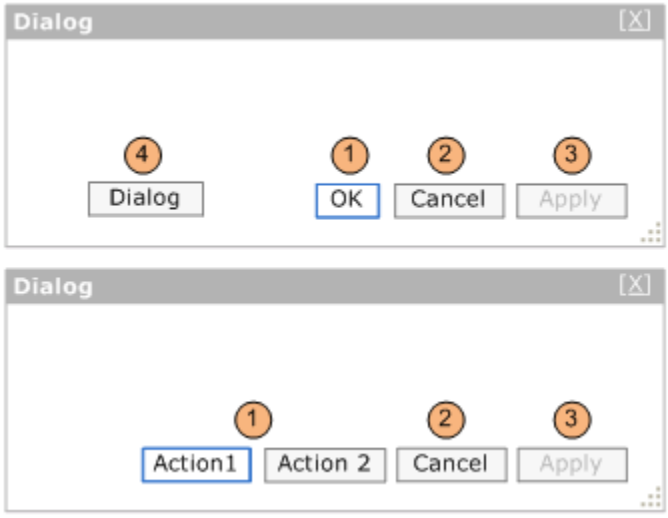


Figure 234 - Standard Commit Button layouts

	Button Type
1	Default (OK and other Action) buttons
2	Cancel or Close Button
3	Apply Button
4	Dialog buttons (optional)

Position the Default, Cancel, and Apply button(s) in this order and right aligned. The Dialog button(s) (if present) are aligned to the left, but to the right of the help button (if present).

Default (OK and other Action) buttons

The dialog must have a default action button. This button should be closely mapped to the primary task of the dialog. This can either be labeled OK or a more descriptive verb that describes the action.

- Make the button with less destructive result to be the Default button
- Enter key is the keyboard access point for the Default button

OK button

OK buttons can be used when saving a setting or series of settings. OK button rules:

- OK button should be used when it is the ONLY action (besides cancel) that can be committed from the dialog. Do not mix OK with other action buttons
- In modal dialogs, clicking OK means apply the values, perform the task, and close the window
- Do not use OK buttons to respond to questions
- Label OK buttons correctly. The OK button should be labeled OK, not Ok or Okay
- Do not use OK buttons in modeless dialog boxes. Use a action button or Close button

Action buttons

Action buttons have descriptive verbs that will be defined by the designer. Action button rules:

- Action buttons can be used to describe more clearly the action that will be taken when clicked
- One action button must be set as the default. This should be the action most closely mapped to the primary task of the dialog
- There can be one or more action buttons, but do not mix OK button with action buttons
- Use Cancel or Close button for negative commit buttons instead of specific responses to the main instruction
  Otherwise, if user wants to cancel, the negative commit would require more thinking than needed for this particular small task

Cancel or Close Button

- Verify the Close button on the title bar has the same effect as Close or Cancel
- Esc is the keyboard shortcut for Cancel or Close

Cancel button

- Cancel button should only be used when a task will be aborted and no change will be committed
- Clicking the Cancel button means abandon all changes, cancel the task, close the window, and return the environment to its previous state and leaving no side effect
- For nested choice dialog boxes, clicking the Cancel button in the owner choice dialog typically means any changes made by owned choice dialogs are also abandoned.
- Don't use Cancel button in modeless dialog boxes. Use Close button instead

Close Button

- Use Close button for modeless dialog boxes, as well as modal dialogs that cannot be canceled
- Clicking Close button means close the dialog box window, leaving any existing side effects

Apply button (optional)

Apply button will commit any changes made within the dialog on all tabs, pages, or levels within a hierarchy without closing the dialog. Optimally, the user will receive visual feedback of the applied changes. Here are some basic Apply Button rules:

- In modal or modeless dialogs, clicking Apply means apply the values, perform the task, and do not close the window
- In modeless dialog use Apply button only on those tasks that require significant or unknown upfront time to be performed, otherwise data change should be applied immediately
- The Apply button is disabled when no changes have been made. It becomes enabled when changes have been made
- Clicking cancel will NOT undo any changes that have been already committed with the Apply button
- Interacting with a child dialog (such as a confirmation) should not cause the Apply function to become enabled
- Clicking the Apply button after committing a child dialog (such as a confirmation message) will apply all the changes made previous to the action triggering the confirmation

Dialog Button (optional)

A dialog button performs an action on the dialog itself. Examples include: Reset and Tools for managing Favorites in an Open Dialog. They should be aligned to the far left of the dialog (to the right of the help button if present) and should never be the default.

Implementation Notes

- Keyboard Access - each commit button should have a keyboard access key mapped to it. The default button should be mapped to Enter
- The close button (whether it is Cancel or Close) should be mapped to Esc
- If Apply exists, and is NOT the default button, it should be mapped to Alt-A

Ribbon Guidelines

The following are aspects of the ribbon UI that can be modified by individual API developers. These guidelines must be followed to make your application's user interface (UI) compliant with standards used by Autodesk.

Ribbon Tab Placement

To make more room on the ribbon, third-party applications can now add ribbon controls to the Analyze tab as well as the Add-Ins tab.

- Applications that add and/or modify elements within Revit should be added to the Add-Ins tab.
- Applications that analyze existing data within the Revit model should be added to the Analyze tab.
- Applications MUST NOT be added to both the Add-Ins and Analyze tabs.

Contextual Tab Focus User Option

The Revit 2011 product line now contains a user option (located on the User Interface tab of the Options dialog) which allows users to choose whether or not to automatically switch to a contextual tab upon selection. This option is set to automatically switch by default. For some API applications, it may be favorable to have this option disabled, to prevent users from being switched away from the Add-ins or Analyze tab. In these cases, it is best to inform users of this option in the documentation and/or as informational text in the installer user interface.

Number of Panels per Tab

Each API application SHOULD add only one panel to either the Add-Ins tab.

Panel Layout

The following guidelines define the proper way to lay out a panel on the Add-ins tab. The following panel under General Layout provides an example to follow.

General layout

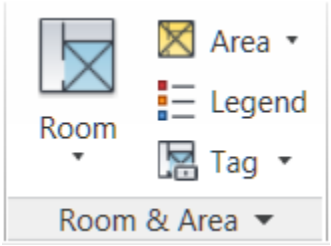


Figure 235 - Room & Area panel in the 2011 Revit products

A panel SHOULD have a large button as the left-most control. This button SHOULD be the most commonly accessed command in the application. The left-most button icon will represent the entire panel when it collapses (see PANEL RESIZING AND COLLAPSING below.) This button MAY be the only button in the group, or this button MAY be followed by a large button and/or a small button stack.

Panels SHOULD NOT exceed three columns. If more controls are necessary, use a drop-down button.

Panels SHOULD only contain controls for launching commands and controlling the application. Controls for managing settings or launching help and "about this application" should be located in a SLIDE-OUT PANEL.

Small button stack

- The stack MUST have at least two buttons and MUST NOT exceed three.
- The order of the small buttons SHOULD follow most frequent on bottom to least frequent on top. This is because the more frequently accessed command should be closer to the modeling window.

Panel Resizing and Collapsing

By default, panels will be placed left to right in descending order left to right based on the order in which they were installed by the customer. Once the width of the combined panels exceeds the width of the current window, the panels will start to resize starting from the right in the following order:

1. Panels with large buttons:
    a. Small buttons lose their labels, then:
    b. The panel collapses to a single large button (the icon representing the panel will be the first icon on the left.)
2. Panels with ONLY small button stack(s):
    a. Small buttons lose their labels and the panel label gets truncated to four characters and an ellipsis (three periods in a row.)
    b. If a small button stack is the left-most control in a panel, then the top button must have a large icon associated with it. This icon will represent the panel when collapsed.

The About button/link should be located within the main user interface and not on a ribbon panel.

Note: Panel resizing and collapsing is handled automatically by the ribbon component.

Ribbon Controls

Ribbon button

A Ribbon button is the most basic and most frequently-used control. Pressing a button invokes a command.

Ribbon buttons can be one of the three sizes:

- Large: MUST have a text label
- Medium: MAY have a text label
- Small: MAY have a text label

Radio Buttons

A radio button group represents a set of controls that are mutually exclusive; only one can be chosen at a time. These groups can be stacked horizontally (as seen in the justification buttons in the example below.)

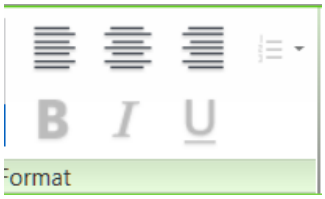


Figure 236 - The Format text panel from Revit 2011

Drop-down button

- The top label SHOULD sufficiently describe the contents of the drop-down list.
- Every item in the list SHOULD contain a large icon.
- A horizontal separator can be optionally added between controls. This should be used if the items are logically grouped under one button, but are separated into distinct groups.

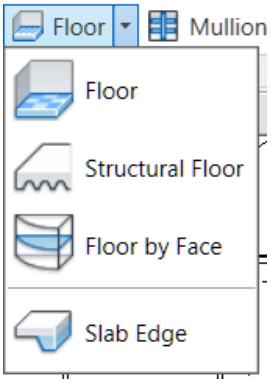


Figure 237 – Floor drop-down button in Revit Architecture

Split Button

A split button is a drop-down button with a default command that can be accessed by pressing the left side of the button. The right side of the button, separated by a small vertical separator, opens a drop-down list. The default command SHOULD be duplicated with the top command in the list.

A split button's default command can be *synchronized*. That is, the default command changes depending on the last used command in the drop-down list.

Combo Box and Text Box

The guidelines for combo boxes and text boxes in the ribbon are the same for those used within dialogs. See the DIALOG CONTROLS SECTION.

Slide-out Panel

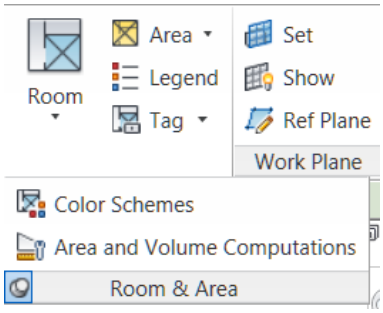


Figure 238 – Floor drop-down button in Revit Architecture

In general slide-outs should be used for commands relevant to the panel, but not primary or commonly used ones.

Each open panel can be optionally pinned open. Otherwise, once the mouse leaves the panel, it closes by itself.

Three suggested uses of slide outs are commands that launch settings dialogs related to the panel's task(s), a Help button, and an About button.

Vertical separator

A vertical separator MAY be added between a control or sets of controls to create distinct groupings of commands within a panel. A panel SHOULD have no more than two separators.

Icons

For proper icon design, see the icon design guidelines.

Text Usage

Button Labels

These guidelines are for English language only.

- MUST not have any punctuation (except hyphen, ampersand or forward slash)
- MUST be no longer than three words
- MUST be no longer than 36 characters
- MUST be Title Case; e.g. Show Mass
- The ampersand '&' MUST be used instead of 'and'. A space should appear before and after the ampersand
- The forward slash '/' MUST be used instead of 'or'. No spaces should appear before and after the slash
- Only large buttons MAY have two line labels but MUST NOT have more than two lines. Labels for all other controls MUST fit on a single line
- Button labels MUST NOT contain ellipses (…)
- Every word MUST be in capital case except articles ("a," "an," and "the"), coordinating conjunctions (for example, "and," "or," "but," "so," "yet," "with," and "nor"), and prepositions with fewer than four letters (like "in"). The first and last words are always capitalized

Panel Labels

These guidelines are English-only. All rules from the Command Labels section apply to Panel Labels in addition to the following:

- The name of the panel SHOULD be specific. Vague, non-descriptive and unspecific terms used to describe panel content will reduce the label's usefulness
- Applications MUST NOT use panel names that use the abbreviations 'misc.' or 'etc'
- Panel labels SHOULD NOT include the term 'add-ins' since it is redundant with the tab label
- Panel labels MAY include the name of the third party product or provider

Tooltips

The following are guidelines for writing tooltip text. Write concisely. There is limited space to work with.

Localization Considerations

- Make every word count. This is particularly important for localizing tooltip text to other languages
- Do not use gerunds (verb forms used as nouns) because they can be confused with participles (verb forms used as adjectives). In the example, "Drawing controls", drawing could be used as a verb or a noun. A better example is "Controls for drawing"
- Do not include lengthy step-by-step procedures in tooltips. These belong in Help
- Use terminology consistently
- Make sure that your use of conjunctions does not introduce ambiguities in relationships. For example, instead of saying "replace and tighten the hinges", it would be better to split the conjunction up into two simple (and redundant) sentences – "Replace the hinges. Then tighten the hinges"
- Be careful with "helping" verbs. Examples of helping verbs include shall, may, would have, should have, might have, and can. For example, can and may could be translated as "capability" and "possibility" respectively

- Watch for invisible plurals such as “object and attribute settings”. Does this mean “the settings for one object and one attribute” or “the settings for many objects and many attributes”?
- Be cautious about words that can be either nouns or verbs. Use articles or rewrite phrases like “Model Display” where model can be a noun or a verb in our software. Another example is “empty file”. It can mean “to empty a file” or “a file with no content”
- Be careful using metaphors. Metaphors can be subtle and are often discussed in the context of icons that are not culturally appropriate or understood across cultures. Text metaphors (such as “places the computer in a hibernating state”) can also be an issue. Instead, you might say “places the computer in a low-power state”

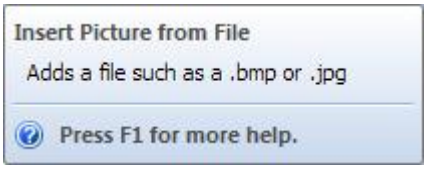
Writing/Wording Considerations

- Use simple sentences. The “Verb-Object-Adverb” format is recommended
- Use strong and specific verbs that describe a specific action (such as “tile”) rather than weak verbs (such as “use to…”)
- Write in the active voice (for example, “Moves objects between model space and paper space”)
- Use the descriptive style instead of the imperative style (“Opens an existing drawing file” vs. “Open an existing drawing file”)
- Make the tooltip description easily recognizable by using the third person singular (for example – “Specifies the current color” instead of “Specify the current color”)
- Don’t use slang, jargon, or hard to understand acronyms

Formatting Considerations

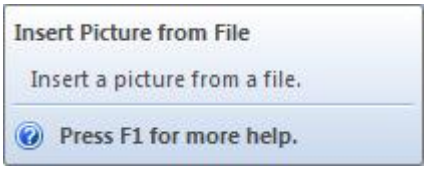
- Use only one space between sentences.
- Avoid repetitive text. The content in the tooltip should be unique and add value.
- Focus on the quality and understandability of the tooltip. Is the description clear? Is it helpful?
- Unless it’s a system variable or command, do not use bold. Although bold is supported in Asian languages, it is strongly recommended to avoid using bold and italics, because of readability and stylistic issues.
- Avoid abbreviations. For example, the word “Number” has many common abbreviations: No., Nbr, Num, Numb. It is best to spell out terms.

Good Example:



An example of a more useful descriptive sentence might be “Adds a file such as a .bmp or .jpg”. This provides more detailed information and gives the user more insight into the feature.

Poor Example:



In this example, the tooltip content repeats the tooltip title verbatim and does not add value to the tooltip. Additionally, if the translator cannot identify whether this string is a name/title or a descriptive sentence, it will be difficult for them to decide on the translation style.

As with other guideline issues, follow MICROSOFT GUIDELINES FOR TITLE AND SENTENCE CASE (listed below):

Title Case

- Capitalize all nouns, verbs (including is and other forms of to be), adverbs (including than and when), adjectives (including this and that), and pronouns (including its)
- Capitalize the first and last words, regardless of their parts of speech (for example, The Text to Look For)
- Capitalize prepositions that are part of a verb phrase (for example, Backing Up Your Disk)
- Do not capitalize articles (a, an, the), unless the article is the first word in the title
- Do not capitalize coordinate conjunctions (and, but, for, nor, or), unless the conjunction is the first word in the title
- Do not capitalize prepositions of four or fewer letters, unless the preposition is the first word in the title
- Do not capitalize to in an infinitive phrase (for example, How to Format Your Hard Disk), unless the phrase is the first word in the title
- Capitalize the second word in compound words if it is a noun or proper adjective, an "e-word," or the words have equal weight (for example, E-Commerce, Cross-Reference, Pre-Microsoft Software, Read/Write Access, Run-Time). Do not capitalize the second word if it is another part of speech, such as a preposition or other minor word (for example, Add-in, How-to, Take-off)
- Capitalize user interface and application programming interface terms that you would not ordinarily capitalize, unless they are case-sensitive (for example, The fdisk command)
- Follow the traditional capitalization of keywords and other special terms in programming languages (for example, The printf function, Using the EVEN and ALIGN Directives)
- Capitalize only the first word of each column heading

Sentence Case

- Always capitalize the first word of a new sentence
- Do not capitalize the word following a colon unless the word is a proper noun, or the text following the colon is a complete sentence
- Do not capitalize the word following an em-dash unless it is a proper noun, even if the text following the dash is a complete sentence
- Always capitalize the first word of a new sentence following any end punctuation. Rewrite sentences that start with a case-sensitive lowercase word

Common Definitions

Ribbon

The horizontally-tabbed user interface across the top of (the application frame in) Revit 2010 and later.

Ribbon Tab

The ribbon is separated into tabs. The Add-Ins ribbon tab, which only appears when at least one add-in is installed, is available for third party developers to add a panel.

Ribbon Panel

A ribbon tab is separated into horizontal groupings of commands. An Add-In panel represents the commands available for a third party developer’s application. The Add-In panel is equivalent to the toolbar in Revit 2009.

Ribbon Button

The button is the mechanism for launching a command. They can either be large, medium or small (Both large and small buttons can either be a simple push button or a drop-down button.

Menu button

The default first panel on the Add-Ins tab is the External Tools panel that contains one button titled “External Tools.” The External Tools menu-button is equivalent to the Tools > External Tools menu

in Revit 2009. Any External Command registered in Revit.ini under [ExternalCommands] will appear in this menu button.

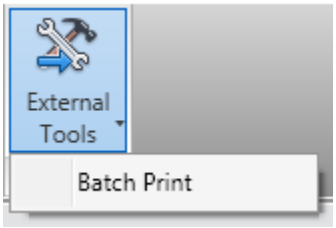


Figure 239 - External Tools menu-button on Add-Ins tab

Drop-down button

A drop-down button expands to show two or more commands in a drop-down menu. Each sub-command can have its own large icon.

Vertical Separator

A vertical separator is a thin vertical line that can be added between controls on a panel.

Tooltip

A tooltip is a small panel that appears when the user hovers the mouse pointer over a ribbon button. Tooltips provide a brief explanation of the commands expected behavior.

Terminology Definitions

Several words are used to signify the requirements of the standards. These words are capitalized. This section defines how these special words should be interpreted. The interpretation has been copied from INTERNET ENGINEERING TASK FORCE RFC 2119.

WORD	DEFINITION
MUST	This word or the term "SHALL", mean that the item is an absolute requirement
MUST NOT	This phrase, or the phrase "SHALL NOT", means that the item is an absolute prohibition
SHOULD	This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore the item, but the full implications must be understood and carefully weighed before choosing a different course
SHOULD NOT	This phrase, or the phrase "NOT RECOMMENDED", mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label
MAY	This word, or the adjective "OPTIONAL", means that the item is truly optional. One product team may choose to include the item because a particular type of user requires it or because the product team feels that it enhances the product while another product team may omit the same item