

REVIT 2025 API DEVELOPERS GUIDE

Jason Chang



2024-5-28

铁四院
杨园

1	Introduction.....	11
1.1	Getting Started.....	11
1.1.1	Welcome to the Revit Platform API.....	11
1.1.2	Using the Autodesk Revit API.....	16
1.1.3	Walkthroughs	29
1.1.4	Walkthrough: Hello World.....	30
1.1.5	Walkthrough: Add Hello World Ribbon Panel	36
1.1.6	Walkthrough: Retrieve Selected Elements	41
1.1.7	Walkthrough: Retrieve Filtered Elements	43
1.2	Add-In Integration.....	44
1.2.1	Overview	45
1.2.2	External Commands	46
1.2.3	External Application	53
1.2.4	Add-in Registration.....	56
1.2.5	Digitally Signing Your Revit Add-in.....	65
1.2.6	Localization.....	71
1.2.7	Attributes.....	72
1.2.8	Revit Exceptions.....	74
1.2.9	Ribbon Panels and Controls.....	75
1.2.10	Revit-style Task Dialogs.....	90
1.2.11	DB-level External Applications.....	94
1.3	Application and Document	94
1.3.1	Application Functions.....	95
1.3.2	Document Functions	100
1.3.3	Document and File Management.....	105
1.3.4	ForgeTypeld	110
1.3.5	Import Functions	111
1.3.6	Settings	113
1.3.7	Units	115

1.3.8	Cloud Models.....	120
1.4	Elements Essentials.....	124
1.4.1	Element Classification	124
1.4.2	Other Classifications.....	126
1.4.3	Element Retrieval	132
1.4.4	General Properties.....	133
2	Basic Interaction with Revit Elements.....	139
2.1	Filtering	139
2.1.1	Create a FilteredElementCollector	140
2.1.2	Applying Filters	140
2.1.3	Getting filtered elements or element ids	153
2.1.4	LINQ Queries	159
2.1.5	Bounding Box filters	161
2.1.6	Element Intersection Filters.....	161
2.2	Selection.....	163
2.2.1	Changing the Selection.....	164
2.2.2	User Selection.....	166
2.2.3	Filtered User Selection	170
2.3	Parameters.....	173
2.3.1	Walkthrough: Get Selected Element Parameters.....	175
2.3.2	Parameter	179
2.3.3	Definition.....	182
2.3.4	Built-In Parameters.....	184
2.3.5	Parameter Relationships.....	185
2.3.6	Shared Parameters	187
2.3.7	Global Parameters	201
2.4	Collections.....	215
2.4.1	Interface	216
2.4.2	Collections and Iterators	217

2.5	Editing Elements	219
2.5.1	Moving Elements.....	219
2.5.2	Copying Elements.....	223
2.5.3	Rotating elements	224
2.5.4	Aligning Elements	227
2.5.5	Mirroring Elements.....	227
2.5.6	Grouping Elements.....	229
2.5.7	Creating Arrays of Elements	231
2.5.8	Deleting Elements	233
2.5.9	Pinned Elements.....	235
2.6	Views	235
2.6.1	About views.....	235
2.6.2	View Graphics.....	240
2.6.3	View Types.....	248
2.6.4	Revisions.....	312
2.6.5	View Filters	320
2.6.6	View Cropping	327
2.6.7	Displaced Views.....	328
2.6.8	UIView	332
2.6.9	View Templates	333
2.6.10	Temporary Graphics	339
2.7	Transactions	341
2.7.1	Transaction Classes.....	341
2.7.2	Transactions in Events	350
2.7.3	Failure Handling Options	350
2.7.4	Getting Element Geometry and AnalyticalElement	351
2.7.5	Temporary transactions.....	354
3	Revit Geometric Elements.....	354
3.1	Walls, Floors, Ceilings, Roofs and Openings.....	354

3.1.1	Walls	354
3.1.2	Floors, Ceilings and Foundations.....	357
3.1.3	Roofs.....	361
3.1.4	Curtains	365
3.1.5	Other Elements.....	365
3.1.6	CompoundStructure.....	365
3.1.7	Opening	370
3.1.8	Thermal Properties.....	375
3.2	Family Instances	377
3.2.1	Identifying Elements.....	377
3.2.2	Family	379
3.2.3	FamilyInstances	379
3.2.4	Code Samples	394
3.2.5	FamilySymbol	401
3.3	Family Documents.....	405
3.3.1	About family documents	405
3.3.2	Creating elements in families	409
3.3.3	Visibility of family elements	417
3.3.4	Managing family types and parameters.....	419
3.4	Conceptual Design	426
3.4.1	Point and curve objects	426
3.4.2	Forms.....	431
3.4.3	Rationalizing a Surface	440
3.4.4	Adaptive Components	449
3.4.5	Create a .addin manifest file.....	450
3.5	Datum and Information Elements.....	451
3.5.1	Levels.....	451
3.5.2	Grids	454
3.5.3	Phase	458

3.5.4	Design Options	460
3.6	Annotation Elements	462
3.6.1	Dimensions and Constraints.....	462
3.6.2	Detail Curve	472
3.6.3	Tags.....	473
3.6.4	Text	476
3.7	FormattedText.....	478
3.8	Text Editor.....	481
3.9	Leaders	481
3.9.1	Annotation Symbol.....	482
3.9.2	Color Fill.....	483
3.10	Geometry	488
3.10.1	Example: Retrieve Geometry Data from a Wall.....	488
3.10.2	GeometryObject Class	491
3.10.3	Geometry Helper Classes	527
3.10.4	Collection Classes	542
3.10.5	Example: Retrieve Geometry Data from a Beam	543
3.10.6	Extrusion Analysis of a Solid	545
3.10.7	Finding geometry by ray projection	549
3.10.8	Geometry Utility Classes	557
3.10.9	Room and Space Geometry.....	558
3.11	Sketching	561
3.11.1	The 2D Sketch Class	564
3.11.2	3D Sketch.....	567
3.11.3	ModelCurve	576
3.12	Material.....	580
3.12.1	General Material Information	580
3.12.2	Material Management.....	586
3.12.3	Element Material.....	590

3.12.4	Material quantities	599
3.12.5	Painting the Face of an Element.....	600
3.13	Stairs and Railings	601
3.13.1	Creating and Editing Stairs.....	601
3.13.2	Railings	609
3.13.3	Stairs Annotations	613
3.13.4	Stairs Components	617
3.14	Surfaces.....	624
3.15	DirectShape	624
3.16	SubElements	640
4	Discipline-Specific Functionality.....	640
4.1	Architecture	640
4.1.1	Rooms.....	641
4.2	Civil Alignments API	657
4.3	MEP Engineering.....	658
4.3.1	MEP Element Creation	658
4.3.2	MEP Analytical Model	671
4.3.3	MEP Systems	672
4.3.4	Connectors	673
4.3.5	MEP Fabrication Detailing	676
4.3.6	Family Creation.....	680
4.3.7	Mechanical Settings.....	684
4.3.8	Electrical Analysis for Preliminary Design.....	691
4.3.9	Electrical Settings.....	695
4.3.10	Routing Preferences	698
4.4	Structural Engineering	700
4.4.1	Structural Model Elements.....	701
4.4.2	Analytical Model.....	751
4.4.3	Loads	755

4.4.4	Analysis Link	764
4.4.5	Analytical Links	766
4.4.6	Steel Fabrication.....	769
4.5	Toposolid	770
5	Advanced Topics.....	771
5.1	Analysis	771
5.1.1	Energy Data	771
5.1.2	Analysis Visualization.....	772
5.1.3	Detailed Energy Analysis Model	781
5.1.4	Path Of Travel	786
5.2	Browser Organization.....	789
5.3	Commands	789
5.4	Construction Modeling	795
5.4.1	Assemblies and Views	795
5.4.2	Parts.....	799
5.5	Context Menus.....	802
5.6	Dockable Dialog Panes	802
5.7	Dynamic Model Update	803
5.7.1	Implementing IUpdater	803
5.7.2	The Execute method.....	808
5.7.3	Registering Updaters	810
5.7.4	Exposure to End-User.....	811
5.8	Storing Data in the Revit model	813
5.8.1	Extensible Storage	813
5.9	Events.....	818
5.9.1	Database Events	818
5.9.2	User Interface Events.....	821
5.9.3	Registering Events	822
5.9.4	Canceling Events.....	825

5.10	Export.....	826
5.10.1	Export Tables	831
5.10.2	IFC Export	834
5.10.3	Custom export.....	834
5.11	External Events.....	837
5.12	Failure Posting and Handling.....	843
5.12.1	Posting Failures.....	843
5.12.2	Handling Failures	849
5.13	Linked Files.....	858
5.13.1	Revit Links.....	860
5.13.2	Managing External Files	874
5.14	Performance Adviser.....	879
5.15	Place and Locations.....	886
5.15.1	Place	886
5.15.2	City.....	888
5.15.3	ProjectLocation.....	888
5.15.4	Project Position.....	888
5.16	Point Clouds	895
5.16.1	Point Cloud Client.....	895
5.16.2	Point Cloud Engine	902
5.17	Transport Layer Security	903
5.18	Window Handle	904
5.19	Worksharing	904
5.19.1	Worksharing Overview	904
5.19.2	Worksets.....	905
5.19.3	Elements in Worksets.....	908
5.19.4	Editing Elements in Worksets.....	913
5.19.5	Opening a Workshared Document	918
5.19.6	Visibility and Display.....	930

5.19.7	Workshared File Management	939
6	Appendices	945
6.1	Glossary.....	945
6.1.1	Array	945
6.1.2	BIM	945
6.1.3	Class.....	946
6.1.4	Events	946
6.1.5	Iterator.....	946
6.1.6	Method.....	946
6.1.7	Namespace	946
6.1.8	Overloading	946
6.1.9	Properties	946
6.1.10	Revit Families.....	946
6.1.11	Revit Parameters	946
6.1.12	Revit Types.....	947
6.1.13	Sets	947
6.1.14	Element ID	947
6.1.15	Element UID.....	947
6.2	Hello World for VB.NET.....	947
6.2.1	Create a New Project.....	947
6.2.2	Add Reference and Namespace	948
6.2.3	Change the Class Name	950
6.2.4	Add Code	951
6.2.5	Create a .addin manifest file.....	952
6.2.6	Build the Program.....	953
6.2.7	Debug the Program	954
6.3	Material Properties Internal Units	955
6.4	Concrete Section Definitions.....	960
6.4.1	Concrete-Rectangular Beam.....	960

6.4.2	Precast-Rectangular Beam.....	963
6.4.3	Precast-L Shaped Beam	965
6.4.4	Precast-Single Tee.....	967
6.4.5	Precast-Inverted Tee	969
6.4.6	Precast-Double Tee.....	971
6.5	API User Interface Guidelines	973
6.5.1	Introduction.....	973
6.5.2	Consistency.....	973
6.5.3	Speak the Users' Language.....	973
6.5.4	Good Layout	974
6.5.5	Good Defaults.....	974
6.5.6	Progressive Disclosure	974
6.5.7	Localization of the User Interface.....	975
6.5.8	Dialog Guidelines.....	976
6.5.9	Ribbon Guidelines	1019
6.5.10	Common Definitions.....	1026
6.5.11	Terminology Definitions	1027
7	FAQ.....	1028

1 Introduction

This API Developer's Guide describes how to use the application programming interface (API) for Autodesk Revit.

Related Concepts

- Dynamo

1.1 Getting Started

The Revit Platform API is fully accessible by any language compatible with Microsoft .NET 8.0, 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#.

1.1.1 Welcome to the Revit Platform API

All Autodesk Revit-based products are Parametric Building Information Modeling (BIM) tools. Such a tool can be looked at as a CAD program that is used to build a 3D model rather than a set of individual drawing files. Autodesk Revit modeling is accomplished with real-world elements like columns, walls, doors and windows. The user can create views of the model, including plans, sections and callouts. All these views are directly generated from the 3D physical model so changes made in one view will automatically propagate through all other views. This process virtually eliminates the need to update multiple drawings and details when a change is made in the model.

The Autodesk Revit API is designed to reflect the same user interaction paradigms as the program's Graphical User Interface. Therefore, the first step to understanding the API is to learn how to use the program. If you are an Autodesk Revit novice, we suggest you first start by going through the Tutorials which you can access through the program's Help menu. You may also find it helpful to take a Training class from your local Autodesk reseller. This will help you quickly get up to speed with the program

Autodesk Resources:

- <https://www.autodesk.com/products/revit/overview>
- <https://www.autodesk.com/solutions/aec/bim>
- <https://forums.autodesk.com/t5/revit-api-forum/bd-p/160>

External Resources:

- <http://forums.augi.com/forumdisplay.php?93-Revit>
- <http://www.revitinside.com/>

1.1.1.1 *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.

Autodesk Revit offers an API designed to allow power users and external application developers to integrate their applications with Autodesk Revit. It is strongly recommended that you become familiar with Autodesk Revit and its features before attempting to use the Autodesk Revit API. Training can be found through the Autodesk Developer Network (ADN).

Learning Revit can help you:

- Maintain consistency with the Revit UI and commands.
- Design your add-in application seamlessly.
- Utilize 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 www.autodesk.com/revit.

1.1.1.2 What Can You Do with the Revit Platform API?

The following are general areas where the API is suitable:

- Creating add-ins and macros to automate repetitive tasks in the Autodesk Revit user interface
- Enforcing project design standards by checking for errors automatically
- Extracting project data for analysis and to generate reports
- Importing external data to create new elements or parameter values
- Integrating other applications, including analysis applications, into Autodesk Revit products
- Creating Autodesk Revit project documentation automatically

1.1.1.3 What You Will Need to Get Started

1. A working understanding of Autodesk Revit.
2. An installation of an Autodesk Revit-based product, including the Software Development Kit.
3. MS Visual Studio 2019 Community Edition (C# or VB.NET). Microsoft Visual Studio 2019 Professional is recommended, though, as Community editions do not support DLL debugging. Alternatively, you can use the built-in SharpDevelop development environment in Revit.
4. Some experience in a .NET based development language. (Autodesk Revit API examples are provided in C# and Visual Basic.NET.)

1.1.1.4 Installation

The Autodesk Revit API is automatically installed with the default installation of the Autodesk Revit based product. 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.

Note that additional API functionality exists in RevitAPIIFC.dll, RevitAPIMacros.dll, and RevitAPIUMacros.dll, which also install with Revit, but these assemblies are not immediately required to get started.

The Autodesk Revit API Software Development Kit (SDK) is installed from the Tools and Utilities section of the Autodesk Revit installation.

1.1.1.5 Development Requirements

The Autodesk Revit API requires Microsoft .NET 8.0.

To edit and debug your API applications, you need an interactive development environment such as Microsoft Visual Studio 2019 Professional or MS Visual Studio Community for C# or Visual Basic.NET. (Visual Studio Professional is recommended, as Express editions do not support DLL debugging.) When developing with the Autodesk Revit API, ensure that your project references two DLLs: **RevitAPI.dll** and **RevitAPIUI.dll** contained in the Autodesk Revit Program directory.

Some programming skills are required to effectively use the API. If you are a beginner in programming, we strongly advise you to learn Microsoft Visual Studio 2019 and one of the compatible languages like C# or Visual Basic.NET. There are many good books and classes to get you started.

Resources:

Online resources

- Free Visual Studio - <http://msdn.microsoft.com/VStudio/Express/>
- <http://www.codeguru.com/>
- <http://devx.com/>
- <http://www.msdn.microsoft.com/>
- [http://msdn.microsoft.com/en-us/library/aa288436\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/aa288436(v=vs.71).aspx) Books:
 - Code Complete, Second Edition, by Steve McConnell
 - Software Project Survival Guide, by Steve McConnell
 - Pro C# 5.0 and the .NET 4.5 Framework, by Andrew Troelsen

1.1.1.6 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.

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 2019 and four other walkthroughs covering primary add-in functions.

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.

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

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.

Walls, Floors, Ceilings, Roofs and Openings - Discusses Elements, their corresponding ElementType 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.

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.

[Transactions](#) - 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.

Discipline Specific

Revit includes discipline-specific features for architecture, structural engineering, and MEP engineering. Some APIs only work for specific feature sets.

[Architecture](#) - Discusses the APIs specific to the architectural features of Revit.

[Structural Engineering](#) - Discusses the APIs specific to the structural engineering features of Revit.

[MEP Engineering](#) - Discusses the APIs specific to the mechanical, electrical, and plumbing features of Revit.

Other

[Glossary](#) - Definitions of terms used in this document.

[Appendices](#) - Additional information such as Frequently Asked Questions, Using Visual Basic.NET for programming, and more.

[1.1.1.7 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.

Indexed Properties

Some Revit Platform API class properties are "indexed", or described as overloaded in the API help file (RevitAPI.chm). For example, the Element.Geometry property. 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 Element.Geometry(Options) property, you use Element.get_Geometry(Options).

[1.1.1.8 What's new in this release](#)

Please see the "What's New" section in RevitAPI.chm (included in the Revit API SDK) for information about changes and new features.

[1.1.2 Using the Autodesk Revit API](#)

Autodesk Revit SDK and Online Help

The Autodesk Revit API Software Development Kit (SDK) is installed from the Tools and Utilities section of the Autodesk Revit installation. In the SDK, there are example files that will help you get a better understanding of the API and its use. Each example file has a sample .addin manifest file with the information that you will need to edit and place into the appropriate folder, which Autodesk Revit will access on launch.

RevitAPI.chm is the Autodesk Revit API reference documentation help chm file, included with the SDK package in the \Revit 2018 SDK\ folder.

For more information:

- [Glossary of Autodesk Revit Terms](#)
- [FAQ](#)

Resources:

- www.autodesk.com/adn - Autodesk Developer Network home (ADN)
- www.autodesk.com/developrevit - Autodesk Revit development resources
- <http://forums.autodesk.com/t5/Revit-API/bd-p/160> - Revit API discussion group
- <http://thebuildingcoder.typepad.com/blog/> - The Building Coder, an ADN blog dedicated to Revit coding
- <http://bimapps.typepad.com/> - Bim Apps, a blog dedicated to BIM applications

1.1.2.1 Deployment Options

The Autodesk Revit API supports in-process DLLs only. This means that your API application will be compiled as a DLL loaded into the Autodesk Revit process.

The Autodesk Revit API supports single threaded access only. This means that your API application must perform all Autodesk Revit API calls in the main thread (which is called by the Autodesk Revit process at various API entry points), and your API application cannot maintain operations in other threads and expect them to be able to make calls to Autodesk Revit at any time. (For possible exceptions, see the Advanced Topic [External Events](#).)

There are two types of DLLs that you can create with the Autodesk Revit API, External Commands and External Applications.

External Commands

The Autodesk Revit API enables you to add new commands to the user interface of Autodesk Revit. These commands will appear in the Add-ins tab under the 'External Tools' pulldown, as seen in Figure 1. Through the API, external tool commands have access to the Autodesk Revit database, as well as the currently selected elements.

Figure 1: External Tool added to Revit

External Applications

The Autodesk Revit API enables you to also add external applications. These applications are invoked during Autodesk Revit startup and shutdown. They can create new panels in the Add-ins tab, as seen in Figure 2. They can also register handlers that can react to events occurring in the Autodesk Revit user interface.

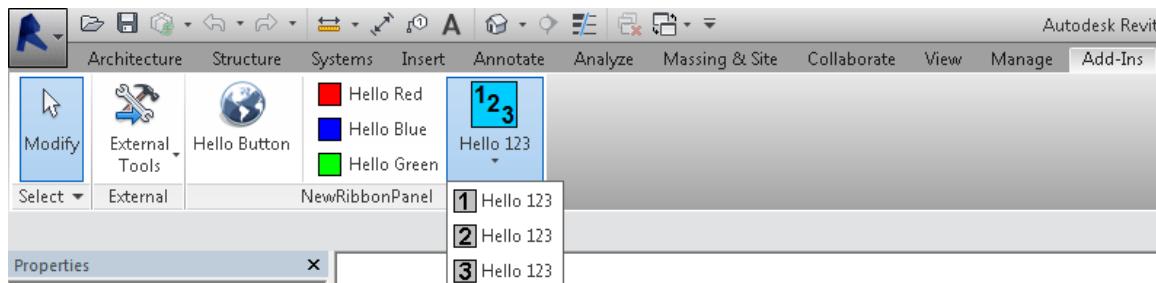


Figure 2: New panels and controls added to Revit

REX addins

REX (Revit Extensions) is an API framework that lets you build applications for Revit in .NET similar to classes that implement `IExternalCommand`. REX is meant to give you a more high-level development environment through built-in resources such as:

- Automatic dialog box creation and display
- Libraries to work with units and geometry
- Built-in command-based architecture to make menu and toolbar development easier.
- A standard mechanism for accessing a reference to the Revit application object.
- Automatic deployment and installation of addins for easy debugging. Please see the "\Revit 2018 SDK\REX SDK" folder for more details.

1.1.2.2 Registration of add-ins

The Revit API offers the ability to register API applications via a .addin manifest file.

Manifest files will be 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"**
- C:\ProgramData\Autodesk\Revit\Addins\<version number>\
- **In a user specific location in "application data"**
- C:\Users\<user>\AppData\Roaming\Autodesk\Revit\Addins\<version number>\

All files named .addin in these locations will be read and processed by Revit during startup.

A basic file adding one ExternalCommand looks like this:

Code Region: Basic manifest file for an ExternalCommand

```
<?xml version="1.0" encoding="utf-16" 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>

</AddIn>

</RevitAddIns>
```

A basic file adding one ExternalApplication looks like this:

Code Region: Basic manifest file for an ExternalApplication

```
<?xml version="1.0" encoding="utf-16" standalone="no"?>

<RevitAddIns>

<AddIn Type="Application">

<Name>My sample application</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.

See [Add-in Registration](#) for more information on the available XML tags for .addin files.

1.1.2.3 External Commands

Technically, an external command is an exposed .NET object that supports the Autodesk.Revit.UI.IExternalCommand interface. Furthermore, there must be a .addin manifest file in the appropriate directory with one entry for each such object in order for Revit to be able to "see" and to use the commands.

The IExternalCommand Interface

The declaration (VB.NET) of the interface is as follows:

Code Region: VB.NET IExternalCommand interface

```
Function Execute(ByVal commandData As Autodesk.Revit.UI.ExternalCommandData,
ByRef message As String,
ByVal elements As Autodesk.Revit.DB.ElementSet)
As Result
```

Parameters

- commandData : The object passed in this parameter contains information important to the command that is being executed. This data includes the Autodesk Revit Application object as well as the currently active view.
- message : The message string can be set to supply a specific message to the user when the command terminates. How this message is displayed is dependent upon the return value of the function. See the remarks section for more details.
- elements : Initially this is an empty set that can contain Autodesk Revit elements. When the command terminates, the elements within this set may be displayed, based on the return value. See the remarks section for more details.

Return value

result: The return value can be one of the following:

- Success : Is returned if the command succeeded as expected without any unhandled error conditions. The external command will appear as an undoable operation in the Autodesk Revit user interface.
- Cancelled : This value specifies that the user requested that the command be cancelled. Any changes that were made to Autodesk Revit objects during the external commands execution will be undone. A message may be posted, see the Remarks section.
- Failure : Failure signifies that the external command failed in some manner from which it cannot recover. Any changes made to Autodesk Revit objects during the execution of the external command will be undone. A message will be posted, see the Remarks section.

Remarks

The message and elements parameters are used if the command was cancelled or failed.

- Cancelled: If the external command was cancelled and the message parameter was set by the external command then the message is displayed when execution is returned back to Autodesk Revit. If the message parameter was not set then no message is displayed and the command will exit silently.
- Failed: If the external command failed then the contents of the message parameter will be displayed. If the element set contains Autodesk Revit elements then these elements will be highlighted when the error message is displayed thus giving the developer the ability to show the user the problem elements.

Using an Autodesk Revit API External Command

1. User opens/creates a project in Autodesk Revit
2. User selects the external command from the External Tools pulldown on the Add-ins tab.
3. The user had the option to select a number of Autodesk Revit elements before invoking the External Tools program. If they did, the program can decide to only perform its function on the selected members.
4. The API program takes focus from Autodesk Revit and performs the required task. Often a dialog box may be required to obtain user input before the application can complete its work.
5. Once the add-in tool has completed its function or has been dismissed by the user the program will update the Autodesk Revit model as required and return from the external command, giving focus back to Autodesk Revit.

External Command Object Lifetime

When no other command or edit modes are active within Autodesk Revit, the registered external command will become enabled. When picked, the command object will be created and the Execute method called. Once this method returns back to Autodesk Revit the command object will be destroyed. Due to this destruction, data cannot persist within the object between command executions. If you wish the data to persist you may use an external file or database to do so. If you wish the data to persist within the Autodesk Revit project you may use the shared parameters mechanism to store this data.

1.1.2.4 *External Applications*

Technically, an external application is an exposed .NET object that supports the Autodesk.Revit.IExternalApplication interface. Furthermore, there must be a .addin manifest file in the appropriate directory with one entry for each such object in order for Autodesk Revit to be able to load these applications when Autodesk Revit starts.

The IExternalApplication Interface

The declaration (C#) of the interface is as follows:

Code Region: IExternalApplication interface

```
Autodesk.Revit.UI.IExternalApplication.Result OnStartup(Autodesk.Revit.UIControlledApplication application)
```

```
Autodesk.Revit.UI.IExternalApplication.Result OnShutdown(Autodesk.Revit.UIControlledApplication application)
```

Parameters

- application: The object passed in this parameter contains information important to the commands OnStartup and OnShutdown that are being called. This object provides limited access methods of Autodesk Revit Application, such as VersionName, VersionNumber; and delegates for some events, such as OnDocumentOpened, OnDocumentSaved

Return Value

result: The return value can be one of the following:

- Success: Is returned if the external application succeeded as expected without any unhandled error conditions.
- Failure: Failure signifies that the external application failed in some manner from which it cannot recover.
- Cancelled: This value specifies that the external application be cancelled.

External Application Object Lifetime

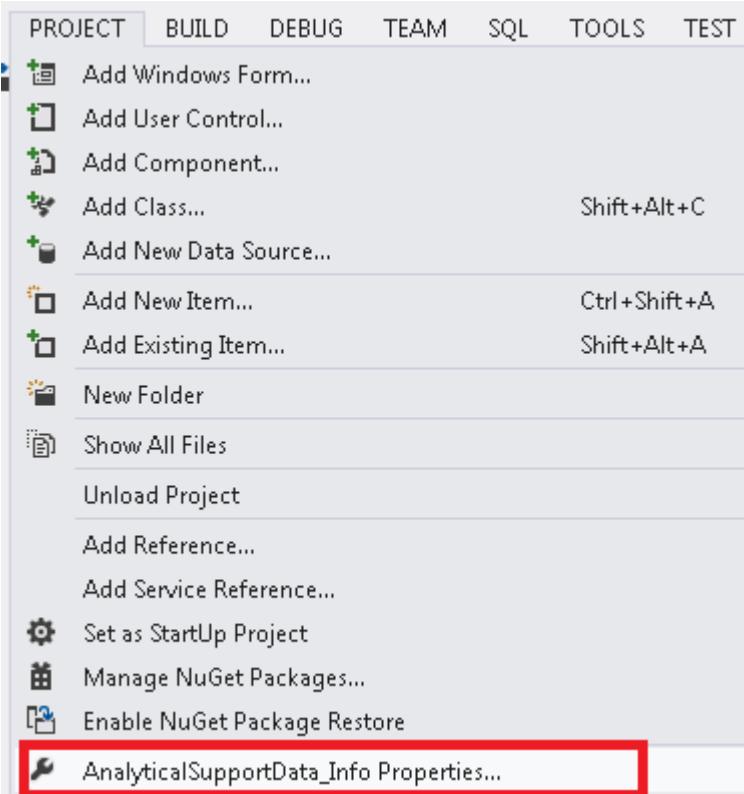
When Autodesk Revit starts, the external application object will be created and the OnStartup method called. Once this method returns back successfully to Autodesk Revit the external application object will be held during the entire Autodesk Revit session. The OnShutdown method will be called when Autodesk Revit shuts down.

1.1.2.5 Debugging Your Application in Microsoft Visual Studio

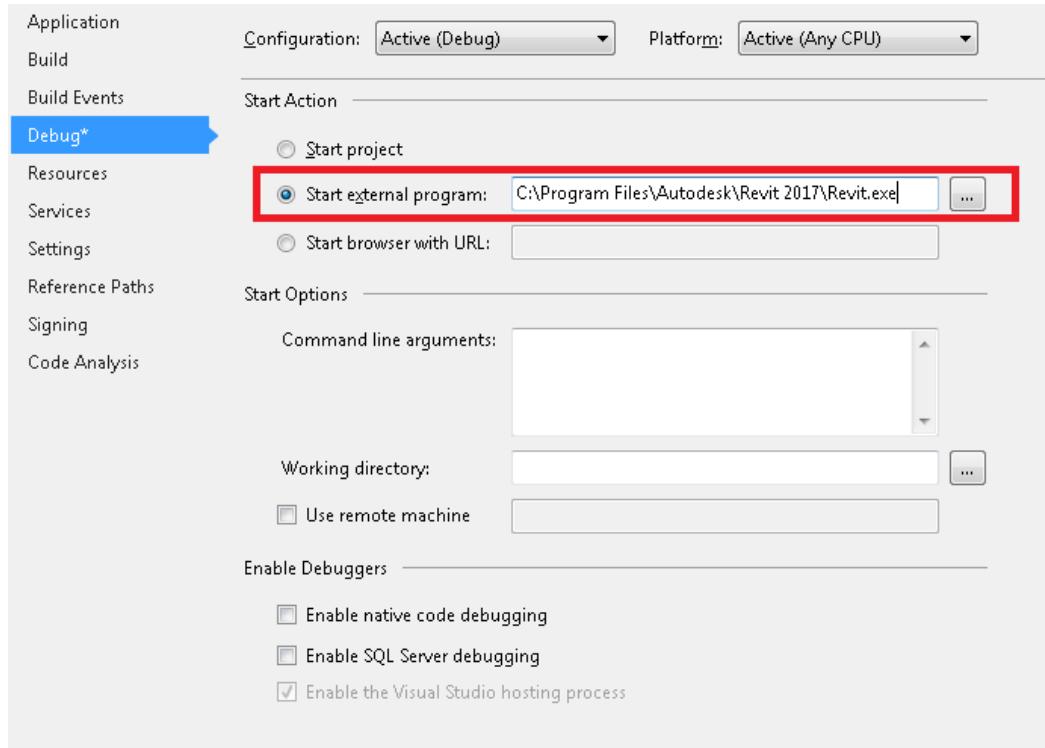
The following instructions apply to Visual Studio Professional. The relevant option is not available in the Visual Studio Community editions.

There are a few differences between debugging a standalone application (EXE) and an external application (DLL) that needs another program to launch it. To debug an application that is using the Autodesk Revit API it needs to be activated by Autodesk Revit. To do this in the developer environment for debugging you will need to:

1. Open up the Visual Studio project for the API application. (For example AnalyticalSupportData_Info.csproj from the Samples folder)
2. From the Project menu select AnalyticalSupportData_Info Properties



3. Select the Debug tab on the left
4. Select the "Start external program:" radio button
5. Press the browse button and find the Revit.exe file and press Open



6. Set some break points in your source code.
7. From Visual Studio, select "Start Debugging". Autodesk Revit will launch.
8. To hit a break point select the option for your program from the External Tools menu. Once the compiler reaches one of your break points it will stop to let you debug your program.

1.1.2.6 The Revit Unit System

The Revit Unit System uses the following base units:

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

Note: Because Revit stores lengths in feet and other quantities in metric, a derived unit involving length uses a non-standard unit using the Imperial and the Metric systems. For example, force is measured in mass-length per time squared and is stored in kg·ft/s².

1.1.2.7 Storing and accessing Custom Data for Applications

Often programs linked to Autodesk Revit require information that is not available in the Autodesk Revit model database. There are a number of ways for the user to enter such additional information. How and where the information is entered depends on its use:

- When the information is of a general type and the user will want to see and edit it inside Autodesk Revit, then it should be stored as a visible Project or Shared Parameter.
- If the information needs to be kept with the Autodesk Revit model as it evolves but does not need to be visible then it can be stored in the Autodesk Revit model as a non-visible Project or Shared Parameter or using Extensible Storage.
- If the information is specific to a single add-on program, and is too large to practically store within the Autodesk Revit model such as specifications for a multitude of building products subject to change, then the best solution may be to create a concurrent model database that stores the program specific information. In this case it may be useful to use the element UniqueId property for each element as a key for the database, because the element's UniqueId is stable within a model.

1.1.2.8 Migrating From .NET 4.8 to .NET 8

Revit 2025 and future releases will be built on .NET 8 and legacy Revit add-ins need to be recompiled for .NET 8.

The move from .NET 4.8 is to .NET 8 is a relatively large jump. .NET 8 comes from the .NET Core lineage, which has significant differences from .NET 4.8.

Upgrade Process

There are many Microsoft documents and tools to help application developers migrate from .NET 4.8 to .NET Core/5/6/7/8. Following is a list of some helpful documents:

- Overview of porting from .NET Framework to .NET document: <https://learn.microsoft.com/en-us/dotnet/core/porting/>
- .NET Upgrade Assistant can help with the project migration: <https://learn.microsoft.com/en-us/dotnet/core/porting/upgrade-assistant-overview>
- The .NET Portability Analyzer - .NET on C# projects to roughly evaluate how much work is required to make the migration as well as dependencies between the assemblies.
- Lists of breaking changes for .NET Core and .NET 5+: <https://learn.microsoft.com/en-us/dotnet/core/compatibility/breaking-changes>
- The .NET 8 SDK can be installed from here: <https://dotnet.microsoft.com/en-us/download/visual-studio-sdks>
- .NET SDK 8.0.100 is used to build the Revit January 2024 preview release.

- The Revit preview release will install .NET 8 Windows Desktop Runtime x64 8.0.0.33101.
- If you use Visual Studio to build .NET 8 code, you'll need [Visual Studio 17.8](#) or later.

Basic upgrade process for projects

For C# projects (CSPROJ)

1. Convert C# projects to the new SDK-style format: <https://learn.microsoft.com/en-us/dotnet/core/project-sdk/overview>
- a. The .NET Upgrade Assistant can help with the project migration: <https://learn.microsoft.com/en-us/dotnet/core/porting/upgrade-assistant-overview>
- b. Convert packages.json into PackageReferences in your CSPROJ. <https://learn.microsoft.com/en-us/nuget/consume-packages/migrate-packages-config-to-package-reference>
2. Update the target framework for your projects from to **net8.0-windows**
 - a. You can run the [.NET Portability Analyzer](#) on C# projects to evaluate how much work is required to make the migration.
 - b. The .NET Upgrade Assistant can help with the .NET version migration: <https://learn.microsoft.com/en-us/dotnet/core/porting/upgrade-assistant-overview>
 - c. If your application is a [WPF application](#), then the CSPROJ will need **net8.0-windows** and **true**.
 - d. If your application uses [Windows forms](#), then use **net8.0-windows** and **true**.
3. System references can be removed from the CSPROJ, as they are available by default.
4. Then address incompatible packages, library references and obsolete (unsupported) code.

For C++/CLI projects (VCXPROJ)

Refer to Microsoft's guide for migrating C++/CLI projects to .NET Core/5+: <https://learn.microsoft.com/en-us/dotnet/core/porting/cpp-cli>

5. Replace **true** with **NetCore**. This property is often in configuration-specific property groups, so you may need to replace it in multiple places.
6. Replace **property** with **net8.0-windows**.
7. Remove any .NET Framework references (like `<Reference Include="System">`) and add **FrameworkReference** when needed. .NET Core SDK assemblies are automatically referenced when using **NetCore** support.
8. Add FrameworkReferences:
 - a. To use Windows Forms APIs, add this reference to the vcxproj file:

```
<Reference Include="System.Windows.Forms" />
```
 - b. To use WPF APIs, add this reference to the vcxproj file:

```
<Reference Include="System.Windows" />
```
 - c. To use both Windows Forms and WPF APIs, add this reference to the vcxproj file:

```
<Reference Include="System.Windows.Forms" />
<Reference Include="System.Windows" />
```
9. Remove any `PlatformToolset` for cpp files. It will be set as NetCore by default. Any other values may cause issues.
10. Then address incompatible packages, library references and obsolete (unsupported) code.

Global.json

You may need to set **net8.0-windows** as the target in your global.json, if you have one. Refer the link for global.json overview: <https://learn.microsoft.com/en-us/dotnet/core/tools/global-json>

Component Versions

Your add-in may avoid instability by matching the version of these key components used by the Revit preview release:

- CefSharp
 - "cef.redist.x64" Version="119.4.3"
 - "cef.redist.x86" Version="119.4.3"
 - "CefSharp.Wpf.HwndHost" Version="119.4.30"
 - "CefSharp.Common.NetCore" Version="119.4.30"
 - "CefSharp.Wpf.NetCore" Version="119.1.20"
- Newtonsoft.Json
 - "Newtonsoft.Json" Version="13.0.1"

Supporting Multiple Revit Releases

A single code base can support older Revit releases on .NET 4.8 as well as Revit 2025 on .NET 8. See this discussion on the Autodesk forums for ideas on configuring your projects and code to support multi-targeting.

Common Issues

Here are some common issues you may encounter when upgrading to .NET 8:

Build Warning MSB3277

When building code that references RevitAPI or RevitUIAPI, you will see the build warning MSB3277. To fix this, add a reference to the Windows Desktop framework:

Build Error CA1416

If your application uses functions that are only available on Windows systems, you may see a CA1416 error. This can be fixed for the project by adding **[assembly: System.Runtime.Versioning.SupportedOSPlatformAttribute("windows")]** to **AssemblyInfo.cs**.

Obsolete Classes and Functions with .NET 8

Your .NET 4.8 application may see compile time or runtime errors if it uses classes or functions that are obsolete or deprecated in .NET Core/5/6/7/8.

Lists of breaking changes for .NET Core/5/6/7/8 are here: <https://learn.microsoft.com/en-us/dotnet/core/compatibility/breaking-changes>

- BinaryFormatter and SOAPFormatter are obsolete.
- Resource files that contain images or bitmaps will need to be updated as BinaryFormatter will not be available in .NET 8 to interpret those images/bitmaps.

- Windows Forms dialogs using ImageList may need to be updated as BinaryFormatter loads the images for the ImageList.
- System.Threading.Thread.Abort is obsolete.
- System.Reflection.AssemblyName.CodeBase and System.Reflection.Assembly.CodeBase are deprecated.
- Delegate.BeginInvoke is not supported.
- Debug.Assert failure or Debug.Fail silently exits the application by default.

Assembly Loading

Your .NET 4.8 application may need updates to help it find and load assemblies:

- .NET 8 uses a different assembly probing approach for DLL loading. When in doubt, try putting the DLL to be loaded in the build output root directory.
- .NET 8 assembly loading details are different than .NET 4.8.
- .NET 8 projects need runtimeconfig.json files for many DLLs. The **runtimeconfig.json** needs to be installed next to the matching DLL, and it configures the behavior of that DLL. These files can be created with **true**
- .NET 8 projects will create **deps.json** files for many DLLs. These **deps.json** files can be deleted if dependencies are placed in the same directory as the application. These files can be deleted with **false**

Assembly Properties

After updating your application to .NET 8, you may see build errors for your assembly properties. Many assembly properties are now auto-generated and can be removed from AssemblyInfo.cs.

Double Numbers To String

If you have unit tests or integration tests that compare doubles as strings, they may fail when you upgrade to .NET 8. This is because the number of decimal places printed by **ToString()** for doubles is different in .NET 4.8 and .NET 8. You can call **ToString("G15")** when converting doubles to strings to use the old .NET 4.8 formatting.

String.Compare

`String.Compare` behavior has changed, see .NET globalization and ICU and Use Globalization and ICU.

Windows Dialogs May Change Appearance

Your dialogs may change appearance with .NET 8.

- WinForms dialogs experience UI layout changes. The workaround is to set `Scale(new SizeF(1.0F, 1.0F))` in the dialog constructor.
- The dialog default font changed from "Microsoft Sans Serif 8 pt" to "Segoe UI". This can change dialog appearance and spacing.

Process.Start() May Fail

If your application is having trouble starting new processes, this may be because [System.Diagnostics.Process.Start\(url\)](#) has a behavior change. The [ProcessStartInfo.UseShellExecute](#) Property defaults to **true** in .NET 4.8 and **false** in .NET 8. Set **UseShellExecute=true** to workaround this change.

Encoding.Default Behaves Differently in .NET 8

If your application is having problems getting the text encoding used by Windows, it may be because **Encoding.Default** behaves differently in .NET 8. In .NET 4.8 **Encoding.Default** would get the system's active code page, but in .NET Core/5/6/7/8 [Encoding.Default](#) is always **UTF8**.

Items Order Differently in Sorted Lists

If you see different orderings of items in sorted lists after updating to .NET 8, this may be because [List.Sort\(\)](#) behaves differently in .NET 8 than .NET 4.8. The change fixes a .NET 4.8 bug which affected **Sort()** of items of equal value.

System.ServiceModel

`System.ServiceModel` has been ported to .NET Core through [CoreWCF](#), which is now available through Nuget packages. There are various changes, including [<System.ServiceModel>](#) not being supported in configuration files.

C# Language Updates

If you are building code from .NET 4.8 in .NET 8, you may see build errors or warnings about C# nullable types.

[C# has introduced nullable value types and nullable reference types](#). Prior to .NET 6, new projects used the default **disable**. Beginning with .NET 6, new projects include the **enable** element in the project file.

You can set **disable** if you want to revert to .NET 4.8 behavior.

Environment Variables

If you use managed .NET to run native C++ code, be aware that environmental variables, including the path variable for DLL loading, are not shared from managed .NET code with native C++ code.

1.1.3 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.

1.1.4 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 [Hello World for VB.NET](#).

The Hello World walkthrough covers the following topics:

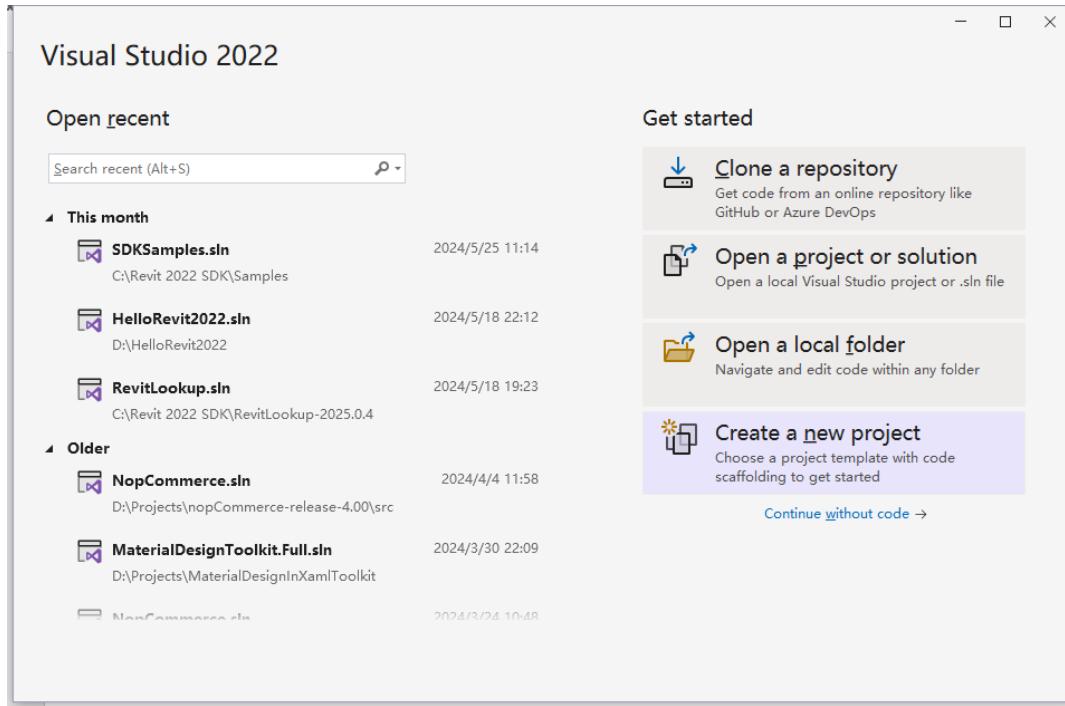
- Create a new project.
- Add references.
- Change the class name.
- Write the code
- Debug the add-in.

All operations and code in this section were created using Visual Studio 2019.

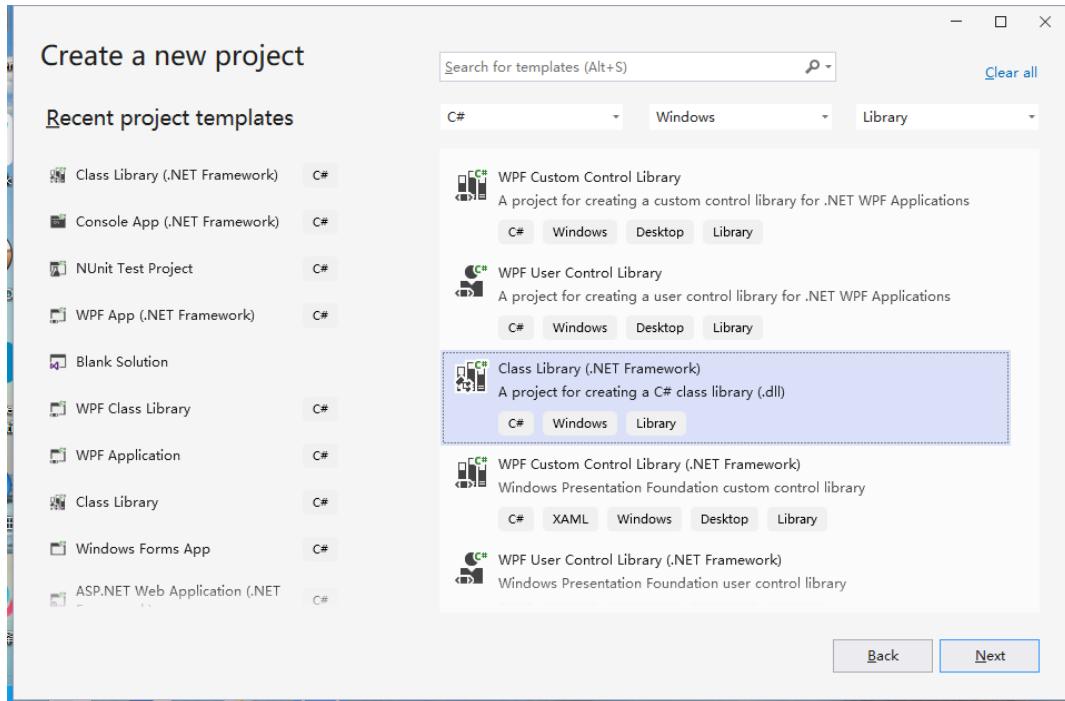
Create a New Project

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

1. Select "Create a new project".



2. Select the C#, Windows, and Library filters and the `Class Library` template, and click Next



3. In the Project Name field, type `HelloWorld` as the project name and click OK
4. Select the `.NET 8.0` framework and click Create

Add References

- If the Solution Explorer window is not open, select Solution Explorer from the View menu
- In the Solution Explorer, right-click the HelloWorld project (not the top-level Solution) and select `Add - Project Reference`
- Click Browse, go to the folder where Revit is installed (such as C:\Program Files\Autodesk\Revit 2025), and select `RevitAPI.dll`
- Click Browse again and select `RevitAPIUI.dll`
- Click OK to add both dll files which will then appear in the Solution Explorer under the HelloWorld project / Dependencies / Assemblies
- Set the `Copy Local` property of RevitAPI.dll and RevitAPIUI.dll to No. This saves disk space, and prevents the Visual Studio debugger from getting confused about which copy of the DLL to use. Select both files, right-click, select Properties, and change the Copy Local setting to No.

Add Code

Open the `Class1.cs` file, delete all of its contents, and add this code

```
using Autodesk.Revit.DB;

using Autodesk.Revit.UI;

namespace HelloWorld

{

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

    public class Class1 : IExternalCommand

    {

        public Result Execute(ExternalCommandData commandData, ref string message, ElementSet elementSet)

        {

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

            return 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. If you are typing this code into Visual Studio (instead of pasting it) After you add ":IExternalCommand" after Class1 in the example above, you can right-click on IExternalCommand, select Quick Actions and Refactorings, and then choose "Implement Interface" to get the code:

Figure 2: Using Intellisense to Implement Interface

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 [Add-In Integration](#).

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.

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, replacing the `Assembly` path with the correct path to the `HelloWorld.dll` on your computer:

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>C:\Users\<your user name>\source\repos\HelloWorld\HelloWorld\bin\Debug\net8.0\HelloWorld.dll</Assembly>

        <AddInId>239BD853-36E4-461f-9171-C5ACEDA4E721</AddInId>

        <FullClassName>HelloWorld.Class1</FullClassName>

        <Text>HelloWorld</Text>

        <VendorId>NAME</VendorId>
```

```
</AddIn>
</RevitAddIns>
```

3. Save the file as HelloWorld.addin and put it in the following location:
 - C:\ProgramData\Autodesk\Revit\Addins\{Release Year}\
 - If your application assembly dll is on a network share instead of your local hard drive, you must modify Revit.exe.config to allow .NET assemblies outside your local machine to be loaded. In the "runtime" node in Revit.exe.config, add the element `<loadFromRemoteSources enabled="true"/>` as shown below.

```
<runtime>
  <generatePublisherEvidence enabled="false" />
  <loadFromRemoteSources enabled="true"/>
</runtime>
```

Refer to [Add-In Integration](#) for more details using manifest files.

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. Start Revit
2. In Visual Studio, choose Debug - Attach To Process, and select Revit.exe
3. From the Debug menu, select Toggle Breakpoint (or press F9) to set a breakpoint on the following line.

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

Test debugging:

- On the Add-Ins tab, HelloWorld appears in the External Tools menu-button. **HelloWorld External Tools command** **Figure 4:**
- Click HelloWorld to execute the program, activating the breakpoint.
- Press F5 to continue executing the program. The following system message appears.
- *Figure 5: TaskDialog message**

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 [Walkthrough: Hello World](#).

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 manifest file, a message similar to the following appears.

Figure 6: External Tools Error Message

In this case, ensure that the .addin file has the correct path to the assembly.

Error messages will also be displayed if the class name specified in FullClassName 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
- The manifest file is not found
- There is a formatting error in the .addin manifest file

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.

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

Figure 7: 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 is an example:

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

```
public Result Execute(ExternalCommandData commandData, ref string message, ElementSet elements)
```

```
{
```

```
ExternalCommandData cdata = commandData;

Autodesk.Revit.ApplicationServices.Application app = cdata.Application;

try

{

    // Your code here

}

catch (Exception ex)

{

    message = ex.Message;

    return Autodesk.Revit.UI.Result.Failed;

}

return Autodesk.Revit.UI.Result.Succeeded;

}
```

1.1.5 Walkthrough: Add Hello World Ribbon Panel

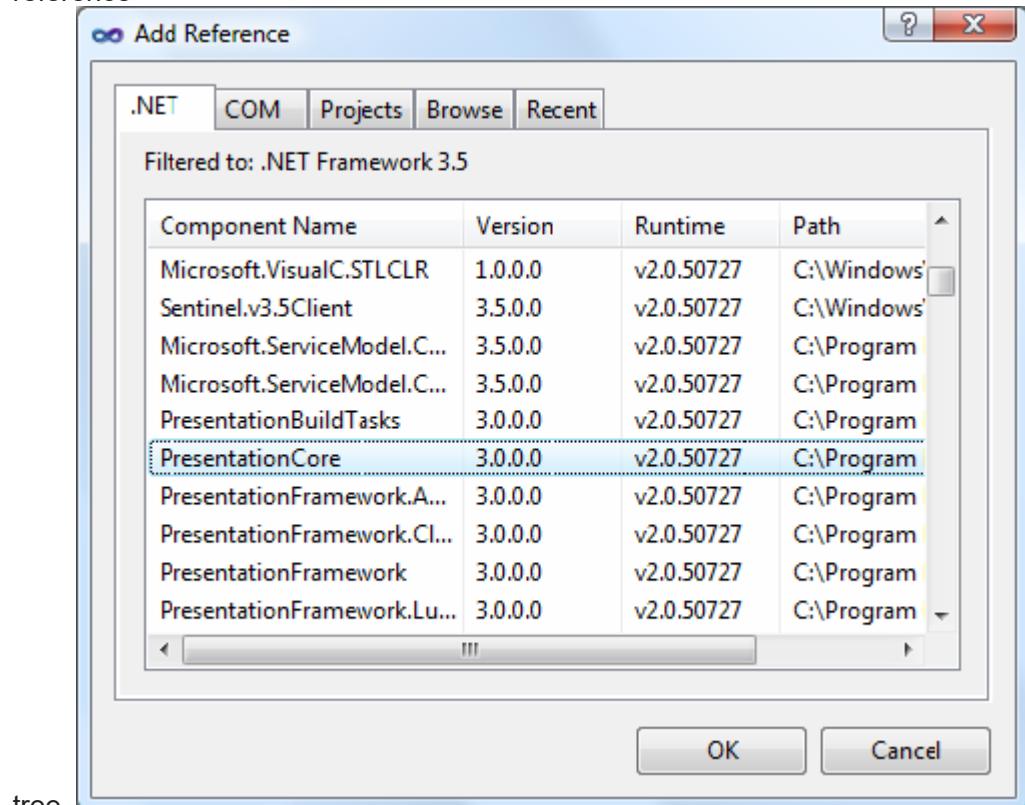
In the Walkthrough: Hello World section you learn how to create an add-in application and invoke it in Revit. You also learn to create a .addin manifest file to register the add-in application as an external tool. Another way to invoke the add-in application in Revit is through a custom ribbon panel.

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 8: Add Reference**
- Add the WindowsBase reference as well as System.Xaml following similar steps as above.

Change the Class Name

To change the class name, complete the following steps:

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

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 [Add-In Integration](#).

Add the following code to the file:

Code Region 2-5: Adding a ribbon panel

```
using System;
using System.Reflection;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
using System.Windows.Media.Imaging;

namespace Walkthrough
{
    /// <remarks>
    /// This application's main class. The class must be Public.
    /// </remarks>
    public class CsAddPanel : IExternalApplication
    {
        // Both OnStartup and OnShutdown must be implemented as public method
        public Result OnStartup(UIControlledApplication application)
        {
            // Add a new ribbon panel
            RibbonPanel ribbonPanel = application.CreateRibbonPanel("NewRibbonPanel");
        }

        // Create a push button to trigger a command add it to the ribbon panel.
        string thisAssemblyPath = Assembly.GetExecutingAssembly().Location;
        PushButtonData buttonData = new PushButtonData("cmdHelloWorld",
            "Hello World", thisAssemblyPath, "Walkthrough.HelloWorld");

        PushButton pushButton = ribbonPanel.AddItem(buttonData) as PushButton;
```

```
// Optionally, other properties may be assigned to the button

// a) tool-tip

pushButton.ToolTip = "Say hello to the entire world.';




// b) large bitmap

Uri uriImage = new Uri(@"C:\Sample\HelloWorld\bin\Debug\39-Globe_32x32.p
ng");

BitmapImage largeImage = new BitmapImage(uriImage);

pushButton.LargeImage = largeImage;




return Result.Succeeded;

}

public Result OnShutdown(UIControlledApplication application)

{

    // nothing to clean up in this simple case

    return Result.Succeeded;

}

/// <remarks>

/// The "HelloWorld" external command. The class must be Public.

/// </remarks>

[Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMo
de.Manual)]



public class HelloWorld : IExternalCommand

{



    // The main Execute method (inherited from IExternalCommand) must be public
```

```

public Autodesk.Revit.UI.Result Execute(ExternalCommandData revit,
    ref string message, ElementSet elements)
{
    TaskDialog.Show("Revit", "Hello World");
    return Autodesk.Revit.UI.Result.Succeeded;
}
}
}
}

```

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.

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 the external application

```

<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
    <AddIn Type="Application">
        <Assembly>C:\Sample\AddPanel\AddPanel\bin\Debug\AddPanel.dll</Assembly>
        <AddInId>604b1052-f742-4951-8576-c261d1993108</AddInId>
        <FullClassName>Walkthrough.CsAddPanel</FullClassName>
        <VendorId>NAME</VendorId>
        <VendorDescription>Your Company Information</VendorDescription>
    </AddIn>
</RevitAddIns>

```

3. Save the file as HelloWorldRibbon.addin and put it in:
C:\ProgramData\Autodesk\Revit\Addins\2025\

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

Refer to [Add-In Integration](#) for more information about .addin manifest files.

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.

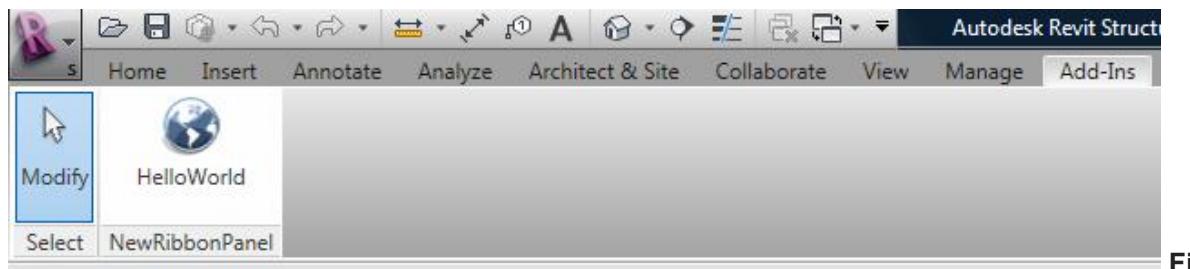


Figure 9: Add a new ribbon panel to Revit

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



Figure 10: Hello World dialog box

1.1.6 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.GetElementIds()` method to retrieve the selected elements.
3. Create a .addin file as explained in previous walkthroughs.

GetElementIds() returns a collection of ElementIds of the selected elements. It can be iterated with a foreach loop. Use the Document.GetElement() method to get the Element object for each ElementId in the selection.

The following code is an example of how to retrieve the ids of the selected element.

Code Region 2-7: Retrieving selected elements

```
[Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.ReadOnly)]  
  
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.  
            ICollection<ElementId> selectedIds = uidoc.Selection.GetElementIds();  
  
            if (selectedIds.Count == 0)  
            {  
                TaskDialog.Show("Revit", "You haven't selected any elements.");  
            }  
        }  
    }  
}
```

```
        else

    {

        string info = "Ids of selected elements in the document are:
";

        foreach (ElementId id in selectedIds)

        {

            info += Environment.NewLine + id.Value;

        }

        TaskDialog.Show("Revit", info);

    }

}

catch (Exception e)

{

    message = e.Message;

    return Autodesk.Revit.UI.ResultFailed;

}

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 [Parameters](#).

1.1.7 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 [Element Retrieval](#).

This example retrieves all the doors in the document and returns the list of door elements.

Code Region 2-8: Retrieve filtered elements

```
public ICollection<Element> CreateLogicAndFilter(Autodesk.Revit.DB.Document document)
{
    // 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).ToList();

    return doors;
}
```

1.2 Add-In Integration

Developers add functionality by creating and implementing External Commands and External Applications. Revit identifies the new commands and applications 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.

1.2.1 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. It also contains IExternalDBApplication and related interfaces.

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.

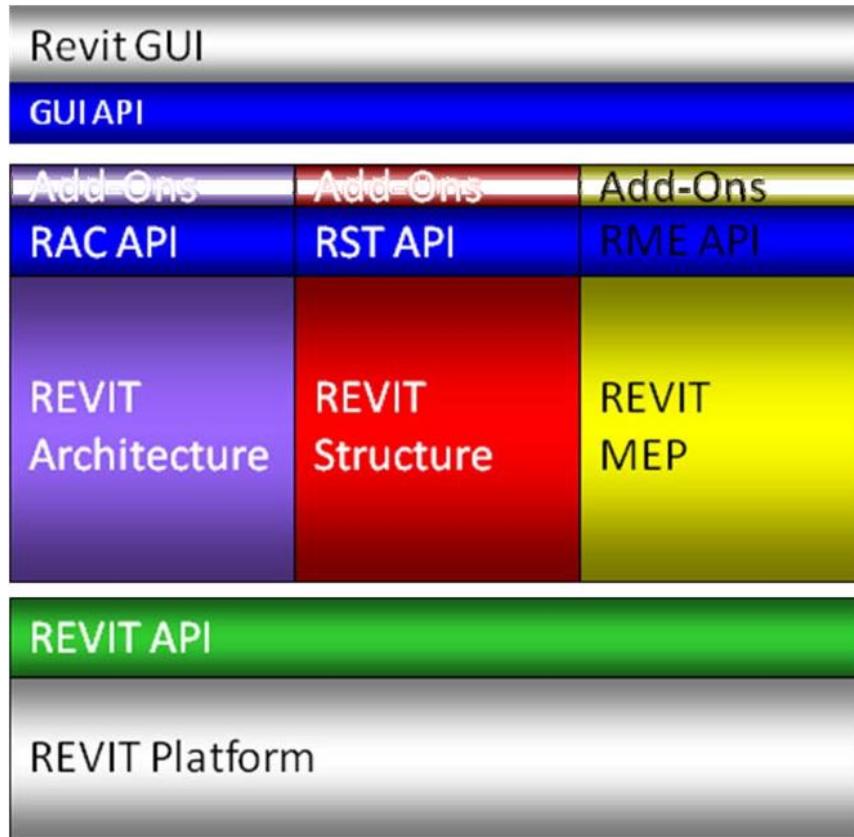


Figure 11: Revit, RevitAPI and Add-ins

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, IExternalApplication, or IExternalDBApplication. In this way, the add-in is run automatically on certain events or, in the case of IExternalCommand and IExternalApplication, manually from the External Tools menu-button.

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

1.2.2 External Commands

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

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, Analyze tab or a new custom ribbon tab. See the [Walkthrough: Hello World](#) and [Walkthrough: Add Hello World Ribbon Panel](#) for examples of these two approaches.

External tools, ribbon tabs 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.

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**)

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

Table 1: ExternalCommandData public properties

Property	Description
Application (Autodesk.Revit.UI.UIApplication)	Retrieves an object that represents the current UIApplication for external command.
JournalData (IDictionary<String, String>>)	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.

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(
        ExternalCommandData commandData, ref string message,
        ElementSet elements)
    {
        message = "Could not locate walls for analysis.";
        return Autodesk.Revit.UI.Result.Failed;
    }
}
```

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

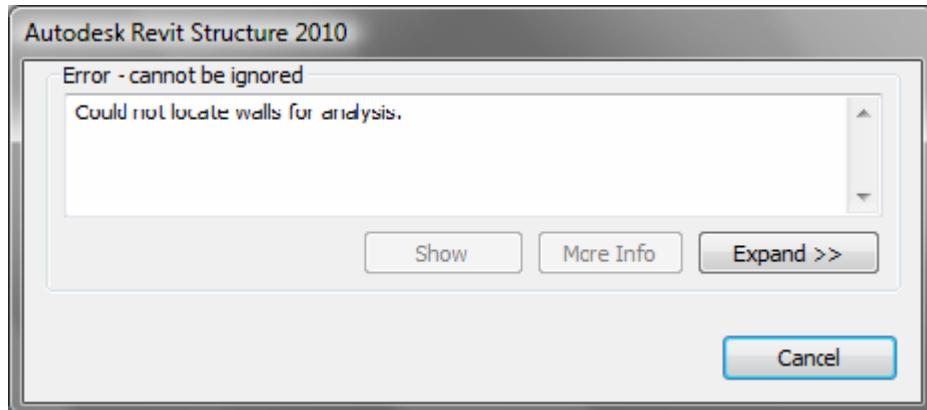


Figure 12: Error message dialog box

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;
}

}

```

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.

Table 2: IExternalCommand.Result

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.ResultFailed	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".

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.GetElementIds());  
  
  
        TaskDialog taskDialog = new TaskDialog("Revit");  
  
        taskDialog.MainContent =  
  
            ("Click Yes to return Succeeded. Selected members will be deleted.  
d.\n" +  
  
             "Click No to return Failed. Selected members will not be deleted.  
d.\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)
```

```
{  
    ICollection<ElementId> selectedElementIds = uidoc.Selection.GetElementIds();  
  
    foreach (ElementId id in selectedElementIds)  
    {  
        elements.Insert( doc.GetElement(id) );  
    }  
  
    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;  
}  
  
}  
}
```

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(Autodesk.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.Value == (int)BuiltInCategory.OST_Walls)
                return true;
        }
        return false;
    }
}
```

1.2.3 External Application

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

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 and the addition of ribbon tabs. 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 and 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 Rev
it 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

    string dialogId = args.DialogId;

    // return if the dialog has no DialogId (such as with a Task Dial
og)
```

```
    if (dialogId == "")  
        return;  
  
    // Show the prompt message and allow the user to close the dialog  
    // directly.  
  
    TaskDialog taskDialog = new TaskDialog("Revit");  
    taskDialog.MainContent = "A Revit dialog is about to be opened.\n"  
    " +  
    "The DialogId of this dialog is " + dialogId + "\n" +  
    "Press 'Cancel' to immediately dismiss the dialog";  
    taskDialog.CommonButtons = TaskDialogCommonButtons.Ok |  
        TaskDialogCommonButtons.Cancel;  
    TaskDialogResult result = taskDialog.Show();  
    if (TaskDialogResult.Cancel == result)  
    {  
        // dismiss the Revit dialog  
        args.OverrideResult(1);  
    }  
}
```

1.2.4 Add-in Registration

External commands and external applications need to be registered in order to appear inside Revit. They can be registered by adding them to a .addin manifest file.

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.

Manifest Files

Revit API applications are registered with Revit 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":

- C:\ProgramData\Autodesk\Revit\Addins\Revit 2018\

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

- C:\Users<user>\AppData\Roaming\Autodesk\Revit\Addins\Revit 2018\

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>

        <VendorId>ADSK</VendorId>

        <VendorDescription>Autodesk, www.autodesk.com</VendorDescription>

        <VisibilityMode>NotVisibleInFamily</VisibilityMode>

        <Discipline>Structure</Discipline>

        <Discipline>Architecture</Discipline>

        <AvailabilityClassName>Revit.Samples.SampleAccessibilityCheck</AvailabilityClassName>

        <LongDescription>
```

```
    This is the long description for my command.  
  
    This is another descriptive paragraph, with notes about how to use the command properly.  
  
    </LongDescription>  
  
    <TooltipImage>c:\MyProgram\Autodesk.png</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>  
  
    <VendorId>ADSK</VendorId>  
  
    <VendorDescription>Autodesk, www.autodesk.com</VendorDescription>  
  
  </AddIn>  
  
</RevitAddIns>
```

A basic file adding one DB-level External Application looks like this:

Code Region: Manifest .addin ExternalDBApplication

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

```

<AddIn Type="DBApplication">
    <Assembly>c:\MyDBLevelApplication\MyDBLevelApplication.dll</Assembly>
    <AddInId>DA3D570A-1AB3-4a4b-B09F-8C15DFEC6BF0</AddInId>
    <FullClassName>MyCompany.MyDBLevelAddIn</FullClassName>
    <Name>My DB-Level AddIn</Name>
    <VendorId>ADSK</VendorId>
    <VendorDescription>Autodesk, www.autodesk.com</VendorDescription>
</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".
VendorId	A unique vendor identifier that may be used by some operations in Revit (such as identification of extensible storage). This must be unique, and thus we recommend to use reversed version of your domain name, for example, com.autodesk or uk.co.autodesk.
VendorDescription	Description containing vendor's legal name and/or other pertinent information. Optional.
Description	<p>Short description of the command, will be used as the button tooltip. Optional; use this tag for ExternalCommands only.</p> <p>The default is a tooltip with just the command text.</p>
VisibilityMode	<p>The modes in which the external command will be visible. Multiple values may be set for this option. Optional; use this tag for ExternalCommands only.</p> <p>The default is to display the command in all modes, 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. See table below for more information.</p>
Discipline	<p>The disciplines in which the external command will be visible. Multiple values may be set for this option. Optional; use this tag for ExternalCommands only.</p> <p>The default is to display the command in all disciplines. If any specific disciplines are listed, the command will only be visible in those disciplines. See table below for more information.</p>
AvailabilityClassName	<p>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.</p> <p>The default is a command that is available whenever it is visible.</p>

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 TooltiplImage are not supplied, the button will not have an extended tooltip.
TooltiplImage	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 TooltiplImage are not supplied, the button will not have an extended tooltip.
LanguageType	Localization setting for Text, Description, LargeImage, LongDescription, and TooltiplImage 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.
AllowLoadIntoExistingSession	The flag for loading permission. Set to false to prevent Revit from automatically loading addins in a newly added .addin manifest file without restarting. Optional. By default. Revit will automatically load addins from newly added .addin manifest files without restarting Revit.

Table 3: VisibilityMode Members

Member Name	Description

AlwaysVisible	The command is available in all possible modes supported by the Revit API.
NotVisibleInProject	The command is invisible when there is a project document active.
NotVisibleInFamily	The command is invisible when there is a family document active.
NotVisibleWhenNoActiveDocument	The command is invisible when there is no active document.

Table 4: Discipline Members

Member Name	Description
Any	The command is available in all possible disciplines supported by the Revit API.
Architecture	The command is visible in Autodesk Revit Architecture.
Structure	The command is visible in Autodesk Revit Structure.
StructuralAnalysis	The command is visible when the Structural Analysis discipline editing tools are available.
MassingAndSite	The command is visible when the Massing and Site discipline editing tools are available.
EnergyAnalysis	The command is visible when Energy Analysis discipline editing tools are available.
Mechanical	The command is visible when the Mechanical discipline editing tools are available, e.g. in Autodesk Revit MEP.
Electrical	The command is visible when the Electrical discipline editing tools are available, e.g. in Autodesk Revit MEP.
Piping	The command is visible when the Piping discipline editing tools are available, e.g. in Autodesk Revit MEP.

MechanicalAnalysis	The command is visible when the Mechanical Analysis discipline editing tools are available.
PipingAnalysis	The command is visible when the Piping Analysis discipline editing tools are available.
ElectricalAnalysis	The command is visible when the Electrical Analysis discipline editing tools are available.

.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

```
public void ManifestFile()
{
    //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", "ADSK");
    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.Discipline = Discipline.Mechanical | Discipline.Electrical |
        Discipline.Piping | Discipline.Structure;
```

```

        command1.VisibilityMode = VisibilityMode.NotVisibleInFamily;

        //create an external application

        RevitAddInApplication application1 = new RevitAddInApplication("appNa
me",

        "full path\\assemblyName.dll", Guid.NewGuid(), "namespace.className",
"ADSK");

        //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.GetAllInstalledRevit
Products()[0];

        Manifest.SaveAs(revitProduct1.AllUsersAddInFolder + "\\RevitAddInUtil
itySample.addin");

    }
}

```

Code Region 3-12: Reading an existing manifest file

```

public void ReadManifest()

{
    RevitProduct revitProduct1 = RevitProductUtility.GetAllInstalledRevit
Products()[0];

    RevitAddInManifest revitAddInManifest =
    Autodesk.RevitAddIns.AddInManifestUtility.GetRevitAddInManifest(
        revitProduct1.AllUsersAddInFolder + "\\RevitAddInUtilitySample.addi
n");
}

```

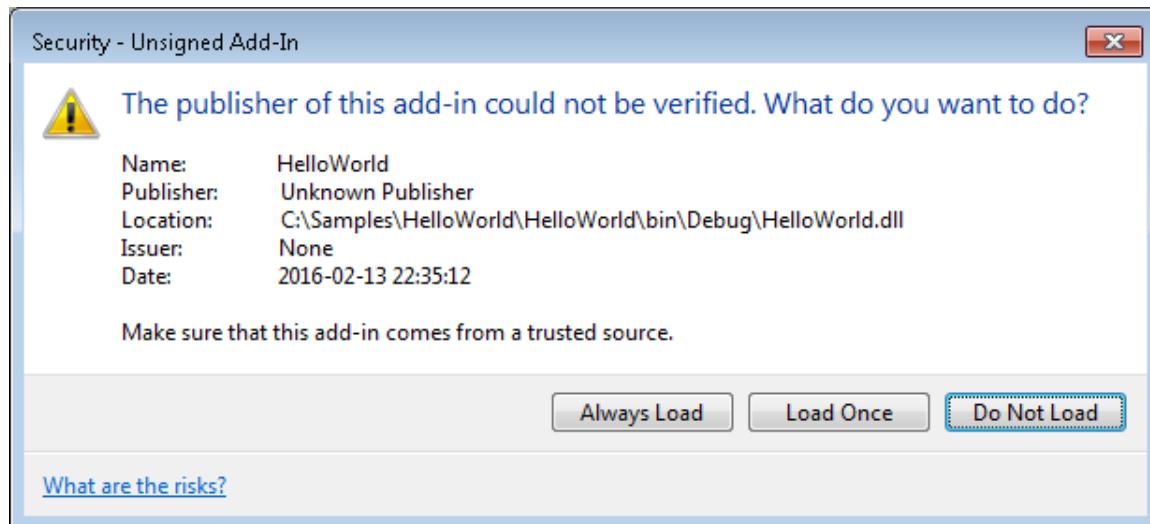
Access to add-in data paths

Autodesk.Revit.ApplicationServices.Application.CurrentUsersAddinsDataFolderPath provides access to add-in data folder for the current Revit version and current user (such as %appdata%\Autodesk\Revit\Autodesk Revit 2019\AddinsData)

1.2.5 Digitally Signing Your Revit Add-in

Revit checks security credentials of add-ins.

If an add-in is not digitally signed with a certificate issued by a trusted certificate authority, Revit pops up a dialog when opening asking the user to confirm if he/she wants to load the application. The figure below shows an example of the security warning dialog when an unsigned add-in is detected. The user is presented with choices of: 1) allowing the same add-in to always load from now on, 2) load only this time and ask again next time, and 3) do not allow to load the add-in.



If you are professional developer and your application is already digitally signed by a trusted certificate authority, your add-in is already compatible with the digital signature checking in Revit. The following sections are intended for developers who author add-ins, but who are not familiar with digital signing in Revit.

1.2.5.1 Digitally Signing Your App

If you are publisher of a Revit add-in, you will have to sign your add-in with your own certificate.

To sign your add-in with your own certificate, you first need to purchase a digital signature from a digital certificate vendor. Once you obtain a certificate (cer) or Personal Information Exchange (pfx) file, you can sign your DLL(s) using **signtool**.

Alternatively, you can also use an online Authenticode signing service, such as Symantec's Secure App Service - <https://www.symantec.com/code-signing/secure-app-service/>.

Digital Certificate Vendors

The following is a non-exhaustive list of vendors that provide digital certificates:

- Symantec - www.symantec.com
- DigiCert - www.digicert.com
- VERISIGN - www.verisign.com
- Thawte - www.thawte.com

Signing using signtool

You can use signtool.exe tool to sign your .NET dll. The tool is automatically installed with Visual Studio. To run the tool, use the Developer Command Prompt. The following is the format of the command line parameters:

Command Region: Using signtool

```
signtool.exe sign /fd SHA256 /f <.pfx-file-name> /p <password> <file-to-sign>.dl
l
```

Where /fd is a flag for the file digest algorithm to use. Here we use SHA256. (SHA stands for Secure Hash Algorithm. The signtool default is SHA1. We recommend SHA256, which is a newer, more secure version.) <.pfx-file-name> is the name of .pfx (Personal Information Exchange) file you obtain from the vendor. <password> is the password that you specify when obtaining the pfx file. <file-to-sign>.dll is the name of the DLL that you want to sign.

For example, if you run the command in an arbitrary folder, the above command may look like this:

Command Region: signtool example

```
"C:\Program Files (x86)\Windows Kits\8.1\bin\x64\signtool" sign /fd SHA256 /f "C:
/Dev/MyCert.pfx" /p "password123" "C:/Dev/HelloRevit.dll"
```

Note: The exact location of signtool may differ in your environment.

Once the DLL is signed with an authorized certification, Revit will no longer pop up a security warning dialog upon loading your add-in.

You can also include the command in the Post-Built Event section of Visual Studio for your application project properties.

Command Region: Post-Build Event signing

```
"C:\Program Files\Microsoft SDKs\Windows\v7.1\Bin\signtool.exe" /fd SHA256 sign /
f

"C:\Autodesk\MyCert.pfx" /p MyPassword "$(TargetDir)$(TargetFileName)"
```

It is also worth adding a time stamp while signing (/td and /tr switches in signtool.exe); otherwise the app becomes untrusted when the certificate expires. Adding the time stamp ensures the app is trusted forever as long as it was signed prior to expiration (unless the certificate gets revoked):

Command Region: Adding a time stamp

```
signtool.exe timestamp /td sha256 /tr <URL-of-time-stamp-server> <file-to-sign>.d  
ll
```

For example, the following uses the verisign timestamp server:

Command Region: Time stamp example

```
signtool.exe timestamp /td sha256 /tr "http://sha256timestamp.ws.symantec.com/sha  
256/" HelloRevit.dll
```

Note: The /td sha256 and /tr switches used in the example above are used to sign with sha256 timestamp. Microsoft treats SHA1 timestamps as unsigned. Please refer to [this article](#) for more details.

1.2.5.2 Making Your Own Certificate for Testing and Internal Use

You can make your own digital certificate for testing or using within your company.

To create your own digital certificate

1. Create a digital certificate using the [MakeCert.exe](#) tool.
2. Create a Personal Information Exchange (pfx) file using the Pvk2Pfx.exe tool.
3. [Digitally Signing Your App](#).
4. Import a Digital Certificate to Windows Certificate Store. ([CertMgr.msc](#) or [CertUtil.exe](#))

Create a Digital Certificate

You can use [MakeCert.exe](#) tool to make your own digital certificate for testing and internal use. The following is the command format:

Command Region: Make a certificate command format

```
MakeCert.exe -r -sv <name-of-private-key-file>.pvk -n "CN=<developer-name>" <  
name-of-certificate-file>.cer -b <start-data> -e <end-date>
```

Where <name-of-private-key-file> is the name of the file where the private key is stored, <developer-name> is the name of the developer, <name-of-certificate-file> is the name of the certificate file, <start-date> is the date when the certificate became valid (format is mm/dd/yyyy), and <end-date> is the date when the validity of the certificate ends.

For example:

Command Region: MakeCert.exe example

```
"C:\Program Files\Microsoft SDKs\Windows\v7.1\Bin\MakeCert.exe" -r -sv MyCert.pvk -n "CN=DevABC" MyCert.cer -b 01/01/2016 -e 12/31/2016
```

Or:

Command Region: MakeCert.exe example

```
"C:\Program Files (x86)\Windows Kits\8.1\bin\x64\makecert.exe" -r -sv MyCert.pvk -n "CN=DevABC" MyCert.cer -b 01/01/2016 -e 12/31/2016
```

This command will bring up "Create Private Key Password" dialog. Enter the private key password in the dialog. If it asks for password, enter again. When everything is done, you will see a message "Succeeded" in the command window and .cer and .pvk files are created.

Convert to PFX

The next step is to convert a digital certificate to a Personal Information Exchange (pfx) file using the pvk2pfx.exe tool. In this step, you need the .pvk file, .cer file, and password you created in the above step. The command format looks like this:

Command Region: Convert to PFX command format

```
pvk2pfx.exe" -pvk <name-of-private-key-file>.pvk -pi <password-for-pvk> -spc <name-of-certification-file-name>.cer  
-pfx <name-of-pfx-file> -po <password-for-pfx>
```

Where `<name-of-private-key-file>` is the name of pvk file you created, `<password-for-pvk>` is the password you assigned to the pvk file. `<name-of-certification-file-name>` is the name of the certification file or .cer file. `<name-of-pfx-file>` is the name of the .pfx. `<password-for-pfx>` is a password to be assigned to the .pfx file.

For example:

Command Region: Convert to PFX example

```
"C:\Program Files (x86)\Windows Kits\8.1\bin\x64\pvk2pfx.exe" -pvk MyCert.pvk  
-pi password123 -spc MyCert.cer -pfx MyCert.pfx -po password234
```

When the operation succeeds, the command ends without error message and a .pfx file will be created.

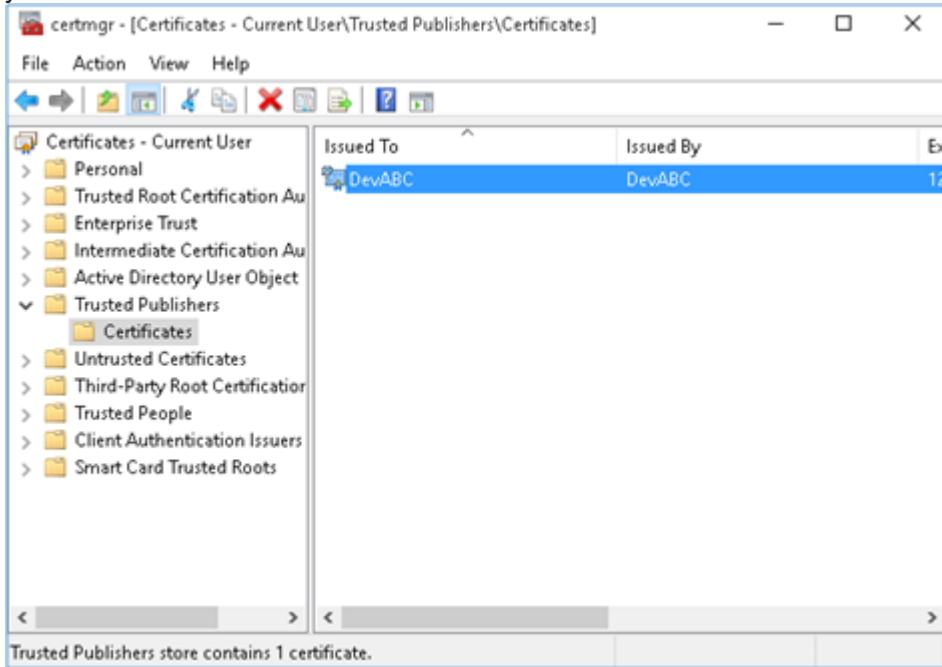
Once you have a pfx file, you can [Digitally Signing Your App](#).

Import the Digital Certificate to Windows Certificate Store

One more step you need when you are making your own digital certificate is to import it to your computer. You can do this in Certificate Manager ([CertMgr.msc](#)) or CertUtil.exe tool. Here we use the UI tool. Please refer [here for alternatives](#).

1. From Start >> Run >> CertMgr.msc. (Or on Windows 8.1/10, right click on Start >> Run >> CertMgr.msc) CertMgr opens.
2. On CertMgr dialog, right click on **Trusted Publishers** >> All Tasks >> Import ...
3. Follow the instructions on Certificate Import Wizard. Click Next.
4. On a dialog which asks "Files to Import", choose the pfx file you want to import.
5. On the "Password" dialog, enter the password. Keep "Include all extended properties" checked.
6. Choose "Place all certificates in the following store" then Click Next.
7. Confirm and Finish.

8. If you see "Import a new Private signature key" dialog, click OK. (This part may differ depending on your



- environment.)
9. Repeat the same step with **Trusted Root Certification Authorities**. This step is to validate digitally signed binary files.

1.2.5.3 *Digital Signature References*

The following references provide more information on digitally signing apps.

Revit Help

- [About Digital Signatures](#)
- [Security: Signed File or Add-In](#)
- [Security: Unsigned File or Add-In](#)
- [Security: Invalid Signature](#)

Microsoft and Other Sites

- [How to sign an app package using SignTool](#)
- [The truth about SHA1, SHA-256 and Code Signing Certificates](#)

AutoCAD References

AutoCAD Blog

- <http://adndevblog.typepad.com/autocad/2015/01/digitally-signing-plug-in-files.html>

- http://through-the-interface.typepad.com/through_the_interface/2015/05/signing-your-application-modules-for-autocad-2016-part-1.html

AutoCAD Help

- [To Digitally Sign a Binary \(ObjectARX or Managed .NET\) File](#)
- [To Digitally Sign a Binary \(ObjectARX or Managed .NET\) File with a Post-Build Event in Microsoft Visual Studio](#)
- [To Make a Digital Certificate](#)
- [To Create A Personal Information Exchange \(PFX\) File](#)
- [To Import a Digital Certificate](#)

1.2.6 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.

1.2.7 Attributes

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

TransactionAttribute

The custom attribute Autodesk.Revit.Attributes.TransactionMode must be applied to your implementation class of the IExternalCommand interface to control transaction behavior for the 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.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 either mode, 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 manual transaction mode:

Code Region 3-18: TransactionAttribute

```
[Transaction(TransactionMode.Manual)]

public class CommandTransaction : IExternalCommand
{
    public Result Execute(
        ExternalCommandData commandData,
        ref string message, Autodesk.Revit.DB.ElementSet elements)
    {
        // Command implementation, which modifies the active document
        // directly
        // after starting a new transaction
        return Result.Succeeded;
    }
}
```

See [Transactions](#).

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.JournalData 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 IDictionary<String, String> supplied in the command data. This will hide all Revit journal entries between the external command invocation and the IDictionary<String, String> 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 CommandJournal : IExternalCommand  
  
{  
  
    public Result Execute(  
  
        ExternalCommandData commandData,  
  
        ref string message, Autodesk.Revit.DB.ElementSet elements)  
  
    {  
  
        return Autodesk.Revit.UI.Result.Succeeded;  
  
    }  
  
}
```

1.2.8 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.

1.2.9 Ribbon Panels and Controls

Revit provides API solutions to integrate custom ribbon panels and controls.

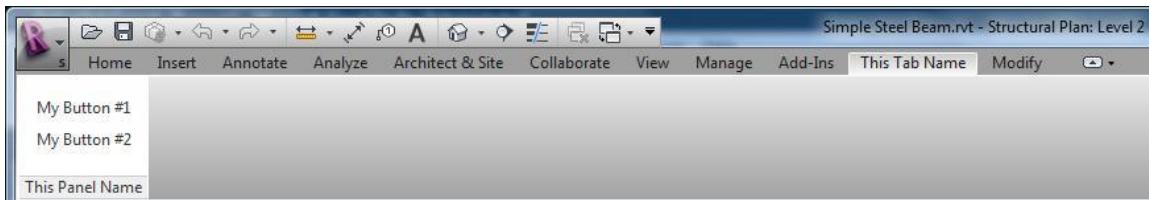
These APIs are used with `IExternalApplication`. Custom ribbon panels can be added to the Add-Ins tab, the Analyze tab or to a new custom ribbon tab.

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 [Ribbon Guidelines](#) in the [API User Interface Guidelines](#) section for information on developing a user interface that is compliant with the standards used by Autodesk.

Create a New Ribbon Tab

Although ribbon panels can be added to the Add-Ins or Analyze tab, they can also be added to a new custom ribbon tab. This option should only be used if necessary. To ensure that the standard Revit ribbon tabs remain visible, a limit of 20 custom ribbon tabs is imposed. The following image shows a new ribbon tab with one ribbon panel and a few simple controls.



Below is the code that generated the above ribbon tab.

Code Region: New Ribbon tab

```
public Result OnStartup(UIControlledApplication application)
{
    // Create a custom ribbon tab

    String tabName = "This Tab Name";
    application.CreateRibbonTab(tabName);

    // Create two push buttons

    PushButtonData button1 = new PushButtonData("Button1", "My Button #1",
    
```

```

        @"C:\ExternalCommands.dll", "Revit.Test.Command1");

    PushButtonData button2 = new PushButtonData("Button2", "My Button #2",
        @"C:\ExternalCommands.dll", "Revit.Test.Command2");

    // Create a ribbon panel

    RibbonPanel m_projectPanel = application.CreateRibbonPanel(tabName, "This
Panel Name");

    // Add the buttons to the panel

    List<RibbonItem> projectButtons = new List<RibbonItem>();

    projectButtons.AddRange(m_projectPanel.AddStackedItems(button1, button
2));

    return Result.Succeeded;

}

```

Create a New Ribbon Panel and Controls

The following image shows a ribbon panel on the Add-Ins tab 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.

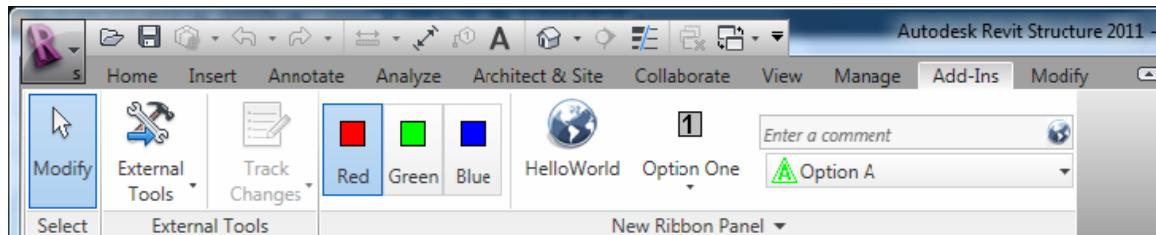


Figure 14: New ribbon panel and controls

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: Ribbon panel and controls

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

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

    return Result.Succeeded;
}
```

Ribbon Panel

Custom ribbon panels can be added to the Add-Ins tab (the default) or the Analyze tab, or they can be added to a new custom ribbon 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.

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.

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.

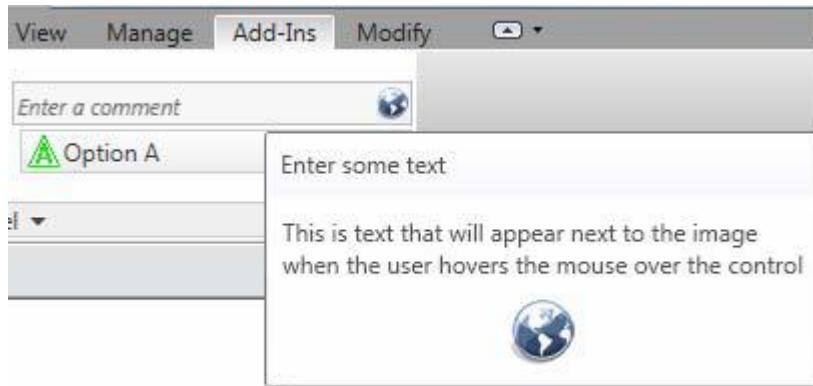
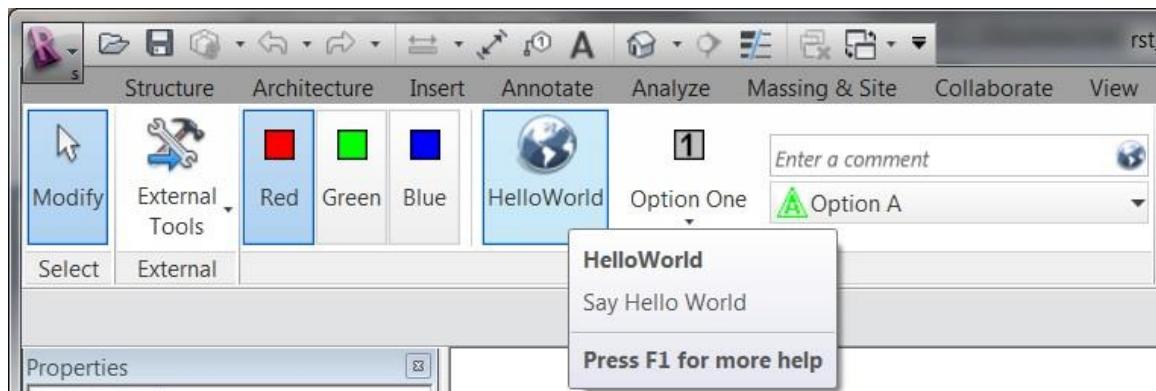


Figure 15: Extended Tooltip

Contextual Help

Controls can have contextual help associated with them. When the user hovers the mouse over the control and hits F1, the contextual help is triggered. Contextual help options include linking to an external URL, launching a locally installed help (chm) file, or linking to a topic on the Autodesk help wiki. The `ContextualHelp` class is used to create a type of contextual help, and then `RibbonItem.SetContextualHelp()` (or `RibbonItemData.SetContextualHelp()`) associates it with a control. When a `ContextualHelp` instance is associated with a control, the text "Press F1 for more help" will appear below the tooltip when the mouse hovers over the control, as shown below.



The following example associates a new `ContextualHelp` with a push button control. Pressing F1 when hovered over the push button will open the Autodesk homepage in a new browser window.

Code Region: Contextual Help

```

private void AddPushButtonWithContextualHelp(RibbonPanel panel)
{
    PushButton pushButton = panel.AddItem(new PushButtonData("HelloWorld",
        "HelloWorld", @"D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll", "Hell
oWorld.CsHelloWorld")) as PushButton;

    // Set ToolTip and contextual help
    pushButton.ToolTip = "Say Hello World";
    ContextualHelp contextHelp = new ContextualHelp(ContextualHelpType.Url,
        "http://www.autodesk.com");
    pushButton.SetContextualHelp(contextHelp);

    // Set the large image shown on button
    pushButton.LargeImage =
        new System.Windows.Media.Imaging.BitmapImage(new Uri(@"D:\Sample\Hell
oWorld\bin\Debug\39-Globe_32x32.png"));
}

```

The ContextualHelp class has a Launch() method that can be called to display the help topic specified by the contents of this ContextualHelp object at any time, the same as when the F1 key is pressed when the control is active. This allows the association of help topics with user interface components inside dialogs created by an add-in application.

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 32×32 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 16×16 pixel image set. Large buttons should also have a 16×16 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 16×16 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.

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.

Ribbon Controls

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

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 14 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: Adding a push button

```
private void AddSimplePushButton(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 System.Windows.Media.Imaging.BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\39-Globe_32x32.png"));
}
```

{}

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.

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 14 is a split button.

Initially, the push button will be the top item in the drop-down list. However, by using the `IsSynchronizedWithSelectedItem` 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 14 above yields:

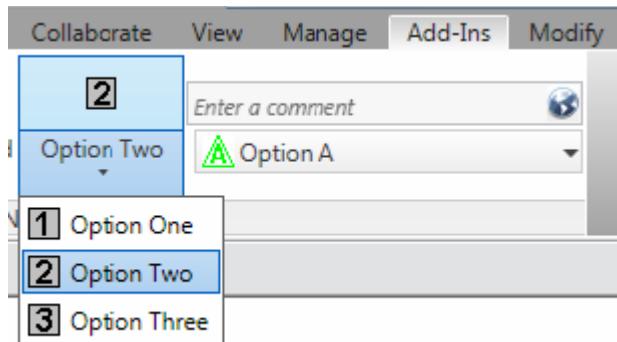


Figure 16: Split button synchronized with current item

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

Code Region: Adding a split button

```
private void AddSplitButton(RibbonPanel panel)
{
    string assembly = @"D:\Sample\HelloWorld\bin\Debug\Hello.dll";
    // Add code here to add the split button to the panel
}
```

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

PushButtonData bTwo = new PushButtonData("ButtonNameB", "Option Two",
assembly, "Hello.HelloTwo");
bTwo.LargeImage =
    new System.Windows.Media.Imaging.BitmapImage(new Uri(@"D:\Sample\He-
lloWorld\bin\Debug\Two.bmp"));

PushButtonData bThree = new PushButtonData("ButtonNameC", "Option Thr-
ee",
assembly, "Hello.HelloThree");
bThree.LargeImage =
    new System.Windows.Media.Imaging.BitmapImage(new Uri(@"D:\Sam-
ple\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);
}
```

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: 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 System.Windows.Media.Imaging.BitmapImage(new Uri
(@"D:\Sample\HelloWorld\bin\Debug\Red.bmp"));

    ToggleButtonData tb2 = new ToggleButtonData("toggleButton2", "Green");

    tb2.ToolTip = "Green Option";

    tb2.LargeImage = new System.Windows.Media.Imaging.BitmapImage(new Uri
(@"D:\Sample\HelloWorld\bin\Debug\Green.bmp"));

    ToggleButtonData tb3 = new ToggleButtonData("toggleButton3", "Blue");

    tb3.ToolTip = "Blue Option";

    tb3.LargeImage = new System.Windows.Media.Imaging.BitmapImage(new Uri
(@"D:\Sample\HelloWorld\bin\Debug\Blue.bmp"));

    radioButtonGroup.AddItem(tb1);
```

```

radioButtonGroup.AddItem(tb2);
radioButtonGroup.AddItem(tb3);
}

```

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 14.

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 14 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: TextBox.EnterPressed event handler

```

void ProcessText(object sender, Autodesk.Revit.UI.Events.TextBoxEnterPressedE
ventArgs 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.

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.

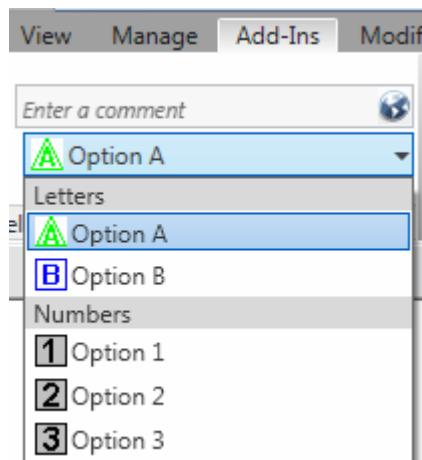


Figure 17: Combo box with grouping

`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
- `DropDownOpened` - 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.

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 14.

Code Region: 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 System.Windows.Media.Imaging.BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\39-Globe_16x16.png"));
    textData.Name = "Text Box";
    textData.ToolTip = "Enter some text here";
    textData.LongDescription = "This is text that will appear next to the
image"
        + "when the user hovers the mouse over the control";
    textData.ToolTipImage =
        new System.Windows.Media.Imaging.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 ComboBoxMemberD
ata("A", "Option A");

        cboxMemDataA.Image =

            new System.Windows.Media.Imaging.BitmapImage
(new Uri(@"D:\Sample\HelloWorld\bin\Debug\A.bmp"));

        cboxMemDataA.GroupName = "Letters";

        cBox.AddItem(cboxMemDataA);





        ComboBoxMemberData cboxMemDataB = new ComboBoxMemberD
ata("B", "Option B");

        cboxMemDataB.Image =

            new System.Windows.Media.Imaging.BitmapImage
(new Uri(@"D:\Sample\HelloWorld\bin\Debug\B.bmp"));

        cboxMemDataB.GroupName = "Letters";
```

```

cBox.AddItem(cboxMemDataB);

ComboBoxMemberData cboxMemData = new ComboBoxMemberDa
ta("One", "Option 1");

cboxMemData.Image =
    new System.Windows.Media.Imaging.BitmapImage
(new Uri(@"D:\Sample\HelloWorld\bin\Debug\One.bmp"));

cboxMemData.GroupName = "Numbers";
cBox.AddItem(cboxMemData);

ComboBoxMemberData cboxMemData2 = new ComboBoxMemberD
ata("Two", "Option 2");

cboxMemData2.Image =
    new System.Windows.Media.Imaging.BitmapImage
(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Two.bmp"));

cboxMemData2.GroupName = "Numbers";
cBox.AddItem(cboxMemData2);

ComboBoxMemberData cboxMemData3 = new ComboBoxMemberD
ata("Three", "Option 3");

cboxMemData3.Image =
    new System.Windows.Media.Imaging.BitmapImage
(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Three.bmp"));

cboxMemData3.GroupName = "Numbers";
cBox.AddItem(cboxMemData3);

}

}

}

```

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.

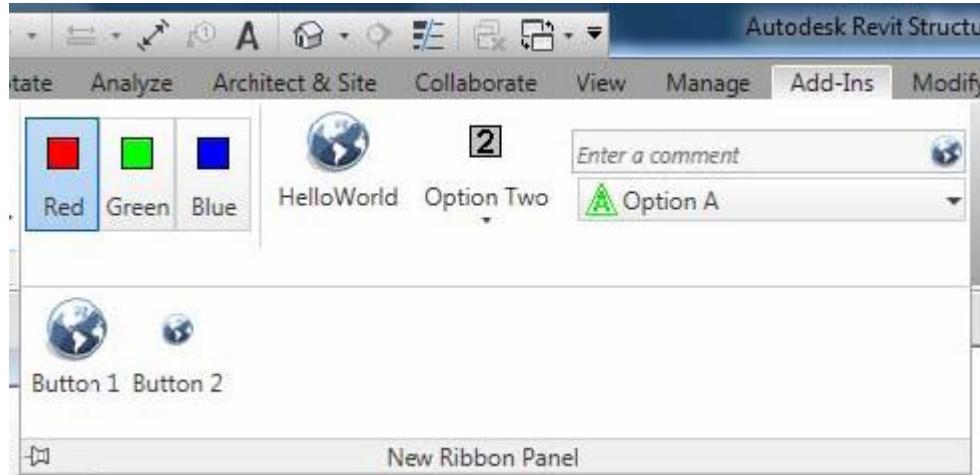


Figure 18:

Slide-out

The following example produces the slide-out shown above:

Code Region: 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 System.Windows.Media.Imaging.BitmapImage(new Uri(@"D:\Sam-
        ple\HelloWorld\bin\Debug\39-Globe_32x32.png"));
    PushButtonData b2 = new PushButtonData("ButtonName2", "Button 2",
        assembly, "Hello.HelloButton");
}
```

```

        assembly, "Hello.HelloTwo");

b2.LargeImage =
    new System.Windows.Media.Imaging.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);

}

```

1.2.10 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.

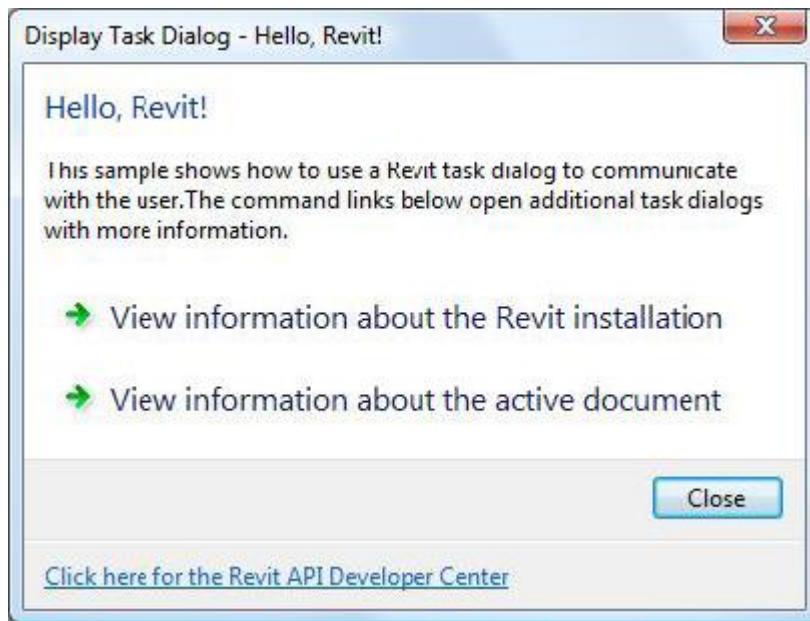


Figure 19: Revit-style Task Dialog

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.

The property `TaskDialog.EnableMarqueeDialogBar` allows the TaskDialog to include a progress bar that has an indeterminate start and stop. The animation continues until the TaskDialog is closed.

The FooterText property can link to the help document and can contain a hyperlink of the form "rvthelptopic:[topic]" to launch Revit's contextual for the topic specified.

Please see [Dialog Guidelines](#) in the [API User Interface Guidelines](#) section 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.ReadOnly)]  
  
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;  
        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 dia-  
loggs with more information.";  
  
  
    // Add commandLink options to task dialog  
  
    mainDialog.AddCommandLink(TaskDialogCommandLinkId.CommandLink  
1,  
                            "View information about the Revit installation");  
  
    mainDialog.AddCommandLink(TaskDialogCommandLinkId.CommandLink  
2,  
                            "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 th-  
e help document.  
  
    mainDialog.FooterText =  
        ""  
        + "Click here for the Rev-  
it API Developer Center";  
  
  
    TaskDialogResult tResult = mainDialog.Show();  
  
  
    // If the user clicks the first command link, a simple Task D-  
ialog
```

```
// with only a Close button shows information about the Revit  
installation.  
  
if (TaskDialogResult.CommandLink1 == tResult)  
{  
  
    TaskDialog dialog_CommandLink1 = new TaskDialog("Revi  
t Build Information");  
  
    dialog_CommandLink1.MainInstruction =  
  
        "Revit Version Name is: " + app.VersionName +  
        "\n"  
  
        + "Revit Version Number is: " + app.VersionNumber +  
        "\n"  
  
        + "Revit Version Build is: " + app.VersionBui  
ld;  
  
  
    dialog_CommandLink1.Show();  
  
}  
  
  
// If the user clicks the second command link, a simple Task  
Dialog  
  
// created by static method shows information about the activ  
e document  
  
else if (TaskDialogResult.CommandLink2 == tResult)  
{  
  
    TaskDialog.Show("Active Document Information",  
  
        "Active document: " + activeDoc.Title + "\n"  
  
        + "Active view name: " + activeDoc.ActiveView.Name);  
  
}  
  
  
return Autodesk.Revit.UI.Result.Succeeded;
```

```

    }
}

```

1.2.11 DB-level External Applications

Database-level add-ins are External Applications that do not add anything to the Revit UI. DB-level External Applications can be used when the purpose of the application is to assign events and/or updaters to the Revit session.

To add a DB-level External Application to Revit, you create an object that implements the `IExternalDBApplication` interface.

The `IExternalDBApplication` interface has two abstract methods, `OnStartup()` and `OnShutdown()`, which you override in your DB-level external application. Revit calls `OnStartup()` when it starts, and `OnShutdown()` when it closes. This is very similar to `IExternalApplication`, but note that these methods return `Autodesk.Revit.DB.ExternalDBApplicationResult` rather than `Autodesk.Revit.UI.Result` and use `ControlledApplication` rather than `UIControlledApplication`.

Code Region: `IExternalDBApplication` `OnShutdown()` and `OnStartup()`

```

public interface IExternalDBApplication
{
    public Autodesk.Revit.DB.ExternalDBApplicationResult OnStartup(ControlledApplication application);

    public Autodesk.Revit.DB.ExternalDBApplicationResult OnShutdown(ControlledApplication application);
}

```

The `ControlledApplication` parameter provides access to Revit [Database Events](../../../../Advanced_Topics/Events/Database_Events.html). Events and Updaters to which the database-level application will respond can be registered in the `OnStartup` method.

1.3 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 [Elements Essentialss](#) and [Editing Elements](#) and refer to the [Views](#) for more details about the view elements.

1.3.1 Application Functions

Application and UIApplication members provide access to application-wide data and settings as well as the active session of Revit.

Application

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

Application Version Information

Application properties include VersionBuild, VersionNumber and VersionName. These can be used to provide add-in behavior based on the release and build of Revit, as shown in [How to use Application properties to enforce a correct version for your add-in](#).

Application-wide Settings

Shared Parameters

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 [Shared Parameters](#).

Library Content

The GetLibraryPaths() and SetLibraryPaths() methods provide access to the path information identifying where Revit searches for content .

Graphical Display

The `BackgroundColor` property allows read and write of the background color to use for model views in this session. The `AllowNavigationDuringRedraw` property enables or disables the option to allow view manipulation during redraw. This can be used to optimize performance during a redraw of the model.

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 [Document and File Management](#).

Session Information

Properties such as `UserName` and methods such as `GetRevitServerNetworkHosts()` provide read-only access to this session specific information.

Login Information

The static `IsLoggedIn` property checks if the user is logged in from this session to their Autodesk A360 account. If they are logged in, the `LoginUserId` property will return the user id of the user currently logged in. (The user id will be empty if the user is not logged in.) Unlike the `UserName` from the section above, the `LoginUserId` value is not a recognizable value, but an internal id. In conjunction with the Store Entitlement REST API, a publisher of an Autodesk app can verify if the current user has purchased their app from the Autodesk App Store. For more information about Store Entitlement API, please refer to www.autodesk.com/developapps.

Shared Parameter Management

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 [Events in the Add-In Integration section](#).

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.

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 [Failure Posting and Handling](#).

Disconnect Warnings

The following properties control whether or not to show the graphical warnings for various types of disconnects.

- ShowGraphicalWarningCableTrayConduitDisconnects
- ShowGraphicalWarningDuctDisconnects
- ShowGraphicalWarningElectricalDisconnects
- ShowGraphicalWarningHangerDisconnects
- ShowGraphicalWarningPipeDisconnects

UIApplication

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

UI Document Management

The UIApplication provides access to the active document using the UIActiveDocument property. Additionally, a Revit document may be opened using the overloaded OpenAndActivateDocument() method. The document will be opened with the default view active. This method may not be called inside a transaction and may only be invoked during an event when no active document is open yet in Revit and the event is not nested inside another event.

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.

Ribbon Panel Utility

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

For more details, see [Ribbon Panel and Controls](#) in the [Add-In Integration](#) section.

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

UI 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 [Events](#) in the [Add-In Integration](#) section.

1.3.1.1 *Discipline Controls*

The properties:

- Application.IsArchitectureEnabled
- Application.IsStructureEnabled
- Application.IsStructuralAnalysisEnabled
- Application.IsMassingEnabled
- Application.IsEnergyAnalysisEnabled
- Application.IsSystemsEnabled
- Application.IsMechanicalEnabled
- Application.IsMechanicalAnalysisEnabled
- Application.IsElectricalEnabled
- Application.IsElectricalAnalysisEnabled
- Application.IsPipingEnabled
- Application.IsPipingAnalysisEnabled

provide read and modify access to the available disciplines. An application can read the properties to determine when to enable or disable aspects of its UI.

When a discipline's status is toggled, Revit's UI will be adjusted, and certain operations and features will be enabled or disabled as appropriate. Enabling an analysis mode will only take effect if the corresponding discipline is enabled. For example, enabling mechanical analysis will not take effect unless the mechanical discipline is also enabled.

1.3.1.2 *How to use Application properties to enforce a correct version for your add-in*

Sometimes you need your add-in to operate only in the presence of a particular Update Release of Revit due to the presence of specific fixes or compatible APIs.

Properties of Application make it possible to check for specific versions of Revit. While the property VersionNumber will return a string representing the primary version number, the VersionBuild property will return the internal build number of the Autodesk Revit application.

Another useful property is the Application.SubVersionNumber property. It returns a string representing the major-minor version number for the Revit application such as "2018.0.0". This string is updated by Autodesk for all major and minor updates. Point releases (such as 2018.1.0) may have additional APIs and functionality not available in the initial customer release (such as 2018.0.0). Add-ins written to support initial releases will likely be compatible with subscription updates, but add-ins using new features in subscription updates would not be compatible with the initial releases.

The following sample code demonstrates a technique to determine if the Revit version is any Update Release after the initial known Revit release.

Code Region: Use VersionBuild to identify if your add-in is compatible

```
public void GetVersionInfo(Autodesk.Revit.ApplicationServices.Application app)
{
    // 20110309_2315 is the datecode of the initial release of Revit 2012
    if (app.VersionNumber == "2012" &&
        String.Compare(app.VersionBuild, "20110309_2315") > 0)
    {
        TaskDialog.Show("Supported version",
            "This application supported in this version.");
    }
    else
    {
        TaskDialog dialog = new TaskDialog("Unsupported version.");
        dialog.MainIcon = TaskDialogIcon.TaskDialogIconWarning;
        dialog.MainInstruction = "This Revit 2012 application is supported in UR
1 and later releases.";
        dialog.Show();
    }
}
```

1.3.2 Document Functions

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

Document

The Document class represents an open Autodesk Revit project.

Settings Property

The Settings property returns an object that provides access to general components within Revit projects. For more details, see [Settings](#).

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 [Place and Locations](#).

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 [Views](#).

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 [Elements Essentials](#).

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 [Document and File management](#).

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 [Editing Elements](#).

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 [Events](#).

Document Status

Several properties provide information on the status of the document:

- IsModifiable - whether the document may currently be modified (meaning that is there is an active transaction in the document and changes are not temporarily blocked by anything else)
- IsModified - whether the document was changed since it was opened or saved
- IsReadOnly - if true, the document is currently read-only and cannot 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

Others

Document also provides other functions:

- ParameterBindings Property - Mapping between parameter definitions and categories. For more details, see [Shared Parameters](#).
- ReactionsAreUpToDate Property - Reports whether the reactionary loads changed. For more details, see [Loads](#).
- Default Types - Access to the default types for family and non-family elements. For more details, see [Default Types](#).

UIDocument

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

Element Retrieval in UIDocument

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 [Elements Essentials](#).

Element Display

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

View Management in UIDocument

The UIDocument class can be used to refresh the active view in the active document by calling the RefreshActiveView() method. The ActiveView property can be used to retrieve or set the active view for the document. Changing the active view has some restrictions. It can only be used in an active document, which must not be in read-only state and must not be inside a transaction. Additionally, the active view may not be changed during the ViewActivating or ViewActivated event, or during any pre-action event, such as DocumentSaving, DocumentClosing, or other similar events.

The UIDocument.ActiveGraphicalView property retrieves the active graphical view for the document. Unlike UIDocument.ActiveView, this property will never return auxiliary views like the Project Browser or System Browser if the user has happened to make a selection in one of those views.

The UIDocument can also be used to get a list of all open view windows in the Revit user interface. The GetOpenUIViews method returns a list of UIViews which contain data about the view windows in the Revit user interface.

1.3.2.1 Default Types

Revit has a default type for different categories. This default type is shown in the Revit User Interface when the related tool is invoked to create an element of this category. The Revit API exposes the default type for both family and non-family types via the Document class.

Family Types

The document members listed in the table below provide access to the default type for a given family category id

Document Method	Description
GetDefaultFamilyTypeId()	Gets the default family type id associated to the given family category id.
SetDefaultFamilyTypeId()	Sets the default family type id associated to the given family category id.
IsDefaultFamilyTypeIdValid()	Checks whether the family type id is valid to set as default for the given family category id.

Additionally, given an ElementType, the ElementType.IsValidDefaultFamilyType() identifies if it is a valid default family type for the given family category id.

The example below demonstrates how to get the default family type Id for the structural column category. It then gets the family symbol for the default type and assigns it to a given column.

Code Region: Get default family type id

```
private void AssignDefaultTypeToColumn(Document document, FamilyInstance column)

{
    ElementId defaultTypeId = document.GetDefaultFamilyTypeId(new ElementId(BuiltInCategory.OST_StructuralColumns));

    if (defaultTypeId != ElementId.InvalidElementId)
    {
        FamilySymbol defaultType = document.GetElement(defaultTypeId) as FamilySymbol;

        if (defaultType != null)
        {
            column.Symbol = defaultType;
        }
    }
}
```

The next example sets the default type for the doors category from a given door, after first checking that it is a valid default family type id.

Code Region: Set default family type id

```
private void SetDefaultTypeFromDoor(Document document, FamilyInstance door)

{
    ElementId doorCategoryId = new ElementId(BuiltInCategory.OST_Doors);
```

Code Region: Set default family type id

```
// It is necessary to test the type suitability to be a default family type, for not every type can be set as default.

// Trying to set a non-qualifying default type will cause an exception

if (door.Symbol.IsValidDefaultFamilyType(doorCategoryId))

{

    document.SetDefaultFamilyTypeId(doorCategoryId, door.Symbol.Id);

}

}
```

Non-family Types

The document members in the table below provide access to the default types for non-Family element types.

Document Method	Description
GetDefaultElementTypeId()	Gets the default element type id for a given non-Family element type.
SetDefaultElementTypeId()	Sets the default element type id for a given non-Family element type.
IsDefaultElementTypeIdValid()	Checks whether the element type id is valid for a given non-Family element type.

The example below checks whether a given wall is using the default element type for wall types.

Code Region: Get default element type id

```
private bool IsWallUsingDefaultType(Document document, Wall wall)
```

```

    {
        ElementId defaultElementTypeId = document.GetDefaultElementTypeId(ElementTypeGroup.WallType);

        return (wall.WallType.Id == defaultElementTypeId);
    }
}

```

1.3.3 Document and File Management

Document and file management make it easy to create and find your documents. For information about cloud-based Revit files, see [Cloud Models](#).

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.

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.

Basic File Info

BasicFileInfo provides fast access to basic information about a Revit file, including worksharing status, Revit version, username and central path. This is done without fully opening the file.

```
static BasicFileInfo.Extract(string file)
```

Returns a BasicFileInfo object whose properties provide information about the file.

Extract is not forward-compatible, meaning that Calling Extract from a version of Revit prior to Revit 2019 will result in an exception if it is called on a Revit 2019 or later file. This may occur again with future versions of Revit.

Open a Document

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

Table 3: Open Document in API

Method	Event
<code>Document OpenDocumentFile(string filename)</code>	<code>DocumentOpened</code>
<code>Document OpenDocumentFile(ModelPath modelPath, OpenOptions openOptions)</code>	
<code>Document OpenDocumentFile(ModelPath modelPath, OpenOptions openOptions, IOpenFromCloudCallback openFromCloudCallback)</code>	

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 second overload takes a path to the model as a ModelPath rather than a string and the OpenOptions parameter offers options for opening the file, such as the ability to detach the opened document from central if applicable, as well as options related to worksharing. For more information about opening a workshared document, see [Opening a Workshared Document](#).

These methods throw specific documented exceptions in the event of a failure. Exceptions fall into 4 main categories.

Table 4: Types of exceptions thrown

Type	Example
Disk errors	File does not exist or is wrong version
Resource errors	Not enough memory or disk space to open file
Central model file errors	File is locked or corrupt
Central model/server errors	Network communication error with server

If the document is opened successfully, the DocumentOpened event is raised.

Create a Document

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

Table 5: Create Document in the API

Method	Event
<code>Document NewProjectDocument(string templateFileName);</code>	DocumentCreated
<code>Document NewProjectDocument(UnitSystem unitSystem);</code>	DocumentCreated
<code>Document NewFamilyDocument(string templateFileName);</code>	DocumentCreated
<code>Document NewProjectTemplateDocument(string templateFilename);</code>	DocumentCreated

For the methods that require a template file name as a parameter, the created document is returned based on the template file. `NewProjectDocument(UnitSystem)` creates a new imperial or metric project document without a specified template.

Save and Close a Document

The Document class provides methods to save or close instances.

Table 6: Save and Close Document in API

Method	Event
<code>Save()</code>	DocumentSaved
<code>SaveAs()</code>	DocumentSavedAs
<code>Close()</code>	DocumentClosed

`Save()` has 2 overloads, one with no arguments and one with a `SaveOptions` argument that can specify whether to force the OS to delete all dead data from the file on disk. If the file has not been previously saved, `SaveAs()` must be called instead.

`SaveAs()` has 3 overloads. One overload takes only the filename as an argument and an exception will be thrown if another file exists with the given filename. The other 2 overloads takes a filename as an argument (in the form of a `ModelPath` in one case) as well as a second `SaveAsOptions` argument that can be used to specify whether to overwrite an existing file, if it exists. `SaveAsOptions` can also be used to specify other relevant options such as whether to remove dead data on disk related to the file and worksharing options.

`Save()` and `SaveAs()` throw specific documented exceptions in the same 4 categories as when opening a document and listed in Table 4 above.

`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. This method will throw an exception if the document's path name is not already set or if the saving target file is read-only.

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.

The `UIDocument` class also provides methods to save and close instances.

Table 7: Save and Close `UIDocument` in API

Method	Event
<code>SaveAndClose()</code>	<code>DocumentSaved</code> , <code>DocumentClosed</code>
<code>SaveAs()</code>	<code>DocumentSavedAs</code>

`SaveAndClose()` closes the document after saving it. If the document's path name has not been set the "Save As" dialog will be shown to the Revit user to set its name and location.

The `SaveAs()` method saves the document to a file name and path obtained from the Revit user via the "Save As" dialog.

Document Preview

The `DocumentPreviewSettings` class can be obtained from the `Document` and contains the settings related to the saving of preview images for a given document.

Code Region: Document Preview

```
public void SaveActiveViewWithPreview(UIApplication application)
{
    // Get the handle of current document.
    Autodesk.Revit.DB.Document document = application.ActiveUIDocument.Document;
```

```
// Get the document's preview settings

DocumentPreviewSettings settings = document.GetDocumentPreviewSettings();

// Find a candidate 3D view

FilteredElementCollector collector = new FilteredElementCollector(document);

collector.OfClass(typeof(View3D));

Func<View3D, bool> isValidForPreview = v => settings.IsViewIdValidForPreview(v.Id);

View3D viewForPreview = collector.OfType<View3D>().First<View3D>(isValidForPreview);

// Set the preview settings

using (Transaction setTransaction = new Transaction(document, "Set preview view id"))

{

    setTransaction.Start();

    settings.PreviewViewId = viewForPreview.Id;

    setTransaction.Commit();

}

// Save the document

document.Save();

}
```

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 [Family](#).

1.3.4 ForgeTypeld

`Autodesk.Revit.DB.ForgeTypeld` is an extensible identifier for a unit, symbol, or other object, and is used throughout the Revit API to identify data such as units of measurement, symbols, and unit types. Unit types are referred to as "specs" to avoid confusion with units themselves.

A `ForgeTypeld` instance holds a string, called a "typeid", that uniquely identifies a Forge schema. A Forge schema is a JSON document describing a data structure, supporting data interchange between applications. A typeid string includes a namespace and version number such as

- `autodesk.spec.aec:length-1.0.0`
- `autodesk.unit.unit:meters-1.0.0`
- `autodesk.revit.category.local:walls-1.0.0`

By default, comparison of `ForgeTypeld` values in the Revit API ignores the version number.

The members of the following classes have a `ForgeTypeld` data type:

- `DisciplineTypeld` - Product disciplines used in the Revit UI such as Architecture, Electrical, HVAC, Piping, and Structural
- `GroupTypeld` - Groupings used for parameters in the Revit UI such as Construction, General, Geometry, IdentityData
- `ParameterTypeld` - Type of parameters such as AllModellInstanceComments, InstanceSillHeightParam, WallTopOffset
- `SpecTypeld` - Type of quantity to be measured such as Area, Currency, HvacDensity, Wattage
- `SymbolTypeld` - Unit symbol displayed in the formatted string representation of a number to indicate the units of the value, such as DegreeC, Ft, KipPerFt, MSup2
- `UnitTypeld` - Units and display format used to format numbers as strings or convert units such as Acres, Degrees, Feet, Liters

For example, the following properties all have a `ForgeTypeld` data type:

- `DisciplineTypeld.Architecture`
- `GroupTypeld.Constraints`
- `ParameterTypeld.WallTopOffset`
- `SpecTypeld.Volume`
- `SymbolTypeld.Hour`
- `UnitTypeld.Millimeters`

These static methods convert a `BuiltInCategory` to a `ForgeTypeld` and vice versa.

- `Category.GetBuiltInCategoryTypeld(BuiltInCategory)`
- `Category.GetBuiltInCategory(ForgeTypeld)`

For example, `Category.GetBuiltInCategoryId(BuiltInCategory.OST_Furniture)` returns a `ForgeTypeId` with a `TypeId` equal to `autodesk.revit.category.family:furniture-1.0.0`

1.3.5 Import Functions

The overloads for the `Document.Import` method allow several different file types to be imported.

CAD file import

- `Import(String, SATImportOptions, View)` - Imports an SAT file into the document.
- `Import(String, SKPImportOptions, View)` - Imports an SKP file into the document.
- `Import(String, DGNIImportOptions, View, out ElementId)` - Imports a DGN file to the document.
- `Import(String, DWGImportOptions, View, out ElementId)` - Imports a DWG or DXF file to the document.

GBXML import

- `Import(String, GBXMLImportOptions)`

Images and PDF import

Revit can import images (JPG, PNG, etc) and raster images generated from PDF files.

The class `ImageInstance` represents an instance of an `ImageType` placed in a view. Its members include:

- `Create(Document, View, ElementId, ImagePlacementOptions)` - The `ImagePlacementOptions` describes where an image instance should be placed in a view.
- `GetLocation(BoxPlacement)` - The `BoxPlacment` enumeration is used to specify which corner (or the center) of the image the location pertains to.
- `SetLocation(XYZ, BoxPlacement)`
- property `Width`
- property `Height`
- property `EnableSnaps`

The class `ImageType` represents a type containing an image. Its members include:

- `Create(Document, ImageTypeOptions)`
- `CanReload()` - validates the corresponding image or PDF file and performs additional validation if the file is served by an external provider.
- `ReloadFrom(ImageTypeOptions)`
- property `ExternalResourceType` - the type of external resources that represents images (and PDF files).

- property PageNumber
- property PathType - the type of path that was used to refer to the file from which the ImageType was loaded.

The class `ImageTypeOptions` represents the options that are used when creating or reloading an ImageType, which contains image data corresponding to an image or PDF file. Its members include:

- `IsValid()` - tests whether `ImageTypeOptions` can be used to create or reload an `ImageType`; additional validation is performed for PDF files and external files.
- property PageNumber
- property Path
- property Resolution - is measured in dpi and relates the number of pixels in raster images to their size.
- `setExternalResourceReference()` - specifies the location of an image or PDF file using an external resource reference.

`ImageTypeOptions` can be used to specify a local file or a file served by an external provider. Local files can be referred to using relative paths. For PDF files, `ImageType` loads a single page at a time, which is rasterized at the resolution specified in `ImageTypeOptions`.

The class `ImagePlacementOptions` is used to describe where an `ImageInstance` should be placed in a view

- `ImagePlacementOptions()` constructs a new `ImagePlacementOptions` that will place an `ImageInstance` with its center at the origin of the model
- `ImagePlacementOptions(XYZ, BoxPlacement)` constructs a new `ImagePlacementOptions` with the provided Location and PlacementPoint
- Location specifies where a point of the `ImageInstance`, determined by the PlacementPoint property, is going to be inserted.
- PlacementPoint uses a `BoxPlacement` to identify which point of the `ImageInstance` will be aligned to the Location

Converting images between links and imports

`ImageTypeOptions.SourceType` along with the new enumerated value `ImageTypeSource` and `ImageType.ReloadFrom()` allow you to create or convert an `ImageType` to a link or import.

The overall process is:

- Create a new `ImageTypeOptions` instance from the existing `ImageType` properties
- Modify the `ImageTypeOptions`, for example by changing the `ImageTypeOptions.SourceType` between Link and Import, or change the path with `ImageTypeOptions.SetPath()`
- Use those new `ImageTypeOptions` in `ImageType.ReloadFrom`

`ImageType` properties include:

- `ImageType.Source` - Indicates how the image is created (as a link, import, or internally-generated image)
- `ImageType.Status` - Indicates whether the image is loaded or unloaded (if applicable)

Two constructors take an `ImageTypeSource` as an argument:

- `ImageTypeOptions(String, Boolean, ImageTypeSource)`
- `ImageTypeOptions(ExternalResourceReference, ImageTypeSource)`

The enumeration: `ImageTypeStatus` contains possible values for the load status of an `ImageType`.

`OptionalFunctionalityUtils.isPDFImportAvailable()` checks whether the installed Revit contains optional modules that are necessary for the PDF import functionality.

Rhino

`Document.Import(String, ImportOptions3DM, View)` and `Document.Link(String, ImportOptions3DM, View)` import or link a 3DM file into the document.

1.3.6 Settings

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

Table 7: Settings in API and UI

UI command	Associated API	Reference
Settings → Project Information	<code>Document.ProjectInformation</code>	See the following note
Settings → Project Parameters	<code>Document.ParameterBindings</code> (Only for Shared Parameter)	See Shared Parameters
Project Location panel	<code>Document.ProjectLocations</code> <code>Document.ActiveProjectLocation</code>	Place and Locations
Settings → Additional Settings → Fill Patterns	FilteredElementCollector filtering on class <code>FillPatternElement</code>	See the following note
Settings → Materials	FilteredElementCollector filtering on class <code>Material</code>	See Material Management

Settings → Object Styles	Document.Settings.Categories	See the following note
Phasing → Phases	Document.Phases	See the following note
Settings → Structural Settings	Loads and related structural settings are available in the API	See Structural Engineering
Settings → Project Units	Document.GetUnits()	See Units
Area and Volume Calculations (on the Room & Area panel)	AreaVolumeSettings.GetAreaVolumeSettings()	See the following note

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.

Table 8: ProjectInformation

Parameters	Corresponding API	Built-in Parameters
Project Issue Date	ProjectInfo.IssueDate	PROJECT_ISSUE_DATE
Project Status	ProjectInfo.Status	PROJECT_STATUS
Client Name	ProjectInfo.ClientName	CLIENT_NAME
Project Address	ProjectInfo.Address	PROJECT_ADDRESS
Project Name	ProjectInfo.Name	PROJECT_NAME
Project Number	ProjectInfo.Number	PROJECT_NUMBER

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 [Parameter](#) in the [Elements Essentials](#) section. The recommended way to get project information is to use the ProjectInfo properties.

- Fill Patterns - Retrieve all Fill Patterns in the current document using a FilteredElementCollector filtering on class FillPatternElement. Specific FillPatterns can be

retrieved using the static methods `FillPatternElement.GetFillPattern(Document, ElementId)` or `FillPatternElement.GetFillPatternByName (Document, string)`.

- Object Styles - Use `Settings.Categories` to retrieve all information in Category objects except for Line Style. For more details, see [Other Classifications](#) in the [Elements Essentials](#) and [Material](#) sections.
- 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 `AreaVolumeSettings` class allows you to enable or disable volume calculations, and to change the room boundary location.

1.3.7 Units

The two main classes in the Revit API for working with units are `Units` and `FormatOptions`. The `Units` class represents a document's default settings for formatting numbers with units as strings. It contains a `FormatOptions` object for each unit type as well as settings related to decimal symbol and digit grouping.

The `Units` class stores a `FormatOptions` object for every valid unit type, but not all of them can be directly modified. Some, like `SpecTypeId.Number` and `SpecTypeId.SiteAngle`, have fixed definitions. Others have definitions which are automatically derived from other unit types. For example, `SpecTypeId.SheetLength` is derived from `SpecTypeId.Length` and `SpecTypeId.ForceScale` is derived from `SpecTypeId.Force`.

The `FormatOptions` class contains settings that control how to format numbers with units as strings. It contains those settings that are typically chosen by an end-user in the Format dialog and stored in the document, such as rounding, accuracy, display units, and whether to suppress spaces or leading or trailing zeros.

The `FormatOptions` class is used in two different ways. A `FormatOptions` object in the `Units` class represents the default settings for the document. A `FormatOptions` object used elsewhere represents settings that may optionally override the default settings.

The `UseDefault` property controls whether a `FormatOptions` object represents default or custom formatting. If `UseDefault` is true, formatting will be according to the default settings in the `Units` class, and none of the other settings in the object are meaningful. If `UseDefault` is false, the object contains custom settings that override the default settings in the `Units` class. `UseDefault` is always false for `FormatOptions` objects in the `Units` class.

Unit Conversion

The Revit API provides utility classes to facilitate working with quantities in Revit. The `UnitUtils` class makes it easy to convert unit data to and from Revit's internal units.

The UnitUtils class offers a set of methods for mapping between enumeration values and ForgeTypeId values to assist clients in migrating code to ForgeTypeId such as:

- UnitUtils.GetDiscipline()
- UnitUtils.IsMeasurableSpec()
- UnitUtils.IsSymbol()

Revit has seven base quantities, each with its own internal unit. These internal units are identified in the following table.

Table 9: 7 Base Units in Revit Unit System

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

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 / s². The following example uses the UnitUtils.ConvertFromInternalUnits() method to get the minimum yield stress for a material in kips per square inch.

Code Region: Converting from Revit's internal units

```
double GetYieldStressInKsi(Material material)
{
    double dMinYieldStress = 0;
```

```
// Get the structural asset for the material

ElementId strucAssetId = material.StructuralAssetId;

if (strucAssetId != ElementId.InvalidElementId)

{

    PropertySetElement pse = material.Document.GetElement(strucAssetId) as PropertySetElement;

    if (pse != null)

    {

        StructuralAsset asset = pse.GetStructuralAsset();




        // Get the min yield stress and convert to ksi

        dMinYieldStress = asset.MinimumYieldStress;

        dMinYieldStress = UnitUtils.ConvertFromInternalUnits(dMinYieldStress,

UnitTypeId.KipsPerSquareInch);

    }

}

return dMinYieldStress;

}
```

The UnitUtils can also be used to convert a value from one unit type to another, such as square feet to square meters. In the following example, a wall's top offset value that was entered in inches is converted to feet, the expected unit for setting that value.

Code Region: Converting between units

```
void SetTopOffset(Wall wall, double dOffsetInches)

{
```

```

// convert user-defined offset value to feet from inches prior to setting
double dOffsetFeet = UnitUtils.Convert(dOffsetInches,
                                         UnitTypeId.Inches,
                                         UnitTypeId.Feet);

Parameter paramTopOffset = wall.GetParameter(ParameterTypeId.WallTopOffset);
paramTopOffset.Set(dOffsetFeet);
}

```

Unit formatting and parsing

Another utility class, UnitFormatUtils, can format data or parse formatted unit data.

The overloaded method Format() can be used to format a value into a string based on formatting options as the following example demonstrates. The material density is retrieved and then the value is then converted to a user-friendly value with unit using the Format() method.

Code Region: Format value to string

```

void DisplayDensityOfMaterial(Material material)
{
    double density = 0;

    // get structural asset of material in order to get the density
    ElementId strucAssetId = material.StructuralAssetId;
    if (strucAssetId != ElementId.InvalidElementId)
    {
        PropertySetElement pse = material.Document.GetElement(strucAssetId) as PropertySetElement;
        if (pse != null)
        {

```

```

    StructuralAsset asset = pse.GetStructuralAsset();

    density = asset.Density;

    // convert the density value to a user readable string that includes the units

    Autodesk.Revit.DB.Units units = material.Document.GetUnits();

    // false for maxAccuracy means accuracy specified by the FormatOptions should be used

    // false for forEditing since this will be for display only and no formatting modifications are necessary

    string strDensity = UnitFormatUtils.Format(units, SpecTypeId.UnitWeight, density, false);

    string msg = string.Format("Raw Value: {0}\r\nFormatted Value: {1}", density, strDensity);

    TaskDialog.Show("Material Density", msg);

}

}

}

```

The overloaded UnitFormatUtils.TryParse() method parses a formatted string, including units, into a value if possible, using the Revit internal units of the specified unit type. The following example takes a user entered length value, assumed to be a number and length unit, and attempts to parse it into a length value. The result is compared with the input string in a TaskDialog for demonstration purposes.

Code Region: Parse string

```

double GetLengthInput(Document document, String userInputLength)

{
    double dParsedLength = 0;

    Autodesk.Revit.DB.Units units = document.GetUnits();

```

```

// try to parse a user entered string (i.e. 100 mm, 1'6")
bool parsed = UnitFormatUtils.TryParse(units, SpecTypeId.Length, userInputLength, out dParsedLength);

if (parsed == true)

{
    string msg = string.Format("User Input: {0}\r\nParsed value: {1}", userInputLength, dParsedLength);

    TaskDialog.Show("Parsed Data", msg);
}

return dParsedLength;
}

```

1.3.8 Cloud Models

Basic Info

- Document.IsModelInCloud indicates if the current document is located in the cloud.
- Document.GetCloudModelPath() returns the cloud model path.

Autodesk provides two different BIM 360 web portals and regions with different URLs. You can save your Revit cloud models to either:

- Autodesk Docs US, at insight.b360.autodesk.com
- Autodesk Docs EU, at insight.b360.eu.autodesk.com

The property

- ModelPath.Region

returns the region of the account and project which contains this model. `ModelPathUtils.CloudRegionUS` and `ModelPathUtils.CloudRegionEMEA` return the region names of different Autodesk cloud services. They can be used as the first argument of the `ModelPathUtils.ConvertCloudGUIDsToCloudPath()` method.

Open

To open a cloud-hosted model with `Application.OpenDocumentFile` a `ModelPath` is needed. Such a `ModelPath` is returned by the method `ModelPathUtils.ConvertCloudGUIDsToCloudPath()` whose inputs are a region, ProjectGUID, and ModelGUID.

- `Document.CloudModelGUID` returns the Model GUID if it is stored in the cloud.
- `ModelPath.GetProjectGUID()` returns the Project GUID.
- `Document.GetWorksharingCentralModelPath()` returns the model path of the central model.

Getting the CloudPath for a Model

The region argument for `ConvertCloudGUIDsToCloudPath` is a string type and should be either "US" or "EMEA", depending on which BIM 360 or Autodesk Docs region account and project the model is stored in.

- US - insight.b360.autodesk.com - `ModelPathUtils.CloudRegionUS`
- EU - insight.b360.eu.autodesk.com - `ModelPathUtils.CloudRegionEMEA`

Depending on where the cloud model is stored, provide the appropriate region argument "US" or "EMEA", respectively.

To get a valid CloudPath with the Revit API call `ModelPathUtils.ConvertCloudGUIDsToCloudPath()`. You will need to register a Forge application and use the Forge Data Management API to get the project Guid and model Guid as the other two arguments.

The [Forge DM API reference on GET hubs](#) shows how to list the hubs your Forge application can access. You can filter out the accounts of interest using the `data.attributes.name` field. You can also get the region information here.

The [Forge DM API reference on GET project folder contents](#) shows how to list the folder contents you plan to open. You can filter out the relevant cloud model using the `included.attributes.name` field; the project Guid and model Guid information is provided in the `included.attributes.extension.data` field.

With these three pieces of information - region, project guid, and model guid - you can obtain a valid cloud path with the `ModelPathUtils.ConvertCloudGUIDsToCloudPath` method and then open the model with the `OpenDocument` or `OpenAndActivateDocument` methods.

Getting the Forge ID for a Model

These methods allow you to identify the Forge IDs for Cloud Models:

- `Document.GetHubId()`: ForgeDM hub id where the model is located
- `Document.GetProjectId()`: ForgeDM project id where the model is located
- `Document.GetCloudFolderId(bool forceRefresh)`: ForgeDM folder id where the model is located
- `Document.GetCloudModelUrn()`: A ForgeDM Urn identifying the model They return strings which will be empty for a document which is not a cloud model.

IOpenFromCloudCallback

An implementation of interface `IOpenFromCloudCallback` can be specified to control Revit's behavior when opening the cloud model. `IOpenFromCloudCallback.OnOpenConflict` method is called when a conflict happens during the model opening.

It receives a value of enum `OpenConflictScenario` that describes the conflict:

- Rollback indicates that the Central model is restored to an earlier version.
- Relinquished indicates that Ownership to model elements is relinquished.
- OutOfDate indicates that the model is out of date
- VersionArchived indicates that last central version merged into the local model to open has been archived in the central model. Editing is limited to elements and worksets the user owns until Reload Latest or Synchronize with Central is conducted after the model is opened.

And it returns a value of enum `OpenConflictResult` that describes the action Revit should take:

- KeepLocalChanges - Keeps the local changes and open the model
- DiscardLocalChangesAndOpenLatestVersion - Discard the local changes and open the latest version of the model
- DetachFromCentral - Detach the local model from its central model, with worksets preserved
- Cancel

`DefaultOpenFromCloudCallback` class is the default callback used by an overload of the `Application.OpenDocumentFile` method. `DiscardLocalChangesAndOpenLatestVersion` is returned for all kinds of conflicts.

Save

- `Document.SaveAsCloudModel()` saves the current model as a cloud model in BIM 360 and supports upload of local workshared file into BIM 360 Design as a Revit Cloud Worksharing central model.
- `Document.SaveCloudModel()` saves the current cloud model.

To save a local Revit file to the cloud as a workshared or non-workshared cloud model, you need to get the BIM 360 or Autodesk Docs account id, project id, folder id, and a model name. There are two ways to retrieve this information:

1. From the web browser
2. Using the Forge DM API

SaveAsCloudModel Information from the Web Browser

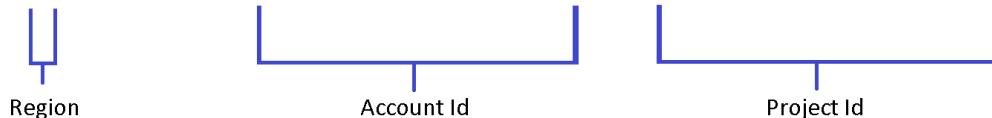
Open a web browser, navigate to your project home page, and copy the URL:



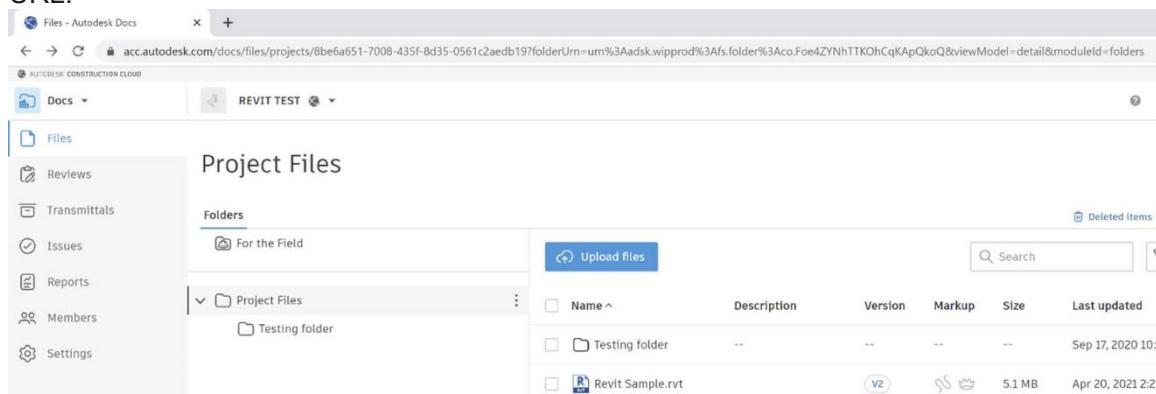
The account id and project id are both GUID strings.

They can be extracted from the URL like this:

insight.b360.eu.autodesk.com/accounts/c0bc50f0-3a89-4792-ab12-949cffb151ce/projects/1fa5d5b7-ef5d-4744-bb18-3a11bcd72b48/home



Navigate to your target Autodesk Docs folder in the web browser and copy the URL:



The folder id is embedded in this URL; in this example, it is "urn:adsk.wipprod:fs.folder:co.Foe4ZYNhTTKOhCqKApQkoQ":

acc.autodesk.com/docs/files/projects/8be6a651-7008-435f-8d35-0561c2aedb19?folderUrn=urn:adsk.wipprod:fs.folder.co.Foe4ZYNhTTKOhCqKApQkoQ&viewMode=detail&moduleId=folders



With this information, you can save a local file which is opened in Revit to Autodesk document management as a cloud model with a call like this:

```
public void SaveAsCloud(Document currentDocument)
{
    Guid account = new Guid("a8d3b76e-cf23-4dd7-a090-9e893efcf949");
    Guid project = new Guid("bf46f5e3-285e-496f-be03-b5b1f8b1e154");
```

```

string folder_id = "urn:adsk.wipemea:fs.folder:co.Jo68ieLRRcKvQr4fI2Q8uQ";

string model_name = "rac_advanced_sample_project.rvt";


currentDocument.SaveAsCloudModel(
    account, // account id
    project, // project id
    folder_id, // folder id
    model_name // model name
);

}

```

SaveAsCloudModel Information with Forge DM API

With your Forge application, you can:

- List hubs with the [GET hubs API](#) to retrieve the region and account ids.
- List projects the [GET projects API](#) to get all the projects of the given hub and their project ids.
- List the top folders with the [GET top folders API](#) to get all accessible top folders (depending on permission) and you their valid folder ids, or continue to get the nested folders with the [list folder contents API](#) until the target folder and its folder id is found and can be stored for later use.

With this information, you can save a local file opened in Revit to Autodesk document management as a cloud model using the same Revit API call as above.

1.4 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.

1.4.1 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.

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 [Family Instances](#).
- 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 [Walls, Floors, Roofs and Openings](#).
- Structure Elements. - Structure Elements contain elements that are only used in the structural features of Revit. For more information about Structure Elements, see [Structural Engineering](#).

View Elements

View Elements represent the way you view and interact with other objects in Revit. For more information, see [Views](#).

Group Elements

Group Elements represent the assistant Elements such as Array and Group objects in Revit. For more information, see [Editing Elements](#).

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 [Annotation Elements](#).

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. NoteFor more information about Common Datum Elements and Datum FamilyInstance, see [Datum and Information Elements](#); for ModelCurve related contents, see [Sketching](#).
 - 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 [Structural Engineering](#).

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 [Sketching](#).

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 [Datum and Information Elements](#).

1.4.2 Other Classifications

Elements can be classified by Category, Family, Symbol and Instance.

There are some relationships between the classifications. For example:

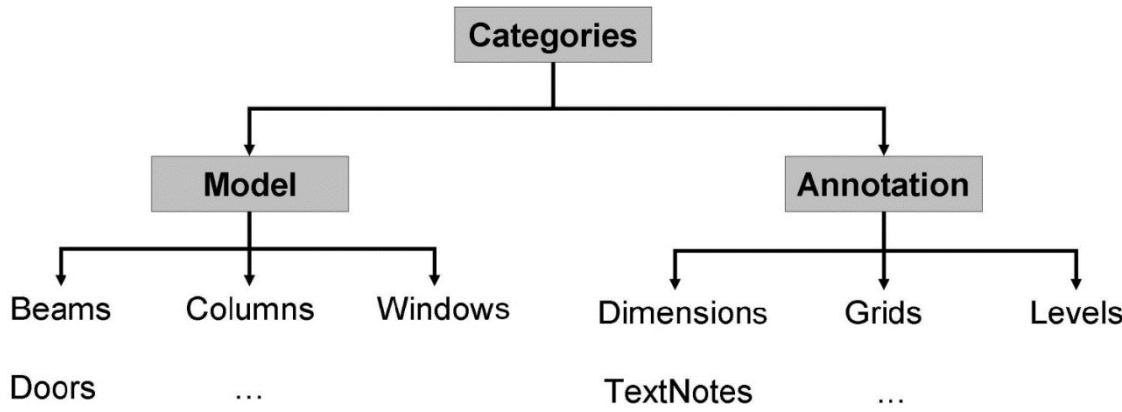
- You can distinguish different kinds of FamilyInstances by the category. Items such as structural columns are in the Structural Columns category, beams and braces are in the Structural Framing category, and so on.
- You can differentiate structural FamilyInstance Elements by their symbol.

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 20: 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. ModellLine and DetailLine are in the Lines category.

To gain access to the categories you may access all categories from the document's Settings class (for example, to insert a new category set), or if you only need access to a category object associated with a built-in category, you may access the category object directly from the static overloaded GetCategory() method of the Category class.

To access categories:

- Get an entire map of Categories from the document properties: Document.Settings.Categories returns a CategoryNameMap containing a map of all Revit categories indexed by their name. `Category.VisibileInUI` returns true if the category is visible to the user in lists of categories in the Revit user interface (dialogs such as Visibility Graphics or View Filters)
- Get a specific built-in category by calling the appropriate overload of the static method Category.GetCategory().
- Get a specific category or subcategory by its ElementId by calling the corresponding overload of the static method Category.GetCategory().

Code Region 5-1: Getting categories from document settings

```
public void GetCategories(Document document)
{
    // Get settings of current document
    Settings documentSettings = document.Settings;

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

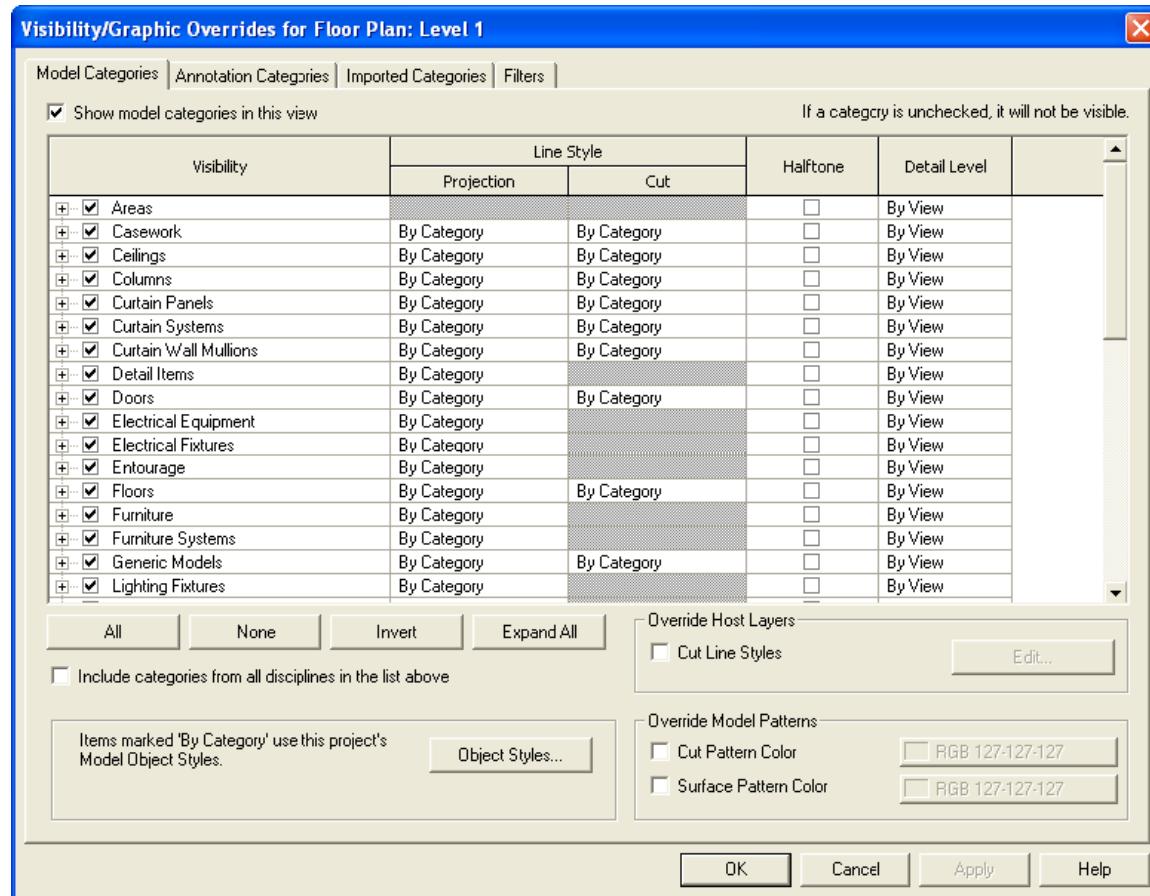
    // Show the number of all the categories to the user
    String prompt = "Number of all categories in current Revit document:" + groups.Size;

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

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 21: 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. `LabelUtils.GetLabelFor(BuiltInCategory)` returns the string name of the given `BuiltInCategory` in the current Revit language.
- If the category is not built-in, the ID is converted to a null value.

Code Region 5-2: Getting element category

```
public void GetElementCategory(UI.Document uidoc)
{
}
```

```

Element selectedElement = null;

foreach (ElementId id in uidoc.Selection.GetElementIds())
{
    selectedElement = uidoc.Document.GetElement(id);

    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.

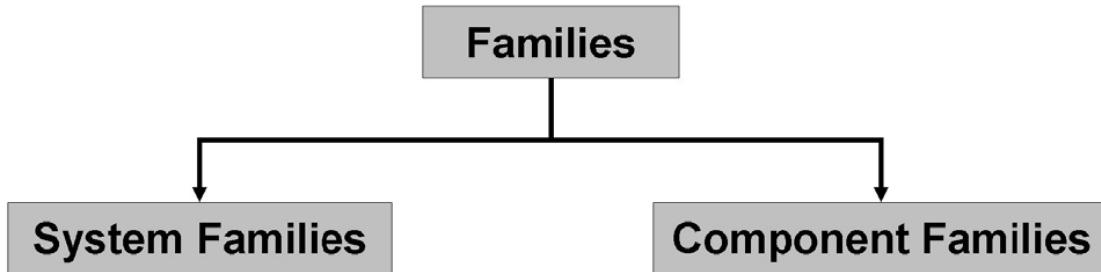
Family

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 22: Families**

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

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: 16×32.

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 16×32 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 [Family Instances](#).

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 × 2032 door, or an 8×4×1/2 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 32×78 six-panel doors belong to one type, while all 24×80 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.

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.

1.4.3 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.

Getting an Element by ID

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

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 [Filtering](#) for more information.

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.GetElementIds` method. For more information on using the active selection, see [Selection](#).

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.

Table 11: Retrieving Elements from document properties

Element	Access in property of Document
ProjectInfo	<code>Document.ProjectInformation</code>
ProjectLocation	<code>Document.ProjectLocations</code> <code>Document.ActiveProjectLocation</code>
SiteLocation	<code>Document.SiteLocation</code>
Phase	<code>Document.Phases</code>

1.4.4 General Properties

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

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.

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

```
private void SetElementId(Element element)
{
    // Get the id of the element
    Autodesk.Revit.DB.ElementId selectedId = element.Id;
    long idLong = selectedId.Value;

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

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.GetElement(ElementId) method.

Code Region 5-4: Using ElementId

```
public void UsingElementId(Element element)
{
    // Get the id of the element
    Autodesk.Revit.DB.ElementId selectedId = element.Id;
```

```

int idLong = selectedId.Value;

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

// Get the element
Autodesk.Revit.DB.Element first = element.Document.GetElement(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.

Uniqueld

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

Code Region 5-5: Uniqueld

```
string uniqueId = element.UniqueId;
```

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

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.

Table 12: Elements Location Information

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

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 ElementTransformUtils.MoveElement() method to change the LocationPoint. This changes both the Group location and the LocationPoint.

For more information about LocationCurve and LocationPoint, see [Moving Elements](#).

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

```
public void AssignLevel(Element element)
{
    // Get the level object to which the element is assigned.

    if (element.LevelId.Equals(ElementId.InvalidElementId))
    {
        TaskDialog.Show("Revit", "The element isn't based on a level.");
    }
    else
    {
        Level level = element.Document.GetElement(element.LevelId) as Level;

        // Format the prompt information(Name and elevation)
        String prompt = "The element is based on a level.";
        prompt += "\nThe level name is: " + level.Name;
        prompt += "\nThe level elevation is: " + level.Elevation;

        // Show the information to the user.

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

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.

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, retrieve all levels using a Level class filter.

For more Level details, see [Datum and Information Elements](#).

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.

Instance Parameters - Control selected or to-be-created instance	
Parameter	Value
Design Return Airflow	0.00 L/s
Actual Return Airflow	0.00 L/s
Design Exhaust Airflow	0.00 L/s
Actual Exhaust Airflow	0.00 L/s
Dimensions	
Area	56.240 m ²
Perimeter	30000.0
Height	2438.4
Volume	Not Computed
Identity Data	
Number	1
Name	Room
Comments	

Figure 25: 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 [Parameters](#).

2 Basic Interaction with Revit Elements

2.1 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

public void GetAllWalls(Document document)
{
    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).WhereElementIsNotEle
mentType().ToElements();

    String prompt = "The walls in the current document are:\n";
    foreach (Element e in walls)
    {
```

```

        prompt += e.Name + "\n";
    }

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

```

2.1.1 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 an attempt to access the elements, otherwise an exception will be thrown.

2.1.2 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.

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.

Table 13: Built-in Quick 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	OfClass()

	class (or derived classes)	
ElementDesignOptionFilter	Elements in a particular design option	ContainedInDesignOption()
ElementIdSetFilter	Elements whose ElementIds are included in a collection	
ElementIsCurveDrivenFilter	Elements which are curve driven	WhereElementIsCurveDriven()
ElementIsElementTypeFilter	Elements which are "Element types"	WhereElementIsElementType() WhereElementIsNotElementType()
ElementMulticategoryFilter	Elements matching any of a given set of categories	None
ElementMulticlassFilter	Elements matching a given set of classes (or derived classes)	None
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	

VisibleInViewFilter	Elements that are most likely visible in the given view
---------------------	---

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

```
public void IntersectsFilterSample(Document document)
{
    // 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

```

public void FindNotSelectedWalls(UIDocument uiDocument)
{
    // 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> selectedIds = uiDocument.Selection.GetElementIds();

    // Use the selection to instantiate an exclusion filter
    ExclusionFilter filter = new ExclusionFilter(selectedIds);

    // For the sake of simplicity we do not test here whether the selection is empty or not,
    // but in production code a proper validation would have to be done to avoid an argument
}

```

```

// exception from the filter's constructor.

// Apply the filter to the elements in the active document,
// Use shortcut method OfClass() to find Walls only

FilteredElementCollector collector = new FilteredElementCollector(uiD
document.Document);

IList<Element> walls = collector.WherePasses(filter).OfClass(typeof(W
all)).ToElements();

}

```

Note: 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

```

public void MakeLoadFilter(Document document)

{
    // Use ElementClassFilter to find all loads in the document

    // Using typeof(LoadBase) will yield all AreaLoad, LineLoad and Point
    Load

    ElementClassFilter filter = new ElementClassFilter(typeof(LoadBase));

    // Apply the filter to the elements in the active document

    FilteredElementCollector collector = new FilteredElementCollector(doc
ument);

    ICollection<Element> allLoads = collector.WherePasses(filter).ToEleme
nts();

}

```

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: 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
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

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.

Table 14: Built-in Slow 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
ElementPhaseStatusFilter	Elements with a given phase status on a given phase	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
StructuralInstanceUsageFilter	FamilyInstances of given structural usage	None
StructuralMaterialTypeFilter	FamilyInstances of given structural material type	None
StructuralWallUsageFilter	Walls of given structural wall usage	None
<u>Element Intersection Filters</u>	Elements that intersect the solid geometry of a given element	None
<u>Element Intersection Filters></u>	Elements that intersect the given solid geometry	None

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

```
public void MakeRoomFilter(Document document)
{
    // 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(doc
ument);
    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

```
public void FindRooms(Document document)
{
    // Creates an ElementParameter filter to find rooms whose area is gre
    ater than specified value

    // Create filter by provider and evaluator
    BuiltInParameter areaParam = BuiltInParameter.ROOM_AREA;
```

```
// provider

    ParameterValueProvider pvp = new ParameterValueProvider(new ElementId
((long)areaParam));

        // Create an ElementParameter filter to filter rooms whose area is gr
eater than 100 SF

            var ruleGreater100 = new FilterDoubleRule(pvp, new FilterNumericGreat
er(), 100, 1E-6);

                ElementParameterFilter filterGreater100 = new ElementParameterFilter
(ruleGreater100);

                    // Apply the filter to the elements in the active document

IList<Element> roomsGreater100 = new FilteredElementCollector(documen
t)

    .OfCategory(BuiltInCategory.OST_Rooms)

    .WherePasses(filterGreater100).ToElements();

                    // Find rooms whose area is less than or equal to 100:

                    // Use inverted filter to match elements

                        ElementParameterFilter filterLessthan100 = new ElementParameterFilter
(ruleGreater100, true);

                            IList<Element> roomsLessthan100 = new FilteredElementCollector(docume
nt)

    .OfCategory(BuiltInCategory.OST_Rooms)

    .WherePasses(filterLessthan100).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

```

public void FindWoodFamilies(Document document)
{
    // 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> woodFamilies = 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();
}

```

The last two slow filters derive from `ElementIntersectsFilter` which is a base class for filters used to match elements which intersect with geometry. See Code Region: Find Nearby Walls in the section [Geometry Utility Classes](#) for an example of the use of this type of filter.

Logical filters

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

Table 15: 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

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

```
public void FindDoors(Document document)
{
    // 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
    ();

}

```

2.1.3 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.GetElement() 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

```
public void ToElementsSample(Document document)
{
    // Use ElementClassFilter to find all loads in the document
    // Using typeof(LoadBase) will yield all AreaLoad, LineLoad and Point
    Load
    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

```
public void GetFirstElement(Document document)
{
    // 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 the `ReferenceIntersector` constructors).

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

```
public void UseExtensionMethods(Document document)
{
    // Use filter to find a non-template 3D view
    // This example does not use FirstElement() since first filtered 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

```
public void GetElementIds(Document document)
{
    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.IsValidForElementIteration(document,
viewId))

    {

        FilteredElementCollector viewCollector = new FilteredElementC
ollector(document, viewId);

        // Get all FamilyInstance items in the view

        viewCollector.OfClass(typeof(FamilyInstance));

        ICollection<ElementId> familyInstanceIds = viewCollector.ToEl
ementIds();

        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

```
public void GetIterator(Document document)

{

    FilteredElementCollector collector = new FilteredElementCollector(doc
ument);

    // 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

```
public void GetIdIterator(Document document)

{

    // 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(doc
ument);
```

```
    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.GetElement(roomId) as Room;
        if (room.Area < 50.0)
        {
            String prompt = "Room is too small: id = " + roomId.ToString();
            TaskDialog.Show("Revit", prompt);
            break;
        }
    }
}
```

In some cases, it may be useful to test a single element against a given filter, rather than getting all elements that pass the filter. There are two overloads for `ElementFilter.PassesFilter()` that test a given `Element`, or `ElementId`, against the filter, returning true if the element passes the filter.

2.1.4 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

```
public void LinqSample(Document document)
{
    // 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>();
```

{}

2.1.5 Bounding Box filters

The BoundingBox filters:

- `BoundingBoxIsInsideFilter`
- `BoundingBoxIntersectsFilter`
- `BoundingBoxContainsPointFilter`

help you find elements whose bounding boxes meet a certain criteria. You can check if each element's bounding box is inside a given volume, intersects a given volume, or contains a given point. You can also reverse this check to find elements which do not intersect a volume or contain a given point.

BoundingBox filters use `Outline` as their inputs. `Outline` is a class representing a right rectangular prism whose axes are aligned to the Revit world coordinate system.

These filters work best for shapes whose actual geometry matches closely the geometry of its bounding box. Examples might include linear walls whose curve aligns with the X or Y direction, rectangular rooms formed by such walls, floors or roofs aligned to such walls, or reasonably rectangular families. Otherwise, there is the potential for false positives as the bounding box of the element may be much bigger than the actual geometry. (In these cases, you can use the actual element's geometry to determine if the element really meets the criteria).

2.1.6 Element Intersection Filters

The element filters:

- `ElementIntersectsElementFilter`
- `ElementIntersectsSolidFilter`

pass elements whose actual 3D geometry intersects the 3D geometry of the target object.

With `ElementIntersectsElementFilter`, the target object is another element. The intersection is determined with the same logic used by Revit to determine if an interference exists during generation of an Interference Report. (This means that some combinations of elements will never pass this filter, such as concrete members which are automatically joined at their intersections, or site elements which are also excluded from interference checks). Also, elements which have no solid geometry, such as Rebar, will not pass this filter.

With `ElementIntersectsSolidFilter`, the target object is any solid. This solid could have been obtained from an existing element, created from scratch using the routines in `GeometryCreationUtilities`, or the result of a secondary operation such as a Boolean operation.

Similar to the ElementIntersectsElementFilter, this filter will not pass elements which lack solid geometry.

Both filters can be inverted to match elements outside the target object volume.

Both filters are slow filters, and thus are best combined with one or more quick filters such as class or category filters.

Code region: using ElementIntersectsSolidFilter to match elements which block disabled egress to doors

```
/// <summary>
/// Finds any Revit physical elements which interfere with the target
/// solid region surrounding a door.</summary>
/// <remarks>This routine is useful for detecting interferences which are
/// violations of the Americans with Disabilities Act or other local disabled
/// access codes.</remarks>
/// <param name="doorInstance">The door instance.</param>
/// <param name="doorAccessibilityRegion">The accessibility region calculated
/// to surround the approach of the door.
/// Because the geometric parameters of this region are code- and
/// door-specific, calculation of the geometry of the region is not
/// demonstrated in this example.</param>
/// <returns>A collection of interfering element ids.</returns>
private ICollection<ElementId> FindElementsInterferingWithDoor(FamilyInstance
doorInstance, Solid doorAccessibilityRegion)
{
    // Setup the filtered element collector for all document elements.
    FilteredElementCollector interferingCollector =
        new FilteredElementCollector(doorInstance.Document);
```

```
// Only accept element instances  
  
interferingCollector.WhereElementIsNotElementType();  
  
  
// Exclude intersections with the door itself or the host wall for the doo  
r.  
  
List<ElementId> excludedElements = new List<ElementId>();  
  
excludedElements.Add(doorInstance.Id);  
  
excludedElements.Add(doorInstance.Host.Id);  
  
ExclusionFilter exclusionFilter = new ExclusionFilter(excludedElements);  
  
interferingCollector.WherePasses(exclusionFilter);  
  
  
// Set up a filter which matches elements whose solid geometry intersects  
// with the accessibility region  
  
ElementIntersectsSolidFilter intersectionFilter =  
    new ElementIntersectsSolidFilter(doorAccessibilityRegion);  
  
interferingCollector.WherePasses(intersectionFilter);  
  
  
// Return all elements passing the collector  
  
return interferingCollector.ToElementIds();  
}
```

2.2 Selection

You can get the selected objects from the current active document using the `UIDocument.Selection.GetElementIds()` method which returns a collection of `ElementIds` of the selected elements. The collection returned by this method can be used directly with `FilteredElementCollector` to filter the selected elements.

The `Selection` object can also be used to change the current selection programmatically using the `SetElementIds()` method.

PickPoint

The `Selection.PickPoint()` method prompts the user to pick a point on the active work plane. By specifying one or more value from the `ObjectSnapTypes` enum, you can determine which objects the user's cursor will snap to during the pick. These values are

- Centers
- CoordinationModelPoints
- Endpoints
- Intersections
- Midpoints
- Nearest
- None
- Perpendicular
- Points
- Quadrants
- Tangents
- WorkPlaneGrid

2.2.1 Changing the Selection

To modify the selected elements:

1. Create a new List of `ElementIds`.
2. Put `ElementIds` in it.
3. Call `SetElementIds()` with the new list.

The following example illustrates how to change the selected Elements by getting the current selection and filtering out just walls to set as the new selection.

It is also possible to select references by using `SetReferences()`. The references can be an element or a sub element in the host or a linked document. `GetReferences()` returns the references that are currently selected.

Code Region 7-1: Changing selected elements

```
private void ChangeSelection(UIDocument uidoc)
{
    // Get selected elements from current document.
```

```
ICollection<ElementId> selectedIds = uidoc.Selection.GetElementIds();  
  
// Display current number of selected elements  
  
TaskDialog.Show("Revit", "Number of selected elements: " + selectedIds.Count.ToString());  
  
// Go through the selected items and filter out walls only.  
  
ICollection<ElementId> selectedWallIds = new List<ElementId>();  
  
foreach (ElementId id in selectedIds)  
{  
    Element elements = uidoc.Document.GetElement(id);  
    if (elements is Wall)  
    {  
        selectedWallIds.Add(id);  
    }  
}  
  
// Set the created element set as current select element set.  
  
uidoc.Selection.SetElementIds(selectedWallIds);  
  
// Give the user some information.  
  
if (0 != selectedWallIds.Count)  
{  
    TaskDialog.Show("Revit", selectedWallIds.Count.ToString() + " Walls are selected!");  
}
```

```

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

```

2.2.2 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 PickBox() method invokes a general purpose two-click editor that lets the user to specify a rectangular area on the screen.

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()

```

public void SelectElements(Document document)
{
    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 selected.");
}

// Use the rectangle picking tool to identify model elements to select.

IList<Element> pickedElements = uidoc.Selection.PickElementsByRectangle("Select by rectangle");

if (pickedElements.Count > 0)
{
    // Collect Ids of all picked elements
    IList<ElementId> idsToSelect = new List<ElementId>(pickedElements.Count);

    foreach (Element element in pickedElements)
    {
        idsToSelect.Add(element.Id);
    }

    // Update the current selection
    uidoc.Selection.SetElementIds(idsToSelect);

    TaskDialog.Show("Revit", string.Format("{0} elements added to Selection.", idsToSelect.Count));
}
}
```

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);
}
```

The PickBox() method takes a PickBoxStyle enumerator. The options are Crossing, the style used when selecting objects completely or partially inside the box, Enclosing, the style used selecting objects that are completely enclosed by the box, and Directional, in which the style of the box depends on the direction in which the box is being drawn. It uses the Crossing style if it is being drawn from right to left, or the Enclosing style when drawn in the opposite direction.

PickBox() returns a PickedBox which contains the Min and Max points selected. The following example demonstrates the use of PickBox() in Point Cloud selection.

Code Region: PickBox

```
public void PromptForPointCloudSelection(UIDocument uiDoc, PointCloudInstance pcInstance)
{
    Autodesk.Revit.ApplicationServices.Application app = uiDoc.Application.Application;
    Selection currentSel = uiDoc.Selection;
```

```
PickedBox pickedBox = currentSel.PickBox(PickBoxStyle.Enclosing, "Select  
region of cloud for highlighting");  
  
XYZ min = pickedBox.Min;  
XYZ max = pickedBox.Max;  
  
//Transform points into filter  
  
View view = uiDoc.ActiveView;  
XYZ right = view.RightDirection;  
XYZ up = view.UpDirection;  
  
List<Plane> planes = new List<Plane>();  
  
// X boundaries  
bool directionCorrect = IsPointAbovePlane(right, min, max);  
planes.Add(Plane.CreateByNormalAndOrigin(right, directionCorrect ? min :  
max));  
planes.Add(Plane.CreateByNormalAndOrigin(-right, directionCorrect ? max :  
min));  
  
// Y boundaries  
directionCorrect = IsPointAbovePlane(up, min, max);  
planes.Add(Plane.CreateByNormalAndOrigin(up, directionCorrect ? min : ma  
x));  
planes.Add(Plane.CreateByNormalAndOrigin(-up, directionCorrect ? max : mi  
n));  
  
// Create filter
```

```

PointCloudFilter filter = PointCloudFilterFactory.CreateMultiPlaneFilter
(planes);

Transaction t = new Transaction(uiDoc.Document, "Highlight");

t.Start();

pcInstance.SetSelectionFilter(filter);

pcInstance.FilterAction = SelectionFilterAction.Highlight;

t.Commit();

uiDoc.RefreshActiveView();

}

```

Selection Events

The `SelectionChanged` event notifies your code after the selection changes. `SelectionChangedEventArgs` provides access to the references and element ids that are selected.

2.2.3 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(selFilte
r,
    "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(document);

    IList<Reference> faces = uidoc.Selection.PickObjects(ObjectType.Face,
        selFilter, "Select multiple planar faces");

}

public class PlanarFacesSelectionFilter : ISelectionFilter

{
    Document doc = null;

    public PlanarFacesSelectionFilter(Document document)
    {
        doc = document;
    }

    public bool AllowElement(Element element)
    {
        return true;
    }

    public bool AllowReference(Reference refer, XYZ point)
    {
        if (doc.GetElement(refer).GetGeometryObjectFromReference(refer) is PlanarFace)
```

```

    {
        // Only return true for planar faces. Non-planar faces
        // will not be selectable

        return true;
    }

    return false;
}

}

```

For more information about retrieving Elements from selected Elements, see [Walkthrough: Retrieve Selected Elements](#) in the [Getting Started](#) section.

2.3 Parameters

Revit provides a general mechanism for giving each element a set of parameters that you can edit.

In the Revit UI, some element parameters are visible in the Element Properties Palette. The following sections describe how to get and use built-in parameters, shared parameters and global parameters.

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

Common ways to get the value of a parameter are shown below. In this sample, all three lines of code get the same parameter. Because this parameter is stored as a string, the `AsString()` method is used to get its value.

```

private void GetStringValue(Wall wall)
{
    string s1 = wall.LookupParameter("Comments").AsString();

    string s2 = wall.GetParameter(ParameterTypeId.AllModelInstanceComments).AsString();

    string s3 = wall.get_Parameter(BuiltInParameter.ALL_MODEL_INSTANCE_COMMENTS).
        AsString();
}

```

Other ways to get a parameter are:

- 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](#)).
- By iterating through the collection returned by Element.GetOrderedParameters(), which returns only parameters visible in the Properties Palette.
- By accessing a parameter by name via the Element.ParametersMap collection or Element.GetParameters()

You can retrieve the Parameter object from an Element using the overloaded Parameter property if you know the built-in ID, definition, or GUID. 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.

The Element.LookupParameter() method 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. Also, keep in mind that multiple matches of parameters with the same name can occur because shared parameters or project parameters can be bound to an element category even if there is already a built-in parameter with the same name. For this reason, it is better to use Element.GetParameters() which will return all parameters matching the given name. Element.LookupParameter() will return the first match found.

Parameters Service

Parameters Service is a semi-automated, fully searchable database of design element parameters. Because Parameters Service lives in the Autodesk Construction Cloud, you can easily share it with teammates regardless of their physical location.

The class `Autodesk.Revit.DB.ParameterDownloadOptions` is an option class used for downloading parameters from the Parameters Service with the following properties:

- `ParameterDownloadOptions.Categories` - Categories for binding.
- `ParameterDownloadOptions.IsInstance` - Returns true if binding to element instances, false if binding to element types.
- `ParameterDownloadOptions.Visible` - Returns true if the parameter is visible to the user, false if it is hidden and accessible only via the API.
- `ParameterDownloadOptions.GroupTypeId` - Properties palette group identifier.

These `ParameterUtils` methods provide functionality relevant to the Parameters Service:

- `ParameterUtils.DownloadParameter()` - Allows users to create a shared parameter element in the given document according to a parameter definition downloaded from the Parameters Service.
- `ParameterUtils.DownloadParameterOptions()` - Allows users to retrieve the requested parameter's category, visibility and group bindings from the Forge Schema Service.
- `ParameterUtils.DownloadCompanyName()` - Allows users to download and record the name of the given parameter schema identifier's owning account in the given document.

2.3.1 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 in subsequent topics.

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: \n\r";

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

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

String GetParameterInformation(Parameter para, Document document)
{
    string defName = para.Definition.Name + "\t : ";
    string defValue = string.Empty;
```

```
// Use different method to get parameter data according to the storage type
pe

switch (para.StorageType)
{
    case StorageType.Double:
        //covert the number into Metric
        defValue = para.AsValueString();
        break;

    case StorageType.ElementId:
        //find out the name of the element
        Autodesk.Revit.DB.ElementId id = para.AsElementId();

        if (id.Value >= 0)
        {
            defValue = document.GetElement(id).Name;
        }
        else
        {
            defValue = id.Value.ToString();
        }
        break;

    case StorageType.Integer:
        if (SpecTypeId.Boolean.YesNo == para.Definition.GetDataType())
        {
            if (para.AsInteger() == 0)
            {
                defValue = "False";
            }
        }
}
```

```
        else
        {
            defValue = "True";
        }
    }
    else
    {
        defValue = para.AsInteger().ToString();
    }
    break;
case StorageType.String:
    defValue = paraAsString();
    break;
default:
    defValue = "Unexposed parameter.";
    break;
}

return defName + defValue;
}
```

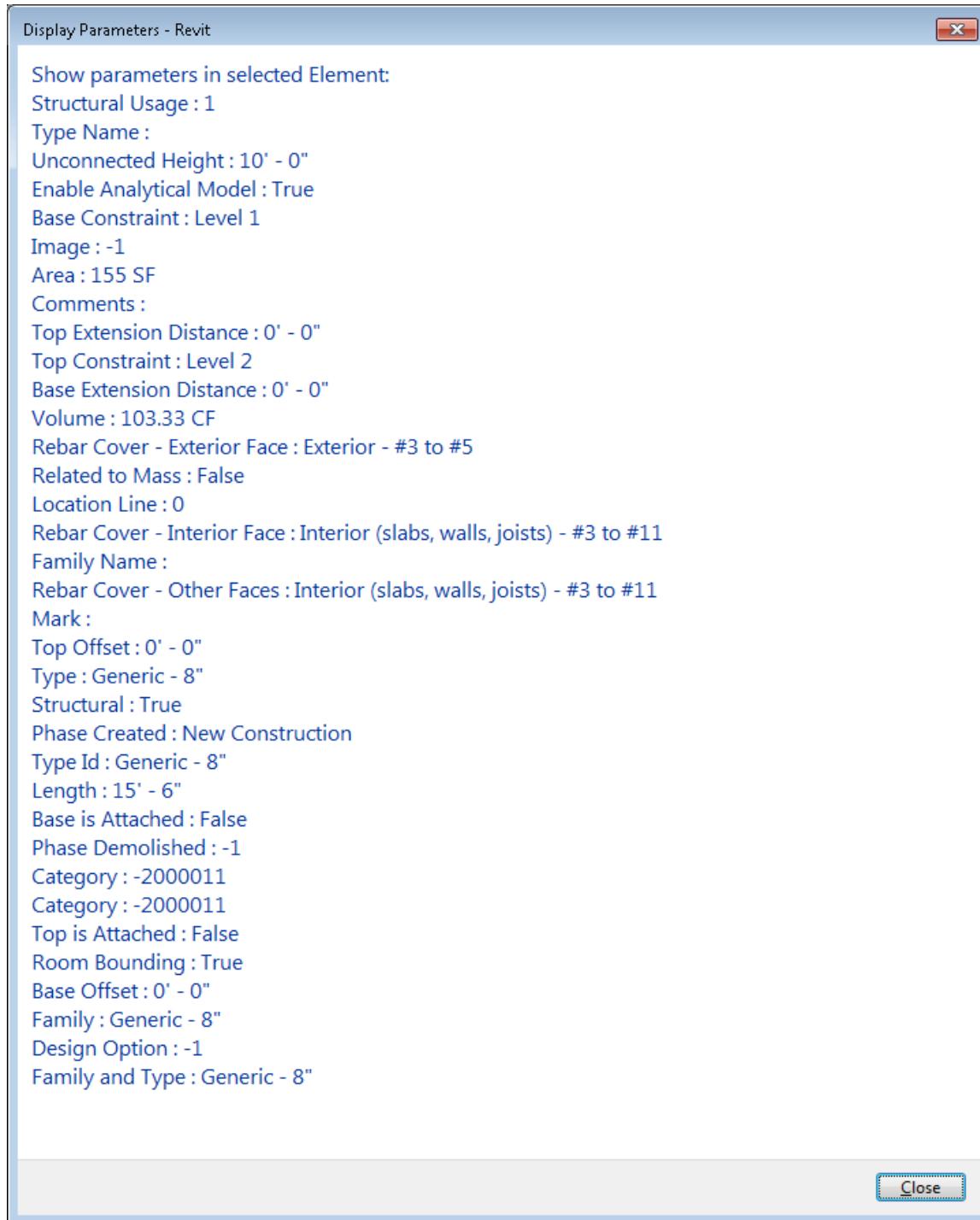


Figure 26: Get wall parameters result

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.

2.3.2 Parameter

The Parameter class contains the value for the given parameter.

All Elements within Autodesk Revit contain Parameters, which can be retrieved as a set or individually. An individual parameter object can be obtained from any Element using either a `BuiltInParameter` enumeration, a `Definition` object or a Shared Parameter GUID.

The data contained within the parameter can be either a Double, Integer, String or `ElementId`, as indicated by its `StorageType` property. For value types, the `DisplayUnitType` property will indicate the display unit used for the parameter value. The Parameter object also contains a [Definition](#) object that describes the data type, name and other details of the parameter.

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	Description
<code>Name</code>	
<code>_String_</code>	Internal data is stored as a string of characters.
<code>_ElementId_</code>	Data type represents an element and is stored as an Element ID.
<code>_Double_</code>	Data is stored internally as an 8-byte floating point number.
<code>_Integer_</code>	Internal data is stored as a signed 32-bit integer.
<code>_None_</code>	None represents an invalid storage type. For internal use only.

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.

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: 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 storageType = parameter.StorageType;
        if (StorageType.Double != storageType)
        {
            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.

AsValueString() and SetValueString()

These two Parameter class methods only apply 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. Using the `AsValueString()` method, you get the `-20'0"` string value directly. Using the `AsDouble()` method, you get a double value like `-20` without the units of measure.

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: 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;
}
```

Global parameter associations

The `Parameter` class has several methods for maintaining associations between element parameters and global parameters. The method `GetAssociatedGlobalParameter()` returns the `ElementId` of a global parameter, if any, currently associated with a parameter. `InvalidElementId` is returned in case this parameter is not associated with any global parameter. `InvalidElementId` is also returned if called for a parameter that cannot even be associated with a global parameters (i.e. a non-parameterizable parameter or a parameter with a formula).

There are two methods to determine if a parameter can be associated with global parameters. `Parameter.CanBeAssociatedWithGlobalParameters()` tests whether a parameter can be associated with any global parameter. Only properties defined as parametrizable can be associated with global parameters. That excludes any read-only and formula-driven parameters, as well as those that have other explicit or implicit restrictions imposed by Revit. To test whether a specific global parameter can be associated with this parameter, use `Parameter.CanBeAssociatedWithGlobalParameter()`. Keep in mind that the parameter's value type must match the type of the global parameter to create an association.

For parameters that can be associated with a global parameter, use `AssociateWithGlobalParameter()` to create the association. Once associated, the parameter can be later dissociated by calling the `DissociateFromGlobalParameter()` method

ParameterUtils

`ParameterUtils` is a class with static utility functions related to `ForgeTypeld` parameter identifiers:

- `ParameterUtils.GetAllBuiltInGroups()`
- `ParameterUtils.GetAllBuiltInParameters()`
- `ParameterUtils.IsBuiltInGroup(ForgeTypeld)`
- `ParameterUtils.IsBuiltInParameter(ForgeTypeld)`

Multiple Values

The `MultipleValuesIndicationSettings` class allows access to the custom value used in instances of the Properties dialog, Tags, and Schedules when multiple elements are referenced and the value of the parameter is differs among those elements

2.3.3 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.GetDataType() == SpecTypeId.Length)
    {
        foundParameter = parameter;
        break;
    }
}

return foundParameter;
}

```

Parameter Data Type

For parameters associated with units of measurement, `GetDataType()` returns a measurable spec identifier. Use `UnitUtils.IsMeasurableSpec(ForgeTypeId)` to detect a measurable spec identifier.

For Family Type parameters, `GetDataType()` returns a category identifier. Use `Category.IsBuiltInCategory(ForgeTypeId)` to detect a category identifier.

For all other parameters, `GetDataType()` returns a spec identifier. Use `Parameter.IsSpec(ForgeTypeId)` to detect a spec identifier, including measurable specs.

GetGroupId()

The Definition class `GetGroupId()` method returns a ForgeTypeid. The members of the `GroupTypeid` class list all value supported by Revit. The `GroupTypeid` is used to sort parameters in the Element Properties dialog box.

InternalDefinition

Every Parameter object has an `InternalDefinition`, which can be obtained from the `Definition` property. The `InternalDefinition` represents the parameter definition in the Revit document. In addition to the properties it inherits from `Definition`, it also has some other key properties.

BuiltInParameter

This property tests whether this definition identifies a built-in parameter or not. For a built-in parameter, this property returns one of the `BuiltInParameter` enumerated values. For custom-defined parameters, such as shared, global, or family parameters the value will be `BuiltInParameter.INVALID`.

Id

This property returns the id for the associated ParameterElement if the parameter is not built-in.

VariesAcrossGroups

This property, and the corresponding SetAllowVaryBetweenGroups() method, determine whether the values of this parameter can vary across the related members of group instances. If False, the values will be consistent across the related members in group instances. This can only be set for non-built-in parameters.

Visible

The visible property indicates whether a shared parameter is hidden from the user. This is useful if you wish to add data to an element that is only meaningful to your application and not to the user. This value can only be set when the shared parameter definition is created.

2.3.4 Built-In Parameters

The Revit Platform API has a large number of built-in parameters.

Built-in parameters are defined in the Autodesk.Revit.Parameters.BuiltInParameter enumeration (see the RevitAPI Help.chm file for the definition of this enumeration). This enumeration has generated documentation visible from Visual Studio intellisense as shown below. The documentation for each id includes the parameter name, as found in the Element Properties dialog in the English version of Autodesk Revit. Note that multiple distinct parameter ids may map to the same English name; in those cases you must examine the parameters associated with a specific element to determine which parameter id to use.

```
BuiltInParameter.WALL_BASE_OFFSET;
:t_Parameter(para1
    BuiltInParameter BuiltInParameter.WALL_BASE_OFFSET
    "Base Offset"
```

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 parameterId
    // to get the base offset parameter of the wall.

    BuiltInParameter paraIndex = BuiltInParameter.WALL_BASE_OFFSET;

    Parameter parameter = wall.get_Parameter(paraIndex);

    return parameter;
}
```

Getting localized parameter names

The method `LabelUtils.GetLabelFor Method (BuiltInParameter, LanguageType)` returns the localized string name for the built-in parameter. If the corresponding resource DLL cannot be found, the US-English name will be returned.

2.3.5 Parameter Relationships

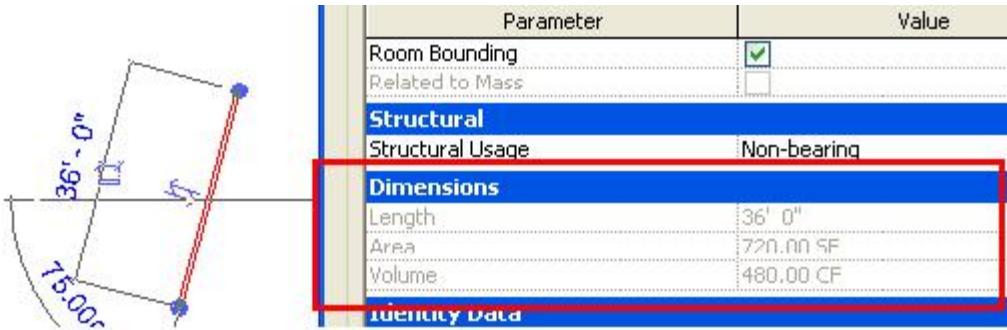
Parameters can be affected by each other.

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 29: 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: 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.GetParameter(ParameterTypeId.InstanceSillHeightParam);

    Parameter headPara = opening.GetParameter(ParameterTypeId.InstanceHeadHeightParam);

    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();

    TaskDialog.Show("Info", "Old head height: " + origHeadHeight + "; new head height: ");
}
```

```
+ newHeadHeight);  
}
```

Global parameters also have relationships with other parameters. See the [GlobalParameter Basics](#) topic for more information.

2.3.6 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.

The main objects associated with shared parameters are:

- DefinitionFile - represents a shared parameters file on disk
- DefinitionGroup - group of shared parameters which are organized into meaningful sets
- ExternalDefinition - represents a shared parameter definition, belongs to a DefinitionGroup
- ExternalDefinitions - supports the creation of new shared parameters definitions
- Binding - binds a parameter definition to one or more categories
- BindingMap - contains all the parameter bindings that exist in the Autodesk Revit project
- ParameterElement - stores information about a particular user-defined parameter in the document
- SharedParameterElement - derived from ParameterElement, stores the definition of a shared parameter. The following sections describe how to gain access to shared parameter definitions through the Revit Platform API, including how to get a shared parameter definition and bind it to Elements in certain Categories.

To access Shared Parameters after they are defined and bound to categories, see [Parameters](#).

Clearing the value of a parameter

`Parameter.ClearValue()` resets the value of a shared parameter with the `HideWhenNoValue` flag to a cleared state. This method is not applicable to clear the value of any other parameter type.

Hiding empty parameters

The property `ExternalDefinition.HideWhenNoValue` indicates if the shared parameter should be hidden from the property palette and `Element.GetOrderedParameters()` when it has no value.

This can be set when creating a shared parameter by setting `ExternalDefinitionCreationOptions.HideWhenNoValue`. There is also `SharedParameterElement.ShouldHideWhenNoValue()`

Filtering by presence or absence of parameter value

These filter classes define filter rules evaluating whether or not a parameter has a value for a specific element:

- `ParameterValuePresenceRule`
- `HasValueFilterRule`
- `HasNoValueFilterRule`

These static methods create an instance of these rules for use in a parameter filter.:

- `FilterRule.CreateHasValueParameterRule()`
- `FilterRule.CreateHasNoValueParameterRule()`

In schedules, use these new enumerated values with the method `ScheduleDefinition.CanFilterByValuePresence()` to filter based on the presence or absence of a value assigned to that parameter:

- `ScheduleFilterType.HasValue`
- `ScheduleFilterType.HasNoValue`

2.3.6.1 Definition File

The `DefinitionFile` object represents a shared parameter file which is a common text file.

Format

The shared parameter definition file is a text file (.txt) with three blocks: META, GROUP and PARAM. The GROUP and PARAM blocks are relevant to the shared parameter functionality in the Revit API. Do not edit the definition file directly; instead, edit it using the UI or the API.

Although the Revit API takes care of reading and writing this file, the following section provides information the format of the file, which closely corresponds to the API objects and methods used to access shared parameters. The file uses tabs to separate fields and can be difficult to read in a text editor. The code region below shows the contents of a sample shared parameter text file.

Code Region 22-1: Parameter definition file example

```
# This is a Revit shared parameter file.  
# Do not edit manually.
```

```

*META      VERSION      MINVERSION

META      2      1

*GROUP      ID      NAME

GROUP      1      MyGroup

GROUP      2      AnotherGroup

*PARAM      GUID      NAME      DATATYPE      DATACATEGORY      GROUP      VISIBLE      DES
CRIPTION      USERMODIFIABLE

PARAM      bb7f0005-9692-4b76-8fa3-30cec8aecf74      Price      INTEGER      2
1      Enter price in USD      1

PARAM      b7ea2654-b206-4694-a087-756359b52e7f      areaTags      FAMILYTYPE      -2
005020      1      1      1

PARAM      d1a5439d-dc8d-4053-99fa-2f33804bae0e      MyParam      TEXT      1
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.
- 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.
 - DATACATEGORY - An optional field for parameters whose DATATYPE is FAMILYTYPE.
 - 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
 - DESCRIPTION - An optional field for a tooltip for this parameter.
 - USERMODIFIABLE - Identifies whether the parameter is editable by the user. 0 = user cannot edit the parameter and it is greyed out in the UI
1 = user can edit the parameter value in the UI

In the sample definition file, there are two groups:

- MyGroup - ID 1 - Contains the parameter definition for MyParam which is a Text type parameter, and the definition for areaTags which is a FamilyType parameter.
- AnotherGroup - ID 2 - Contains the parameter definition for Price which is an Integer type parameter.

Of the 3 parameters in the sample file, only Price has a description. All of the parameters are visible and user modifiable.

2.3.6.2 Working with the Definition File

The definition file provides access to shared parameters.

Accessing parameters in the definition file

Use the following steps to gain access to the definition file and its parameters:

1. Specify the 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 classes and methods in the Autodesk.Revit.DB namespace provide access to shared parameters using the Revit API.

- DefinitionFile Class
 - Retrieved using the Application.OpenSharedParameterFile() method. Revit uses one shared parameter file at a time.
 - Represents one shared parameter file.
 - Contains a number of Group objects.
 - Shared parameters are grouped for easy management and contain shared parameter definitions.
 - New definitions can be added as needed.
- ExternalDefinition 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.
- Application.SharedParametersFilename Property

- Get and set the shared parameter file path using this 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.

Create a Shared Parameter File

Because the definition file is a text file, it can be created manually or using code.

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();
}
```

Access an Existing Shared Parameter File

Since Revit can have many shared parameter files, 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.

Get DefinitionFile from an External Parameter File

Set the shared parameter file path as the following code illustrates, then invoke the 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.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 Application.SharedParametersFilename is set to an invalid 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.

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 definitions
foreach (Definition definition in myGroup.Definitions)
{
    // get definition name
    fileInformation.AppendLine("Definition Name: " + definition.Name);
}
}

TaskDialog.Show("Revit",fileInformation.ToString());
}
```

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 (myGroup != null)
    {
        ExternalDefinition myExtDef = myGroup.Definitions.get_Item("MyParam")
        as ExternalDefinition;
        if (myExtDef != null)
        {
            myExtDef.OwnerGroup = myGroup;
        }
    }
}
```

```
DefinitionGroup newGroup = myGroups.get_Item("AnotherGroup");

if (newGroup != null)

{

    // change the OwnerGroup of the ExternalDefinition

    myExtDef.OwnerGroup = newGroup;

}

}

}
```

2.3.6.3 Binding

Binding is what ties shared parameters to elements in certain categories in the model.

There are two types of binding available, Instance binding and Type binding. The key difference between the two is that the instance bound parameters appear on all instances of the elements in those categories. Changing the parameter on one does not affect the other instances of the parameter. The Type bound parameters appear only on the type object and is shared by all the instances that use that type. Changing the type bound parameter affects all instances of the elements that use that type. Note, a definition can only be bound to an instance or a type and not both.

To bind a 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 `BindingMap` object available from the `Document.ParameterBindings` property.

The following class and method in the Autodesk.Revit.DB namespace provide more information on binding parameters to elements.

- BindingMap Class
 - 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.
 - BindingMap.Insert() 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.

Type Binding

The 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.

The following code sample 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 Revit UI. 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
    ExternalDefinitionCreationOptions option = new ExternalDefinitionCreationOptions("CompanyName", SpecTypeId.String.Text);
    Definition myDefinition_CompanyName = myGroup.Definitions.Create(option);

    // 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 = Category.GetCategory(app.ActiveUIDocument.Document, BuiltInCategory.OST_Walls);

myCategories.Insert(myCategory);

//Create an object of TypeBinding according to the Categories

TypeBinding typeBinding = app.Application.Create.NewTypeBinding(myCategories);

// Get the BindingMap 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;
}

```

Instance Binding

The InstanceBinding object indicates binding between a parameter definition and a parameter in certain category instances.

Once bound, the parameter appears in all property dialog boxes for the instance (if the visible property is set to true). Changing the parameter in any one instance does not change the value in any other instance.

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 SetNewParameterToInstanceWall(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

    ExternalDefinitionCreationOptions option = new ExternalDefinitionCreationOptions("Instance_ProductDate", SpecTypeId.String.Text);

    // Don't let the user modify the value, only the API

    option.UserModifiable = false;

    // Set tooltip

    option.Description = "Wall product date";

    Definition myDefinition_ProductDate = myGroup.Definitions.Create(option);

    // 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 = Category.GetCategory(app.ActiveUIDocument.Document,
        BuiltInCategory.OST_Walls);

    myCategories.Insert(myCategory);

    //Create an instance of InstanceBinding

    InstanceBinding instanceBinding = app.Application.Create.NewInstanceBinding(
        myCategories);

    // Get the BindingMap 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;

}

```

2.3.6.4 SharedParameterElement

SharedParameterElements store information about a particular user-defined shared parameter in the document

User-defined parameters are stored in the document and represented by the ParameterElement class. The subclass SharedParameterElement represents a shared parameter loaded into the document. ParameterElement is also the base class for GlobalParameter.

Once a shared parameter has been loaded into a document, information about it can be retrieved from the SharedParameterElement class. SharedParameterElement inherits the GetDefinition() method from the parent ParameterElement class. GetDefinition() returns the InternalDefinition that represents the definition for the parameter in the document, as opposed to the ExternalDefinition for a shared parameter that is stored in the shared parameter file. SharedParameterElement also provides access to the Guid that identifies the shared parameter via the GuidValue property.

The static Create() method on this class can create a new SharedParameterElement in the document from an ExternalDefinition.

The static Lookup() method can retrieve a SharedParameterElement from a given Guid.

In the following example, a schedule contains a field representing the value of a shared parameter. The definition for the shared parameter is retrieved from the SharedParameterElement.

Code Region: Getting the definition of a shared parameter

```
// Check if a given shared parameter in a schedule can vary across groups
```

```
public bool CanParamVaryAcrossGroups(ViewSchedule schedule, string sharedParamName)

{
    bool variesAcrossGroups = false;

    int numFields = schedule.Definition.GetFieldCount();

    // Find the field with the given name

    for (int i = 0; i < numFields; i++)

    {
        ScheduleField field = schedule.Definition.GetField(i);

        if (field.GetName().Contains(sharedParamName))

        {
            // Get the SharedParameterElement from the field's parameter id

            SharedParameterElement spe = schedule.Document.GetElement(field.ParameterId) as SharedParameterElement;

            if (spe != null)

            {
                InternalDefinition definition = spe.GetDefinition();

                variesAcrossGroups = definition.VariesAcrossGroups;
            }
        }
    }

    return variesAcrossGroups;
}
```

SharedParameterElements are especially useful when working with [RebarContainers](#). A shared parameter can be added as an override to the parameters manager for a RebarContainer. The

shared parameter does not need to be bound to any categories to be added as an override. The following example adds a given shared parameter as an override to a RebarContainer.

Code Region: Using a SharedParameterElement as an override for a RebarContainer

```
// Find the named shared parameter and add it as an override to the parameter
// manger for the given RebarContainer

void AddSharedParameterOverride(RebarContainer container, string sharedParamName)

{
    // find the shared parameter guid

    FilteredElementCollector collector = new FilteredElementCollector(container.Document);

    collector.OfClass(typeof(SharedParameterElement));

    IEnumerable<SharedParameterElement> paramCollector = collector.Cast<SharedParameterElement>();

    foreach (SharedParameterElement spe in paramCollector)
    {
        if (spe.Name.CompareTo(sharedParamName) == 0)
        {
            RebarContainerParameterManager paramManager = container.GetParametersManager();

            paramManager.AddSharedParameterAsOverride(spe.Id);

            break;
        }
    }
}
```

2.3.7 Global Parameters

Global parameters support controlling geometry constraints through special parameters defined in a project document.

Global Parameters can be used for both labeling and reporting to/from dimensions, as well as setting values of instance parameters. They can be used to drive values of dimensions or other elements' parameters and they can be driven by a selected dimension, the value of which then determines the value of the global parameter.

2.3.7.1 *Managing Global Parameters*

The `GlobalParametersManager` class provides access to general information and data of Global Parameter elements in a particular model.

`GlobalParametersManager` provides the main access point to managing global parameters in a project document. It provides static methods to access and reorder global parameters and to test for name uniqueness and Id validity.

Accessing global parameters

Global parameters are only supported in project documents, not family documents. Even with a project document, however, global parameters may be disallowed under certain circumstances, either temporarily or permanently. The `AreGlobalParametersAllowed()` method will indicate whether global parameters are allowed within a specified document.

If global parameters are allowed in a project document, use the methods `GetAllGlobalParameters()` to get all global parameters in a specified document, or `GetGlobalParametersOrdered()` to get an ordered list of the global parameters. When retrieving an ordered list, the order of the items corresponds to the order in which global parameters appear in the standard Global Parameters dialog in the Revit user interface.

To get a global parameter by name, call `FindByName()`, which will return the `ElementId` of the named global parameter, or `ElementId.InvalidElementId` if no global parameter is found with the given name. Since global parameter names must be unique, the `IsUniqueName()` method should be called to check a name prior to creating a new `GlobalParameter`.

Given an `ElementId` for a `GlobalParameter`, the `IsValidGlobalParameter()` will confirm that the given `ElementId` is a valid global parameter id.

The following example demonstrates how to get all global parameters, if global parameters are allowed in the document.

Code Region: Getting global parameters

```
/// <summary>
/// Returns all global parameter elements defined in the given document.
```

```

/// </summary>

/// <param name="document">Revit project document.</param>

/// <returns>A set of ElementIds of global parameter elements</returns>

public ISet<ElementId> GetAllGlobalParameters(Document document)

{
    // Global parameters are not available in all documents.

    // They are available in projects, but not in families.

    if (GlobalParametersManager.AreGlobalParametersAllowed(document))

    {
        return GlobalParametersManager.GetAllGlobalParameters(document);
    }

    // return an empty set if global parameters are not available in the document

    return new HashSet<ElementId>();
}

```

Reordering global parameters

GlobalParametersManager provides methods to change the given order of the global parameters in the project document. These operations have no effect on the global parameters themselves. The rearranged order is only visible in the standard Global Parameters dialog in Revit and reflected in the GetGlobalParametersOrdered() method.

- SortParameters() - sorts the global parameters in either ascending or descending alphabetical order, but only within the range of their respective parameter group.
- MoveParameterDownOrder() - moves a given parameter down in the current order.
- MoveParameterUpOrder() - moves the given parameter up in the current order. A parameter can only be moved within its parameter group, so if a parameter can no longer move because it is at the boundary of its group, the MoveParameter methods will return False.

2.3.7.2 *GlobalParameter Basics*

The GlobalParameter class represents a global parameter in a project document and can be used to create and modify global parameters.

Creating global parameters

Global parameters may be created only in project documents, not in families. Global parameters are created via the static `Create()` method in a given document, with a given name and `SpecTypeId`. Each new parameter must have a name that is unique within the document, which can be determined using the static `GlobalParametersManager.IsUniqueName()` method. Global parameters can be created with almost any type of data, but there are a few types that are not currently supported, such as the `ElementId` type. Test whether a particular data type is appropriate for a global parameter by using the static `SpecUtils.IsSpec()` method.

Parameters are created as non-reporting initially, but can be changed after creation using the `IsReporting` property if the global parameter is of an eligible type. See the [Reporting vs. Non-Reporting Parameters](#) page for more information.

Code Region: Create a new global parameter

```
/// <summary>
/// Creates a new Global Parameter of type Length, assigns it an initial value,
/// and uses it to label a set of input dimension elements.
/// </summary>

/// <param name="document">Revit project document in which to create the parameter.</param>
/// <param name="name">Name of the global parameter to create.</param>
/// <param name="value">A value the new global parameter is to have.</param>
/// <param name="dimensionsToLabel">A set of dimension to labe by the new global parameter.</param>
/// <returns>ElementId of the new GlobalParameter</returns>

public ElementId CreateNewGlobalParameter(Document document, String name, double value, ISet<ElementId> dimensionsToLabel)
{
    if (!GlobalParametersManager.AreGlobalParametersAllowed(document))
        throw new System.InvalidOperationException("Global parameters are not permitted in the given document");
}
```

```
if (!GlobalParametersManager.IsUniqueName(document, name))

    throw new System.ArgumentException("Global parameter with such name already exists in the document", "name");

ElementId gpid = ElementId.InvalidElementId;

// creation of any element must be in a transaction

using (Transaction trans = new Transaction(document, "Create Global Parameter"))

{

    trans.Start();

    // create a GP with the given name and type Length

    GlobalParameter gp = GlobalParameter.Create(document, name, SpecTypeI
d.Length);

    if (gp != null)

    {

        // if created successfully, assign it a value

        // note: parameters of type Length accept Double values

        gp.SetValue(new DoubleParameterValue(value));

        // if a collection of dimensions was given, label them with this new parameter

        foreach (ElementId elemid in dimensionsToLabel)

        {

            // not just any dimension is allowed to be labeled

            // check first to avoid exceptions

            if (gp.CanLabelDimension(elemid))
```

```

    {
        gp.LabelDimension(elemid);
    }

    gpid = gp.Id;
}

trans.Commit();

}

return gpid;
}

```

Getting and setting the value of a global parameter

All global parameters, formula-driven, dimension-driven, or independent, have values. A value can be obtained by calling the `GetValue()` method. The object returned by that method is an instance of one of the classes derived from the `ParameterValue` class:

- `IntegerParameterValue`
- `DoubleParameterValue`
- `StringParameterValue`

All the derived classes have only one property, `Value`, which gets or sets the value as the corresponding type.

The concrete instance is determined by the type of the global parameter which was specified upon creation. Parameters that are neither formula-driven nor dimension-driven (reporting) can have a value assigned. The method to use is `SetValue()` and it accepts the same type of `ParameterValue` that is returned by `GetValue()`. However, the type can also be deduced easily: **Text** parameters accept only `StringParameterValue`. **Integer** and **YesNo** parameters accept only `IntegerParameterValue`. All other parameters accept only `DoubleParameterValue`.

Elements affected by a global parameter

Global parameters can be associated with other global parameters as well as regular family instance parameters (which may report global parameters as their values via the assignment formula.) There are two methods available to find relations among parameters: `GlobalParameter.GetAffectedGlobalParameters()` and `GlobalParameter.GetAffectedElements()`.

The former returns all other global parameters that refer to a particular global parameter in their respective formulas. The latter method returns a set of all elements of which some parameters are controlled by the global parameter. These two methods together with the `GlobalParameter.GetLabeledDimensions()` can help determine out how model elements relate to each other via global parameters.

Methods for maintaining associations between element properties and global parameters can be found in the [Parameter](#) class.

2.3.7.3 Reporting vs. Non-Reporting parameters

The most significant difference in types of global parameters is whether they are reporting or non-reporting.

What are reporting and non-reporting parameters?

There are several ways global parameters can be categorized, but probably the most significant categorization stems from the `GlobalParameter.IsReporting` property which divides global parameters into two groups - Reporting and Non-Reporting. The significance of reporting parameters lies in the fact that their values are driven by the dimension that has been labeled by a reporting parameter. It means that the value of a reporting parameter reflects the value of a dimension (length or angle) and gets updated anytime the dimension changes. Non-Reporting parameters behave in the opposite manner - they drive value of dimensions that have been labeled by them, which results in controlling the model's geometry through global parameters' values.

Reporting parameters are limited in several ways. They can be only of **Length** or **Angle** type, a requirement due to the fact that a dimension must be able to drive the value. For the same reason reporting parameters may not have formulas.

Non-Reporting parameters, on the other way, can be of almost any type (**Length**, **Integer**, **Area**, etc.) with the exception of `ElementId` type. Also, Non-Reporting parameters may have assigned formulas in which other global parameters may be used as arguments. This way one global parameter's value can be derived from other parameter (or parameters), and the other parameter can be either reporting or non-reporting.

Other important properties of global parameters are `IsDrivenByDimension` and `IsDrivenByFormula`, which are mutually exclusive - a parameter that has a formula assigned cannot be driven by a dimension (nor can be reporting) and vice versa.

Making a global parameter reporting or non-reporting

Global parameters are initially non-reporting upon creation, but can be set to reporting use the `GlobalParameter.IsReporting` property once a global parameter is created and is of an eligible type. Use `GlobalParameter.HasValidTypeForReporting()` to ensure that a certain data type can be made reporting. Note, that a parameter may not be made reporting after more than one dimension has been labeled by it. It is because reporting parameters can label (and be driven) by one dimension only.

An alternative way of making a parameter reporting is via the `GlobalParameter.SetDrivingDimension()` method which labels one dimension by a global parameter and also makes the parameter reporting if it is not reporting yet.

Although parameters driven by a dimension are automatically made reporting, parameters driven by a formula are not. In order to set a formula, the global parameter must be non-reporting. Therefore a reporting parameter must be changed to non-reporting first before assigning a formula.

2.3.7.4 *Formulas and Global Parameters*

Formulas may be assigned to non-reporting parameters.

As with family parameters, formulas may be assigned to global parameters using the `GlobalParameter.SetFormula()` method. Since a global parameter must be non-reporting to set a formula, a reporting parameter must be changed to non-reporting prior to assigning a formula.

Value of the evaluated formula must be compatible with the value-type of the parameter. For example, it is permitted to use **Integer** parameters in a formula assigned to a Double (**Number**) parameter, or vice versa. It is not allowed, however, to use **Length** or **Angle** arguments in a formula in a parameter whose type is either **Integer** or **Number**.

Formulas may include all standard arithmetic operations and logical operations (as functions **and**, **or**, **not**.) Input to logical operations must be Boolean values (parameters of YesNo type). Consequently, arithmetic operations can be applied to numeric values only. While there are no operations supported for string (text) arguments, strings can be used as results of a logical **If** operation. Depending on their type (and units), parameters of different value types can be combined. However, unit-less values such as **Integer** and **Number** (double) may only be combined with each other.

Since formulas can get quite complicated, and since some formulas cannot be assigned to certain parameters, the method `IsValidFormula()` can be used to test whether a formula is valid for a global parameter. If `SetFormula()` is called with an invalid formula for the global parameter, an Exception will be thrown.

`GetFormula()` will return the current formula in the form of a string.

The following code sample creates four global parameters and then sets a formula to one so that it has a value of either of two other parameters depending on the boolean value of the fourth one.

Code Region: Setting a formula

```
public void SetCombinationParameters(Document document)
{
    GlobalParameter gpB = null;
```

```
GlobalParameter gpT = null;
GlobalParameter gpF = null;
GlobalParameter gpX = null;

int TRUE = 1;
int FALSE = 0;

// transaction to create global parameters and set their values

using (Transaction trans = new Transaction(document, "Creating global parameters"))

{
    // create 4 new global parameters

    trans.Start();

    gpB = GlobalParameter.Create(document, "GPB", SpecTypeId.Boolean.YesNo);
    gpT = GlobalParameter.Create(document, "GPT", SpecTypeId.String.Text);
    gpF = GlobalParameter.Create(document, "GPF", SpecTypeId.String.Text);
    gpX = GlobalParameter.Create(document, "GPX", SpecTypeId.String.Text);

    // assign initial values and a formula to the global parameters

    gpB.SetValue(new IntegerParameterValue(TRUE));
    gpT.SetValue(new StringParameterValue("TypeA"));
    gpF.SetValue(new StringParameterValue("TypeB"));
}
```

```
// Set the formula to GPX so that its final value is either the value  
of GPT (TypeA)
```

```
// or GPF (TypeB) depending on whether the value of GPB is True or Fa  
lse.
```

```
// Note: in this particular case we are certain the formula is valid,  
but if weren't
```

```
// certain, we could use a validation method as we are now going to i  
llustrate here:
```

```
string expression = "if(GPB,GPT,GPF)"; // XPX <== if (GPB == TRUE) th  
en GPT else GPF
```

```
if (gpX.IsValidFormula(expression))
```

```
{
```

```
gpX.SetFormula(expression);
```

```
}
```

```
trans.Commit();
```

```
}
```

```
// we can test that the formula works
```

```
// since the boolean value is TRUE, the value of the GPX parameter
```

```
// should be the same as the value of the GPT parameters
```

```
StringParameterValue sTrue = gpT.GetValue() as StringParameterValue;
```

```
StringParameterValue sFalse = gpF.GetValue() as StringParameterValue;
```

```
StringParameterValue sValue = gpX.GetValue() as StringParameterValue;
```

```
if (sValue.Value != sTrue.Value)
```

```
{
```

```
    TaskDialog.Show("Error", "Unexpected value of a global parameter");

}

// we can also test that evaluation of the formula is affected by changes

using (Transaction trans = new Transaction(document, "Change value of a YesNo parameter"))

{
    trans.Start();
    gpB.SetValue(new IntegerParameterValue(FALSE));
    trans.Commit();
}

sValue = gpX.GetValue() as StringParameterValue;

if (sValue.Value != sFalse.Value)
{
    TaskDialog.Show("Error", "Unexpected value of a global parameter");
}

}
```

2.3.7.5 Labeling Dimensions with Global Parameters

A key feature of global parameters is their ability to "label" dimensions.

When a dimension is labeled by a global parameter, then its value is either controlled by the parameter (non-reporting), or drives the value of the parameter (reporting). It is important to note that a reporting parameter can label at most one dimension object - meaning, a parameter can be driven by one dimension only. If the dimension has several segments and is labeled by a

non-reporting parameter, the value of each segment will be driven by this parameter. Multi-segmented dimensions cannot be labeled by reporting parameters.

If the dimension is already labeled by another global parameter, labeling it again will automatically detach it from that parameter.

Presently, only single **Linear** and **Angular** dimensions can be labeled, but there are other restrictions in effect too. Use the `CanLabelDimension()` method to find out whether a particular dimension element may be labeled or not. Also, since the value of the parameter and the dimension labeled by it depend on each other, the data type of the global parameter must be either **Length** or **Angle**, since those are the only units a dimension can represent.

The following example creates a global parameter and uses it to label the set of given dimension elements.

Code Region: Labeling dimensions

```
public int DriveSelectedDimensions(Document document, string name, double value, ISet<ElementId> dimset)
{
    if (!GlobalParametersManager.AreGlobalParametersAllowed(document))
        throw new System.InvalidOperationException("Global parameters are not permitted in the given document");

    if (!GlobalParametersManager.IsUniqueName(document, name))
        throw new System.ArgumentException("Global parameter with such name already exists in the document", "name");

    if (value <= 0.0)
        throw new System.ArgumentException("Value of a global parameter that drives dimension must be a positive number", "value");

    int nLabeledDims = 0; // number of labeled dimensions (for testing)

    // creation of any element must be in a transaction
```

```
    using (Transaction trans = new Transaction(document, "Create Global Parameter"))
    {
        trans.Start();

        // create a GP with the given name and type Length
        // Note: Length (or Angle) is required type of global parameters that
        // are to label a dimension

        GlobalParameter newgp = GlobalParameter.Create(document, name, SpecTypeId.Length);

        if (newgp != null)
        {
            newgp.SetValue(new DoubleParameterValue(value));

            // use the parameter to label the given dimensions
            foreach (ElementId elemid in dimset)
            {
                // not just any dimension is allowed to be labeled
                // check first to avoid exceptions
                if (newgp.CanLabelDimension(elemid))
                {
                    newgp.LabelDimension(elemid);
                    nLabeledDims += 1;
                }
            }

            trans.Commit();
        }
    }
}
```

```
}

    // for illustration purposes only, we'll test the results of our modifications

    // 1.) Check the new parameter can be found

    ElementId gpid = GlobalParametersManager.FindByName(document, name);

    if (gpid == ElementId.InvalidElementId)
    {
        TaskDialog.Show("Error", "Failed to find a newly created global parameter");
    }

    GlobalParameter gp = document.GetElement(gpid) as GlobalParameter;

    // 2\). Check the number of labeled dimension is as expected

    ISet<ElementId> labeledSet = gp.GetLabeledDimensions();

    if (labeledSet.Count != nLabeledDims)
    {
        TaskDialog.Show("Error", "Have not found all the dimension that were labeled.");
    }

    return labeledSet.Count;
}
```

The SetDrivingDimension() method combines two actions: a) Making the parameter reporting if it is not yet, and b) Labeling the given dimension with it. Therefore, the global parameter must be eligible for reporting, and must not be used to label more than one dimensions yet. See the [Reporting vs. Non-Reporting Parameters](#) page for more information on reporting parameters

In case this parameter is already driven by another dimension, the other dimension will be unlabeled first before the given one is labeled. This is because a reporting parameter can only label one dimension at a time (i.e. it can be driven by one dimension only.)

The next example creates a global parameter to be driven by the value of a dimension.

Code Region: Assign driving dimension

```
public bool AssignDrivingDimension(Document document, ElementId gpid, ElementId dimid)
{
    // we expect to find the global parameter in the document
    GlobalParameter gp = document.GetElement(gpid) as GlobalParameter;
    if (gp == null)
        return false;

    // we expect to find the given dimension in the document
    Dimension dim = document.GetElement(dimid) as Dimension;
    if (dim == null)
        return false;

    // not every global parameter can label
    // and not every dimension can be labeled
    if (!gp.CanLabelDimension(dimid))
        return false;

    // we need a transaction to modify the model
}
```

```
    using (Transaction trans = new Transaction(document, "Assign a driving dimension"))

    {

        trans.Start();

        // we cannot assign a driving dimension to a global
        // parameter that is already used to label other dimensions

        ISet<ElementId> dimset = gp.GetLabeledDimensions();

        foreach (ElementId elemid in dimset)

        {

            gp.UnlabelDimension(elemid);

        }

        // with the GP free of all previously labels (if there were any)
        gp.SetDrivingDimension(dimid);

        // we should be able to commit, but we test the result anyway

        if (trans.Commit() != TransactionStatus.Committed)

            return false;

    }

    return true;

}
```

2.4 Collections

Most Revit Platform API properties and methods use .NET Framework collection classes when providing access to a group of related items.

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

2.4.1 Interface

The following sections discuss interface-related collection types.

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#.

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.

2.4.2 Collections and Iterators

In the Revit Platform API, Collections and Iterators are generic and type safe.

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.

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

Code Region 9-2: Using collections

```
public void UsingCollections(Document document)
{
    UIDocument uidoc = new UIDocument(document);

    ICollection<ElementId> selectedIds = uidoc.Selection.GetElementIds();

    // Store the ModelLine references
    ModelCurveArray lineArray = new ModelCurveArray();

    // ... Store operation
    Autodesk.Revit.DB.ElementId id = new Autodesk.Revit.DB.ElementId((long)131943); //assume 131943 is a model line element id
    lineArray.Append(document.GetElement(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  
  
    selectedIds.Remove(modelCurve.Id);  
  
  
    // 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 = Line.CreateBound(startPoint, endPoint);  
  
  
    // 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.");
}
```

2.5 Editing Elements

In Revit, you can move, copy, rotate, align, 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.

2.5.1 Moving Elements

The ElementTransformUtils class provides two static methods to move one or more elements from one place to another.

Table 19: Move Methods

Member	Description
<code>MoveElement(Document, ElementId, XYZ)</code>	Move an element in the document by a specified vector.
<code>MoveElements(Document, ICollection<ElementId>, XYZ)</code>	Move several elements by a set of IDs in the document by a specified vector.

Note: When you use the MoveElement() or MoveElements() methods, the following rules apply.

The methods 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).

Code Region 10-1: Using MoveElement()

```

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);
    ElementTransformUtils.MoveElement(document, column.Id, 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.
- Pinned Elements cannot be moved.

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.ApplicationServices.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.

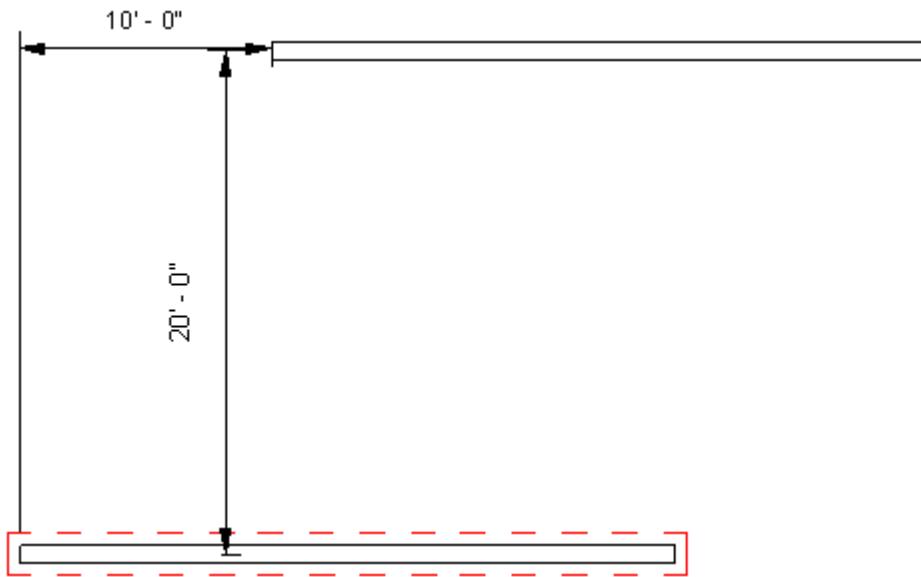


Figure 30: Move a wall using the LocationCurve

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 = Line.CreateBound(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;
    }
}
```

{}

2.5.2 Copying Elements

The `ElementTransformUtils` class provides several static methods to copy one or more elements from one place to another, either within the same document or view, or to a different document or view.

Table: Copy Methods

Member	Description
<code>CopyElement(Document, ElementId, XYZ)</code>	Copies an element and places the copy at a location indicated by a given transformation..
<code>CopyElements(Document, ICollection<ElementId>, XYZ)</code>	Copies a set of elements and places the copies at a location indicated by a given translation.
<code>CopyElements(Document, ICollection<ElementId>, Document, Transform, CopyPasteOptions)</code>	Copies a set of elements from source document to destination document.
<code>CopyElements(View, ICollection<ElementId>, View, Transform, CopyPasteOptions)</code>	Copies a set of elements from source view to destination view.

All of the methods return a collection of `ElementIds` of the newly created elements, including `CopyElement()`. The collection includes any elements created due to dependencies.

The method for copying from one document to another can be used for copying non-view specific elements only. Copies are placed at their respective original location or locations specified by the optional transformation.

View-specific elements should be copied using the method that copies from one view to another. That method can be used for both view-specific and model elements however, drafting views cannot be used as a destination for model elements. The pasted elements are repositioned to ensure proper placement in the destination view. For example, the elevation is changed when copying from one level to another. An additional transformation within the destination view can be performed by providing the optional `Transform` argument. This additional transformation must be within the plane of the destination view.

When copying from one view to another, both the source and destination views must be 2D graphics views capable of drawing details and view-specific elements, such as floor and ceiling plans, elevations, sections, or drafting views. The `ElementTransformUtils.GetTransformFromViewToView()` method will return the transformation that is applied to elements when copying from a source view to a destination view.

When copying between views or between documents, an optional `CopyPasteOptions` parameter may be set to override default copy/paste settings. By default, in the event of duplicate type

names during a paste operation, Revit displays a modal dialog with options to either copy types with unique names only, or to cancel the operation. `CopyPasteOptions` can be used to specify a custom handler, using the `IDuplicateTypeNamesHandler` interface, to handle duplicate type names.

See the Duplicate Views sample in the Revit SDK for a detailed example of copying between documents and between views.

2.5.3 Rotating elements

The `ElementTransformUtils` class provides two static methods to rotate one or several elements in the project.

Table 20: Rotate Methods

Member	Description
<code>RotateElement(Document, ElementId, Line, double)</code>	Rotate an element in the document by a specified number of radians around a given axis.
<code>RotateElements(Document, ICollection<ElementId>, Line, double)</code>	Rotate several elements by IDs in the project by a specified number of radians around a given axis.

In these 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.

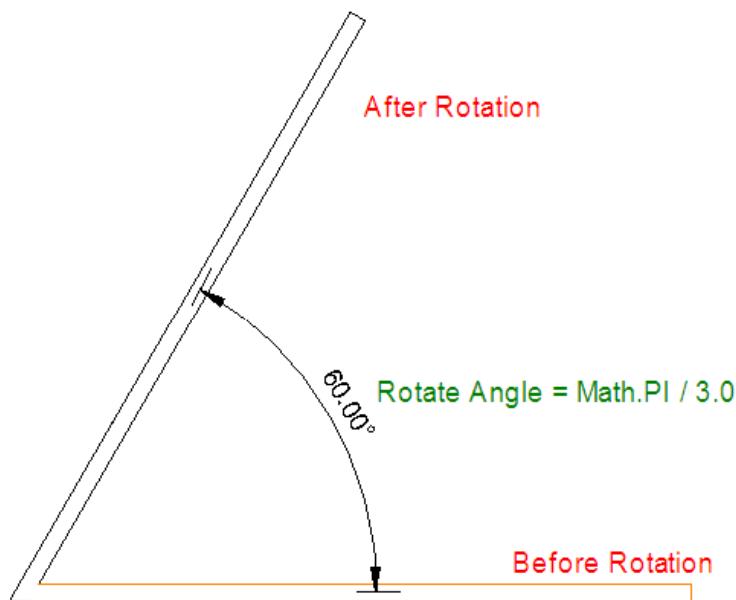
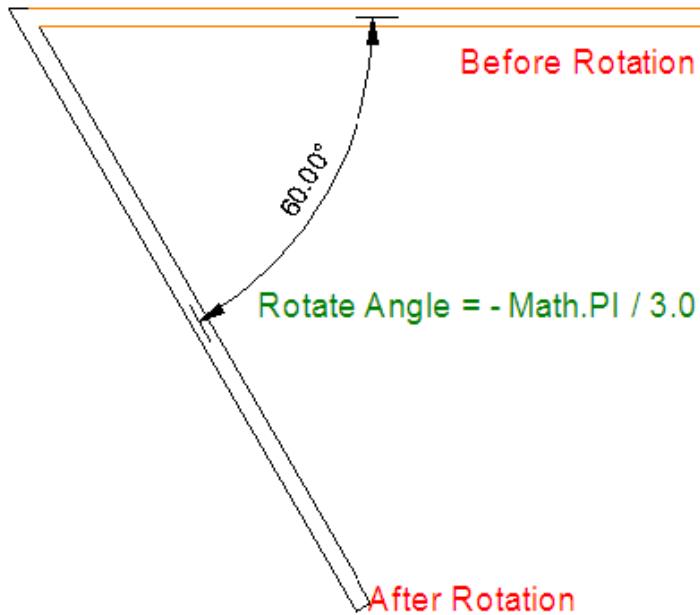


Figure 31: Counterclockwise rotation

**Figure 32: Clockwise rotation**

Note that pinned elements cannot be rotated.

Code Region 10-5: Using RotateElement()

```
public void RotateColumn(Autodesk.Revit.DB.Document document, Autodesk.Revit.DB.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 = Line.CreateBound(point1, point2);
    ElementTransformUtils.RotateElement(document, element.Id, axis, Math.PI /
    3.0);
}
```

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.DB.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.GetEndPoint(0);

        XYZ cc = new XYZ(aa.X, aa.Y, aa.Z + 10);

        Line axis = Line.CreateBound(aa, cc);

        rotated = curve.Rotate(axis, Math.PI / 2.0);
    }

    return rotated;
}
```

Code Region 10-7: Rotating based on location point

```
bool LocationPointRotate(Autodesk.Revit.ApplicationServices.Application application, 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 = Line.CreateBound(aa, cc);
    rotated = location.Rotate(axis, Math.PI / 2.0);

}

return rotated;
}

```

2.5.4 Aligning Elements

The ItemFactoryBase.NewAlignment() method can create a new locked alignment between two references. These two references must be one of the following combinations:

- 2 planar faces
- 2 lines
- line and point
- line and reference plane
- 2 arcs
- 2 cylindrical faces

These references must be already geometrically aligned as this function will not force them to become aligned. If the alignment can be created a new Dimension object is returned representing the locked alignment. Otherwise an exception will be thrown.

The NewAlignment() method also requires a view which will determine the orientation of the alignment.

See the CreateTruss example in the FamilyCreation folder included with the SDK Samples. It has several examples of the use of NewAlignment(), such as locking the bottom chord of a new truss to a bottom reference plane.

2.5.5 Mirroring Elements

The ElementTransformUtils class provides two static methods to mirror one or more elements in the project.

Table 21: Mirror Methods

Member	Description
<code>MirrorElement(Document, ElementId, Plane)</code>	Mirror one element about a geometric plane.
<code>MirrorElements(Document, ICollection<ElementId>, Plane, Boolean)</code>	Mirror several elements about a geometric plane. Can be performed on original geometry or a copy.

After performing the mirror operation, you can access the new elements from the Selection ElementSet.

`ElementTransformUtils.CanMirrorElement()` and `ElementTransformUtils.CanMirrorElements()` can be used to determine if one or more elements can be mirrored prior to attempting to mirror an element.

The following code illustrates how to mirror a wall using a plane calculated based on a side face of the wall.

Code Region 10-8: Mirroring a wall

```
public void MirrorWall(Autodesk.Revit.DB.Document document, Wall wall)
{
    Reference reference = HostObjectUtils.GetSideFaces(wall, ShellLayerType.Exterior).First();

    // get one of the wall's major side faces
    Face face = wall.GetGeometryObjectFromReference(reference) as Face;

    UV bboxMin = face.GetBoundingBox().Min;
    // create a plane based on this side face with an offset of 10 in the
    // X & Y directions
    Plane plane = Plane.CreateByNormalAndOrigin(face.ComputeNormal(bboxMin),
        face.Evaluate(bboxMin).Add(new XYZ(10, 10, 0)));
}
```

```
ElementTransformUtils.MirrorElement(document, wall.Id, plane);  
}
```

Every FamilyInstance has a Mirrored property. It indicates whether a FamilyInstance (for example a column) is mirrored.

2.5.6 Grouping Elements

The Revit API uses the Creation.Document.NewGroup() method to select an element or multiple elements or groups and then combine 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

```
public void MakeGroup(Document document)  
{  
    Group group = null;  
  
    UIDocument uidoc = new UIDocument(document);  
  
    ICollection<ElementId> selectedIds = uidoc.Selection.GetElementIds();  
  
  
    if (selectedIds.Count > 0)  
    {  
        // Group all selected elements  
  
        group = document.Create.NewGroup(selectedIds);  
  
        // 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:  
  
        // 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.

Attached Detail Groups

To determine if a group is an attached detail group, use the `Group.IsAttached` property. To find the group to which a detail group is attached, use the `Group.AttachedParentId` property.

If a Model Group has attached detail groups, the visibility of these detail groups can be controlled with:

- `Group.ShowAttachedDetailGroups()`
- `Group.ShowAllAttachedDetailGroups()`
- `Group.HideAttachedDetailGroups()`
- `Group.HideAllAttachedDetailGroups()`

The attached detail groups available for a group or group type can be found with:

- `Group.GetAvailableAttachedDetailGroupTypeIds()`
- `GroupType.GetAvailableAttachedDetailGroupTypeIds()`

`Group.IsCompatibleAttachedDetailGroupType()` checks if the orientation of an attached detail group matches the orientation of a view. To prevent displaying detail groups in the wrong view, verify that the `OwnerViewId` of a detail group matches the view in which you are trying to display it.

Loading Groups from File

The `GroupLoadOptions` class provides options for loading a Revit group from file.

It has the following properties:

- `ReplaceDuplicatedGroups` - If there are groups with the same names in source and destination documents set this property to true to replace existing groups, otherwise the operation will be canceled. The default value is false.
- `IncludeGrids` - Returns true if grids should be brought in from the input file, false otherwise.
- `IncludeLevels` - Returns true if levels should be brought in from the input file, false otherwise.
- `IncludeAttachedDetails` - Returns true if attached detail groups should be included, false otherwise.

The methods:

- `GroupLoadOptions.GetDuplicateTypeNamesHandler()`
- `GroupLoadOptions.SetDuplicateTypeNamesHandler(IDuplicateTypeNamesHandler)`

allow you to set or retrieve a duplicate type names handler. If this value is not set, the default handler is used.

`GroupType.LoadFrom(string, GroupLoadOptions)` allows a group type to be replaced with the contents of the input file.

2.5.7 Creating Arrays of Elements

The Revit Platform API provides two classes, `LinearArray` and `RadialArray` to array one or more elements in the project. These classes provide static methods to 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.

Both `LinearArray` and `RadialArray` also provide methods to array one or several elements without being grouped and associated. Although similar to the `Create()` methods for arraying elements, each resulting element is independent of the others, and can be manipulated without

affecting the other elements. See the tables below for more information on the methods available to create linear or radial arrays.

Table 22: LinearArray Methods

Member	Description
<code>Create(Document, View, ElementId, int, XYZ, ArrayAnchorMember)</code>	Array one element in the project by a specified number.
<code>Create(Document, View, ICollection<ElementId>, int, XYZ, ArrayAnchorMember)</code>	Array a set of elements in the project by a specified number.
<code>ArrayElementWithoutAssociation(Document, View, ElementId, int, XYZ, ArrayAnchorMember)</code>	Array one element in the project by a specified number. The resulting elements are not associated with a linear array.
<code>ArrayElementsWithoutAssociation(Document, View, ICollection<ElementId>, int, XYZ, ArrayAnchorMember)</code>	Array a set of elements in the project by a specified number. The resulting elements are not associated with a linear array.

Table 23: RadialArray Methods

Member	Description
<code>Create(Document, View, ElementId, int, Line, double, ArrayAnchorMember)</code>	Array one element in the project based on an input rotation axis.
<code>ArrayElementWithoutAssociation(Document, View, ElementId, int, Line, double, ArrayAnchorMember)</code>	Array one element in the project based on an input rotation axis.. The resulting elements are not associated with a linear array.
<code>ArrayElementsWithoutAssociation(Document, View, ICollection<ElementId>, int, Line, double, ArrayAnchorMember)</code>	Array a set of elements in the project based on an input rotation axis.. The resulting elements are not associated with a linear array.

The methods for arraying elements are 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 methods for arraying elements, 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.

2.5.8 Deleting Elements

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

Table 23: Delete Members

Member	Description
Delete(ElementId)	Delete an element from the project using the element ID
Delete(IICollection)	Delete several elements from the project by their IDs.

The first method deletes a single element based on its Id, as shown in the example below.

Code Region: 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 sample above.

The API also provides a way to delete several elements.

Code Region: Deleting multiple elements based on Id

```
public void DeleteSelected(Document document)
{
    // Delete all the selected elements via the set of elements
    UIDocument uidoc = new UIDocument(document);

    ICollection<ElementId> elements = uidoc.Selection.GetElementIds();

    ICollection<Autodesk.Revit.DB.ElementId> deletedIdSet = document.Delete(elements);

    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.

Element Dependencies

`Element.GetDependentElements()` returns a list of ids of elements which are "children" of this element; that is, those elements which will be deleted along with this element. The method optionally takes an `ElementFilter` which can be used to reduce the output list to the collection of elements matching specific criteria.

2.5.9 Pinned Elements

Elements can be pinned to prevent them from moving. The `Element.Pinned` property can be used to check if an Element is pinned or to pin or unpin an element.

When `Element.Pinned` is set to true, the element cannot be moved or rotated.

2.6 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.

The `Autodesk.Revit.DB.View` class is the base class for all view types in the Revit document. The `Autodesk.Revit.UI.UIView` class represents the window view in the Revit user interface.

In the following sections, you learn how views are generated, the types of views supported by Revit, the features for each view, as well as the functionality available for view windows in the user interface.

2.6.1 About views

The Revit API provides access to View properties and the ability to create and delete views programmatically.

This section is a high-level overview that includes the following:

- How views are generated
- View types
- View navigation tools
- Creating and deleting views.

View process

The following figure illustrates how a view is generated.

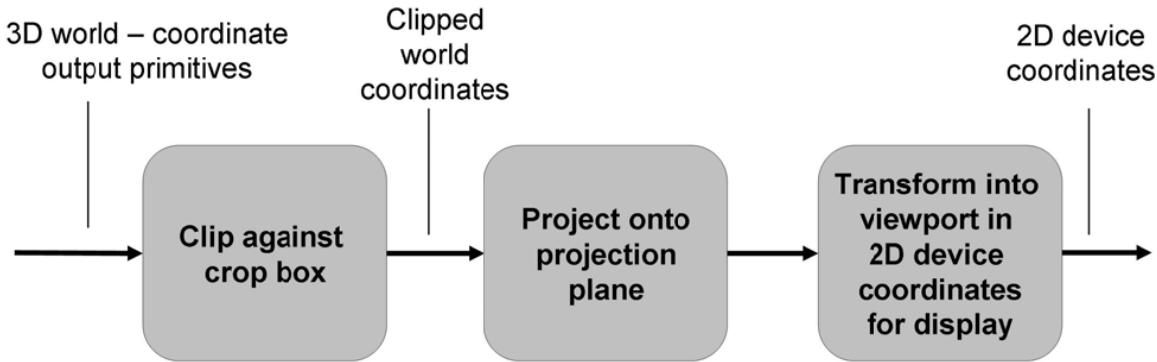


Figure 94: Create view process

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 [View3D](#) and the Parallel Projection picture in [View3D](#).

Some portions of a 3D model space that do not display, such as those that are behind the viewer or are 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 region are no longer in the view.

- The View.GetCropRegionShapeManager method returns a ViewCropRegionShapeManager which provides the boundary information for the crop region, which may or may not be rectangular.
- 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 shape on the projection plane.
- The size and position of the crop region is determined by the View.OutLine property.

You can transform between model space and a view's projection space with the following methods:

- TransformWithBoundary.GetModelToProjectionTransform()
- TransformWithBoundary.GetBoundary() - Returns the boundary for the model space to view projection space transform.
- View.GetModelToProjectionTransforms() - Gets the transforms from the model space to the view projection space. Views with split crop regions have more than one transform.
- View.HasViewTransforms() - Returns true if the view reports model space to view projection space transforms. Schedules and legends, for example, do not report any.

Transforming between a view's projection space and sheet space can be done with `Viewport.GetProjectionToSheetTransform()`.

View navigation tools

You may access information about the home view camera currently set in the view cube settings. There may only be one home view camera set for the document. This corresponds to the view orientation and other camera parameters saved when the user invokes the ViewCube UI command to "Set current view as home" in the ViewCube right-click context menu.

Access the ViewNavigationToolSettings by calling the static method `ViewNavigationToolSettings.GetViewNavigationToolSettings()` which will return the `ViewNavigationToolSettings` element associated with the document.

The `ViewNavigationToolSettings` will allow you to query whether a home view has been set with the method `IsHomeCameraSet()` which returns a boolean indicating the current state of the home view setting.

Access read-only information about the home camera set in the ViewCube by getting a copy of the home camera with the ViewNavigationToolSettings.GetHomeCamera() method. This function returns Null if the home camera is not yet set. The HomeCamera class provides information about the camera and view for the Home view orientation stored in the model such as EyePosition and UpDirection.

Creating and deleting views

The Revit Platform API provides numerous methods to create the corresponding view elements derived from Autodesk.Revit.DB.View class. Most view types are created using static methods of the derived view classes. If a view is created successfully, these methods return a reference to the view, otherwise they return null. The methods are described in the following sections specific to each view class.

Views can also be created using the View.Duplicate() method. A new view can be created from an existing view with options for the new view to be dependent or to have detailing. The following example demonstrates how to create a new dependent view.

Code Region: Create a dependent view

```
public View CreateDependentCopy(View view)
{
    View dependentView = null;
    ElementId newViewId = ElementId.InvalidElementId;
    if (view.CanViewBeDuplicated(ViewDuplicateOption.AsDependent))
    {
        newViewId = view.Duplicate(ViewDuplicateOption.AsDependent);
        dependentView = view.Document.GetElement(newViewId) as View;
        if (null != dependentView)
        {
            if (dependentView.GetPrimaryViewId() == view.Id)
            {
                TaskDialog.Show("Dependent View", "Dependent view created successfully!");
            }
        }
    }
}
```

```

    }

}

return dependentView;
}

```

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.

Dependent Views

As seen above, dependent views can be created using the View.Duplicate() method and passing in the AsDependent value for the ViewDuplicationOption enumerator. A dependent view will remain synchronous with the primary view and all other dependent views , so that when view-specific changes (such as view scale and annotations) are made in one view, they are reflected in all views. Not all view types can be duplicated and you cannot create a dependent view from another dependent view. Use View.CanViewBeDuplicated() to make sure a dependent view can be generated from the view. This method takes a ViewDuplicationOption enum to check whether a view can be duplicated in a specific way. You may be able to duplicate a view as an independent view, but not a dependent view.

Dependent views have a valid primary view element Id that can be obtained from the method View.GetPrimaryViewId(). Independent views have InvalidElementId as their primary view Id. A dependent view can be converted to an independent view using the View.ConvertToIndependent() method. This method will thrown an exception if the view is not dependent.

Code Region: Make a dependent view independent

```

public void MakeViewIndependent(View view)
{
    // Independent views will have an InvalidElementId for the Primary View Id
    if (view.GetPrimaryViewId() != ElementId.InvalidElementId)
    {

```

```

        view.ConvertToIndependent();

    }

}

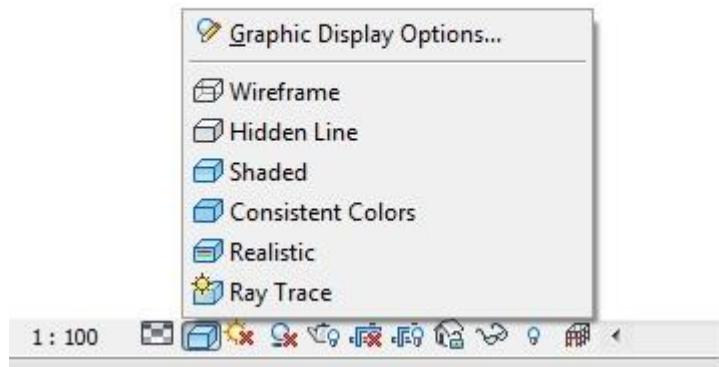
```

2.6.2 View Graphics

Many elements of the graphics and display options for views are exposed via the API.

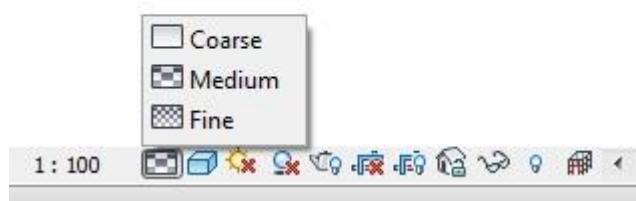
Display settings

The view class has properties to get and set the display style settings and the detail level settings. The View.DisplayStyle property uses the `DisplayStyle` enumeration and corresponds to the display options available at the bottom of the Revit window as shown below.



Note: The display style cannot be set to Raytrace from the Revit API because this display style puts Revit into a restricted mode with limited capabilities.

The View.DetailLevel property uses the `ViewDetailLevel` enumeration and corresponds to the detail level options available at the bottom of the Revit window as shown below.



The `ViewDetailLevel` enumeration includes `Undefined` in the case that a given View does not use detail level.

Thin Lines

The Thin Lines option, available in the Revit UI on the Graphics panel of the View tab, controls how lines are drawn in a view. Typically, when you zoom in on a model in a small scale view, element lines appear much thicker than they actually are. When Thin Lines is enabled, all lines are drawn as a single width regardless of zoom level. This option is made available via the `ThinLinesOptions` utility class, which has a property called `AreThinLinesEnabled`. It is a static property which affects the entire Revit session.

Temporary view modes

The `TemporaryViewModes` class allows for control of temporary view modes. It can be accessed from the `View.TemporaryViewModes` property. For views that do not support temporary view modes, this property will be null. The `RevealConstraints`, `RevealHiddenElements` and `WorksharingDisplay` properties can be used to get and set the current state of these modes in the corresponding view. Note that some modes are only available in certain views and/or in a certain context. Additionally an available mode is not necessarily enabled in the current context. The `TemporaryViewModes` methods `IsModeAvailable()` and `IsModeEnabled()` can be used to test that a particular mode is both available and enabled before use. These methods take a `TemporaryViewMode` enum. The possible options are shown below.

Member Name	Description
<code>RevealHiddenElements</code>	The reveal hidden elements mode
<code>TemporaryHidesIsolate</code>	The temporary hide/isolate mode
<code>WorksharingDisplay</code>	One of the worksharing display modes
<code>TemporaryViewProperties</code>	The temporary View Properties mode
<code>ExplodedView</code>	The mode that shows the model in exploded view and allows user changes/configurations
<code>RevealConstraints</code>	The mode that reveals constraints between elements in the model
Code Region: Reveal hidden elements in a view	
<pre>public bool RevealHiddenElementsInView(View view) { </pre>	

```
bool hiddenRevealed = false;

TemporaryViewModes viewModes = view.TemporaryViewModes;

if (viewModes == null)
{
    TaskDialog.Show("Invalid View", "This view does not support temporary
view modes.");
}

else
{
    // Mode must be available and enabled to be activated

    if (viewModes.IsEnabled(TemporaryViewMode.RevealHiddenElements) &
& viewModes.IsModeAvailable(TemporaryViewMode.RevealHiddenElements))

    {
        viewModes.RevealHiddenElements = true;

        hiddenRevealed = viewModes.RevealHiddenElements;
    }
}

return hiddenRevealed;
}
```

The `TemporaryViewModes.IsEnabled()` method tests whether a given mode is currently active in the view. Use the `DeactivateAllModes()` method to deactivate all currently active views, or use `DeactivateMode()` to deactivate a specific mode.

The `PreviewFamilyVisibility` property gets and sets the current state of the `PreviewFamilyVisibility` mode in the associated view. This mode is only available when the document of the view is in the environment of the family editor. This property is a `PreviewFamilyVisibilityMode` enumerated value rather than a `bool`. Possible states for this mode are:

Member	Description
Name	
Off	Element Visibility is not applied. All family elements visible.
On	Element Visibility of a view is applied to show visible elements only. Elements that are cut by a reference plane will be shown with their respective cut geometry.
Uncut	Element Visibility of a view is applied to show elements visible if instance is not cut. Note that this state is only available in certain views, such as floor plan and ceilings.

Even if the PreviewFamilyVisibility mode is available and enabled for a view, not all states are valid in all views. Before applying one of these states to a view, call IsValidState() to ensure it can be applied.

Code Region: Turn off preview family visibility mode

```
public void TurnOffFamilyVisibilityMode(View view)
{
    TemporaryViewModes viewModes = view.TemporaryViewModes;

    if (viewModes != null)
    {
        if (viewModes.IsEnabled(TemporaryViewMode.PreviewFamilyVisibility) && viewModes.IsValidState(PreviewFamilyVisibilityMode.Off))
        {
            viewModes.PreviewFamilyVisibility = PreviewFamilyVisibilityMode.Off;
        }
    }
}
```

```

    }
}

```

When a view is opened for the first time, its state of the PreviewFamilyVisibility mode is determined based on the default settings which is controlled through the static TemporaryViewModes properties PreviewFamilyVisibilityDefaultOnState and PreviewFamilyVisibilityDefaultUncutState as shown below.

Code Region: Set default preview family visibility state

```

public void SetDefaultPreviewFamilyVisibilityState()
{
    TemporaryViewModes.PreviewFamilyVisibilityDefaultOnState = true;
    TemporaryViewModes.PreviewFamilyVisibilityDefaultUncutState = true;
}

```

The PreviewFamilyVisibilityDefaultOnState value controls whether each newly opened view is to have the PreviewFamilyVisibility mode turned On by default or not. This property is applicable to all types of views. Views that support both cut and uncut previews (such as floor plans) will use the cut/uncut state indicated by the PreviewFamilyVisibilityDefaultUncutState property, but only if the PreviewFamilyVisibilityDefaultOnState property is set to true.

These settings are applicable to the whole application rather than to individual family documents; the values persists between Revit sessions. Although the value is allowed to be set at any time, any changes made after the Revit application has been initialized will not have effect until the next session of Revit.

Note that once the PreviewFamilyVisibility property is explicitly modified, the applied setting stays in effect for the respective view even after the view is closed and later reopened again.

Custom Temporary View Modes

The Revit API allows customization to create "custom temporary view modes" applied to a view as a temporary view property. For example, the "Reveal Obstacles for Path of Travel" temporary mode is implemented this way. It uses the Analysis Visualization Framework (AVF) to display additional graphics on top of the view contents.

These properties provide access to read and modify a custom temporary view mode. `CustomTitle` should be set to cause the view to display the customized frame. The application is responsible to adjust the appearance of elements in the view related to the mode.

- TemporaryViewModes.CustomTitle
- TemporaryViewModes.CustomColor
- TemporaryViewModes.RemoveCustomization()
- TemporaryViewModes.IsCustomized()

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.

Individual elements can be hidden in a particular view. The method Element.IsNullOrHidden() indicates if an element is hidden in the given view, and Element.CanBeHidden() returns whether the element can be hidden. To hide individual elements, use View.HideElements() which takes a collection of ElementIds corresponding to the elements you wish to hide.

Element visibility can also be changed by category.

- The View.GetCategoryHidden() method queries a category id to determine if it is hidden or visible in the view.
- The View.SetCategoryHidden() method sets all elements in a specific category to hidden or visible.
- The View.CanCategoryBeHidden() method indicates if a specific category of elements can be hidden in the view.

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: Counting elements in the active view

```
private void CountElements(UIDocument uiDoc)
{
}
```

```

    StringBuilder message = new StringBuilder();

    FilteredElementCollector viewCollector =
        new FilteredElementCollector(uiDoc.Document, uiDoc.ActiveView.Id);

    viewCollector.OfCategory(BuiltInCategory.OST_Walls);

    message.AppendLine("Wall category elements within active View: "
        + viewCollector.ToElementIds().Count);

    FilteredElementCollector docCollector = new FilteredElementCollector(uiDo
c.Document);

    docCollector.OfCategory(BuiltInCategory.OST_Walls);

    message.AppendLine("Wall category elements within document: "
        + docCollector.ToElementIds().Count);

    TaskDialog.Show("Revit", message.ToString());
}

```

Temporary view modes can affect element visibility. The `View.IsInTemporaryViewMode()` method can be used to determine if a View is in a temporary view mode. The method `View.IsElementVisibleInTemporaryViewMode()` identifies if an element should be visible in the indicated view mode. This applies only to the `TemporaryHidelsolate` and `AnalyticalModel` view modes. Other modes will result in an exception.

Depth Cueing

The `ViewDisplayDepthCueing` class provides control of the display of distant objects in section and elevation views. When depth cueing is active, objects blend into the background color (fade) with increasing distance from the viewer. The current depth cueing settings for a view can be retrieved using `View.GetDepthCueing()`. If changes are made to the returned `ViewDisplayDepthCueing`, they will not be applied to the view until calling `View.SetDepthCueing()`.

The `ViewDisplayDepthCueing` class has the following properties:

Member Name	Description

EnableDepthCueing	True to enable depth cueing.
StartPercentage	Indicates where depth cueing begins. A value of 0 indicates that depth cueing begins at the front clip plane of the view.
EndPercentage	Indicates where depth cueing ends. Objects further than the end plane will fade the same amount as objects at the end plane. A value of 100 indicates the far clip plane.
FadeTo	Indicates the maximum amount to fade objects via depth cueing. A value of 100 indicates complete invisibility.

The `SetStartEndPercentages()` method can be used to set the start and end percentages in one call.

The following example demonstrates how to get the current depth cueing, change its properties and set it back to the view. Note that not all views can use depth cueing.

Code Region: Change the depth cueing for a view

```
private void AdjustDepthCueing(View view)
{
    if (view.CanUseDepthCueing())
    {
        using (Transaction t = new Transaction(view.Document, "Change depth cueing"))
        {
            t.Start();
            ViewDisplayDepthCueing depthCueing = view.GetDepthCueing();
            depthCueing.EnableDepthCueing = true;
            depthCueing.FadeTo = 50;      // set fade to percent
            depthCueing.SetStartEndPercentages(0, 75);
            view.SetDepthCueing(depthCueing);
        }
    }
}
```

```

        t.Commit();
    }
}
}

```

2.6.3 View Types

Different types of Revit views are represented by different classes in the Revit API. See the following topics for more information on each type of view.

2.6.3.1 Overview

A project model can have several view types. In the API, there are three 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.

Table 44: Autodesk.Revit.DB.ViewType

Member Name	Description
AreaPlan	Area view.
CeilingPlan	Reflected ceiling plan view.
ColumnSchedule	Column 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.
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.

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.

Table 45: Project Browser Views

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.ViewSection
Structural Plans	ViewType.EngineeringPlan	Elements.ViewPlan
Floor Plans	ViewType.FloorPlan	Elements.ViewPlan
Legends	ViewType.Legend	Elements.View
Reports (MEP engineering)	ViewType.LoadsReport	Elements.View
Reports (MEP engineering)	ViewType.PanelSchedule	Elements.PanelScheduleView
Reports (MEP engineering)	ViewType.PresureLossReport	Elements.View

Renderings	ViewType.Rendering	Elements.ViewDrafting
Reports	ViewType.Report	Elements.View
Schedules/Quantities	ViewType.Schedule	Elements.ViewSchedule
Sections	ViewType.Section	Elements.ViewSection
3D Views	ViewType.ThreeD	Elements.View3D
Walkthroughs	ViewType.Walkthrough	Elements.View3D

This example shows how to use the ViewType property of a view to determine the view's type.

Code Region: 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 += "a floor plan view.";
        break;

    // ...

    default:
        break;
    }

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

The third way to classify views is using the ViewFamilyType class. Most view creation methods required the Id of a ViewFamilyType for the new view. The Id of the ViewFamilyType can be retrieved from the View.GetTypeId() method. The ViewFamilyType.ViewFamily property returns a ViewFamily enumeration which specifies the family of the ViewFamilyType and similar to the ViewType enum documented above. The following example shows how to get the ViewFamily from a View.

Code Region: Determining view type from ViewFamilyType

```
public ViewFamily GetViewFamily(Document doc, View view)
{
    ViewFamily viewFamily = ViewFamily.Invalid;

    ElementId viewTypeId = view.GetTypeId();

    if (viewType.Id > 1) // some views may not have a ViewFamilyType
    {
        ViewFamilyType viewFamilyType = doc.GetElement(viewType) as ViewFamilyType;
        viewFamily = viewFamilyType.ViewFamily;
    }

    return viewFamily;
}
```

2.6.3.2 View3D

View3D is a freely-oriented three-dimensional view.

There are two kinds of 3D views, perspective and isometric, also referred to as orthographic in the Revit user interface. The difference is based on the projection ray relationship. The View3D.IsPerspective property indicates whether a 3D view is perspective or isometric.

Perspective view

The following picture illustrates how a perspective view is created.

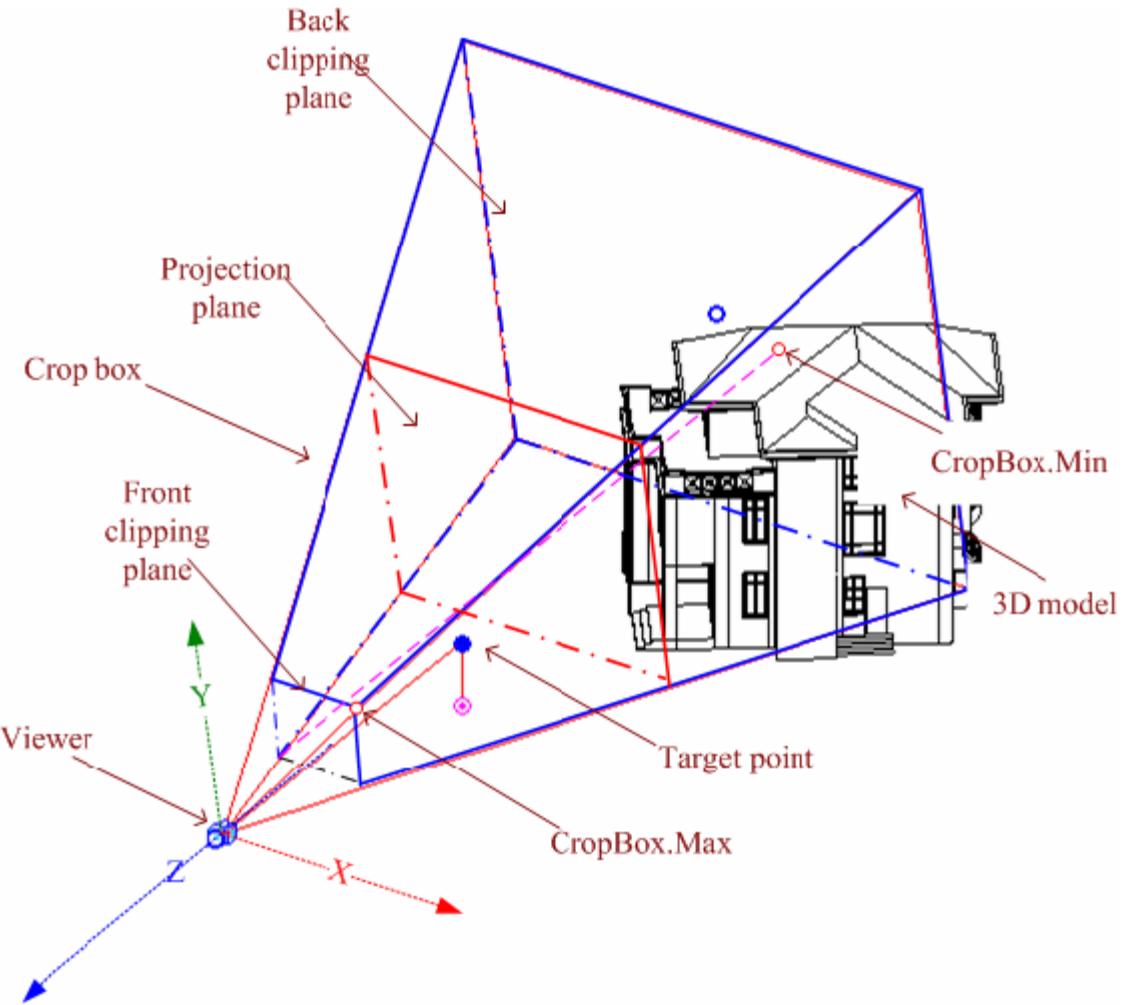


Figure 96: Perspective projection

- 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 `ViewOrientation3D.EyePosition` property (retrieved from `View3D.GetOrientation()`). Therefore, in 3D views, `View.Origin` is always equal to the corresponding `ViewOrientation3D.EyePosition`.
- As described in the diagram above, 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 static method View3D.CreatePerspective() method can be used to create new perspective views. The viewFamilyTypeId parameter needs to be a three dimensional ViewType.

The following code sample illustrates how to create a perspective 3D view.

Code Region: Creating a Perspective 3D view

```
public void CreatePerspective(Document document)
{
    // Find a 3D view type
    IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in new FilteredElementCollector(document).OfClass(typeof(ViewFamilyType))
        let type = elem as ViewFamilyType
        where type.ViewFamily == ViewFamily.ThreeDimensional
        select type;

    // Create a new Perspective View3D
    View3D view3D = View3D.CreatePerspective(document, viewFamilyTypes.First().Id);

    if (null != view3D)
    {
        // By default, the 3D view uses a default orientation.

        // Change the orientation by creating and setting a ViewOrientation3D

        XYZ eye = new XYZ(0, -100, 10);
        XYZ up = new XYZ(0, 0, 1);
        XYZ forward = new XYZ(0, 1, 0);
        view3D.SetOrientation(new ViewOrientation3D(eye, up, forward));

        // turn off the far clip plane with standard parameter API
        Parameter farClip = view3D.LookupParameter("Far Clip Active");
    }
}
```

```

        farClip.Set(0);

    }

}

```

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 [Geometry Helper Classes in the Geometry section](#).

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.

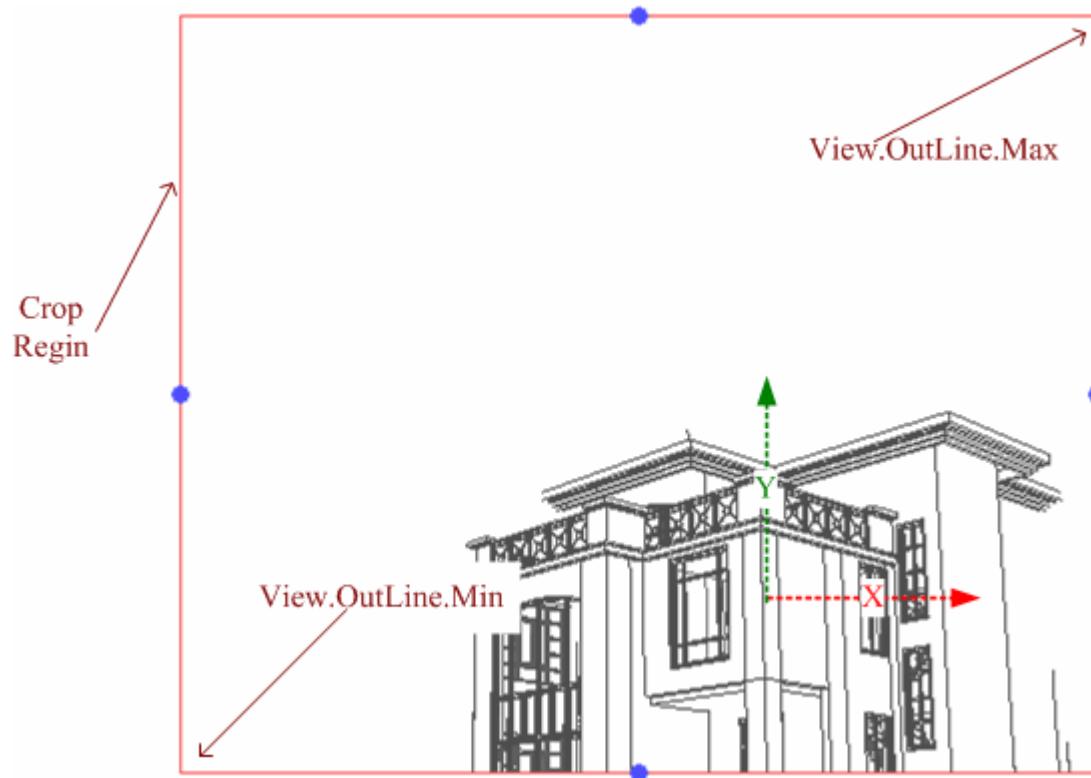


Figure 97: Perspective 3D view

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.Outlet.Max property points to the upper right corner.
- The View.Outlet.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.Outlet.Max.X(Y) == View.CropBox.Min.X(Y) / View.Outlet.Min.X(Y)
```

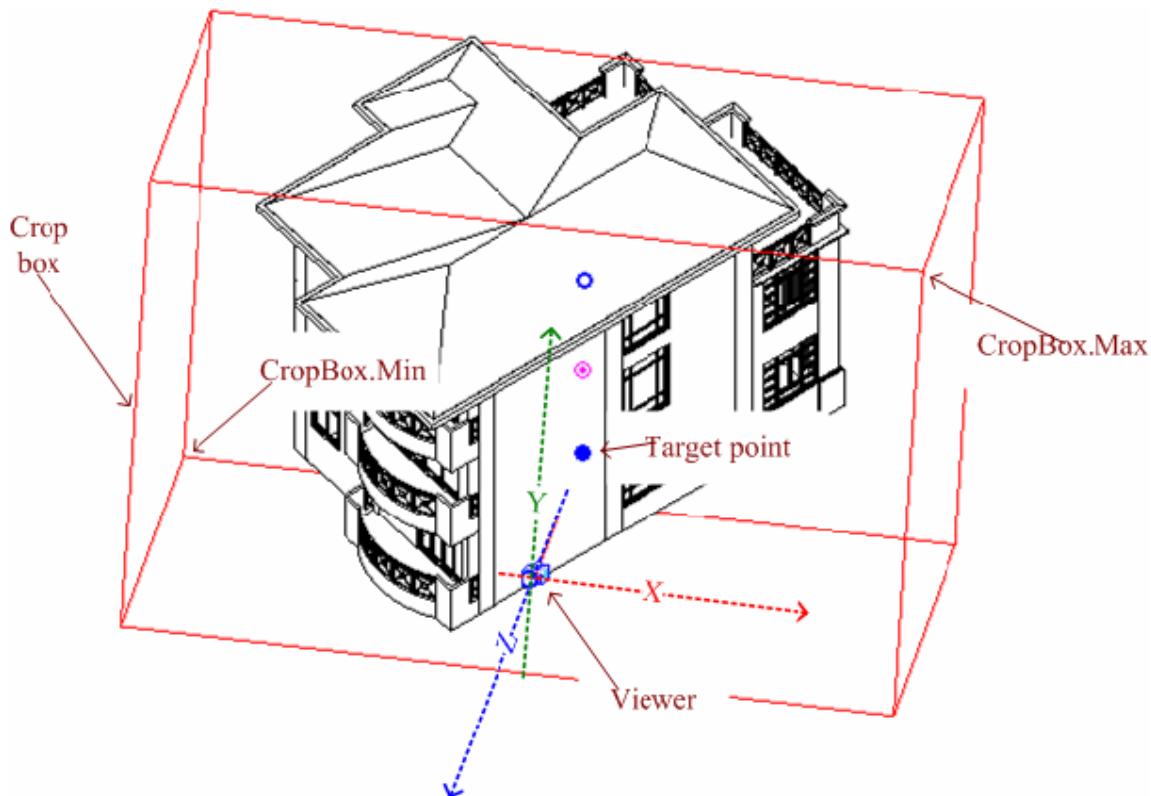
Since the size of an object's perspective projection varies inversely with the distance from that object to the center of the projection, scale is meaningless for perspective views. The perspective 3D view Scale property always returns zero.

Managing the camera target

The camera represents the direction that the viewer of the perspective view is looking. If the user or an API application adjusts the crop region to expose a wider field of view or an unsymmetrical field of view, the distortion of the perspective view can become too drastic. The camera target can be forced to the center of the viewing region by calling the View3D method RestCameraTarget() which positions the target at the center of the field of view. Before calling, check to see if the camera can be reset in this view with the method View3D.CanResetCameraTarget() which indicates whether the camera target may be reset. The main situation where a target cannot be reset is if the View3D is currently in Isometric projection. Attempting to reset the camera target in an isometric view will throw an Autodesk.Revit.Exceptions.InvalidOperationException.

Isometric view

A new isometric view can be created with the static View3D.CreateIsometric() method.

**Figure 98: Parallel projection**

Isometric 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 99: 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:

```
public void ViewScale(View view)
{
    view.CropBox.Max.X(Y) / view.Outlet.Max.X(Y) == view.CropBox.Min.X(Y) / view.Outlet.Min.X(Y) == view.Scale;
}
```

The `viewFamilyTypeId` parameter needs to be a three dimensional `ViewType`. 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. You can also change the default orientation. The API does not support modifying the viewing coordinate system.

The following code sample illustrates how to create an isometric 3D view.

Code Region: Creating an Isometric 3D view

```
public void CreateIso(Document document)
{
    // Find a 3D view type

    IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in new FilteredElementCollector(document).OfClass(typeof(ViewFamilyType))
        let type = elem as ViewFamilyType
        where type.ViewFamily == ViewFamily.ThreeDimensional
        select type;

    // Create a new View3D

    View3D view3D = View3D.CreateIsometric(document, viewFamilyTypes.First().Id);
```

```

if (null != view3D)

{
    // By default, the 3D view uses a default orientation.

    // Change the orientation by creating and setting a ViewOrientation3D

    XYZ eye = new XYZ(10, 10, 10);

    XYZ up = new XYZ(0, 0, 1);

    XYZ forward = new XYZ(0, 1, 0);

    ViewOrientation3D viewOrientation3D = new ViewOrientation3D(eye, up,
forward);

    view3D.SetOrientation(viewOrientation3D);

}
}

```

Switching between isometric and perspective

Most of the time a View3D can be toggled between Isometric and Perspective, provided that there are no view-specific elements in the view. The View3D class provides methods for toggling the view to and from Isometric and Perspective mode. Before toggling, use the `CanToggleBetweenPerspectiveAndIsometric()` method which indicates whether toggling is ok.

To toggle the view call one of the two View3D class methods: `ToggleToPerspective()` or `ToggleToIsometric()`. In the case that the view cannot be toggled (perhaps due to the presence of view specific elements in the view) either of these methods will throw `Autodesk.Revit.Exceptions.InvalidOperationException`.

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 IsSectionBoxActive property. In the example below, if the active view is a 3D view, it sets whether the section box is active.

Code Region: Showing/Hiding the Section Box

```
private void ShowHideSectionBox(UIDocument document, bool active)
{
    if (document.ActiveView is View3D)
    {
        View3D view3d = document.ActiveView as View3D;
        view3d.IsSectionBoxActive = active;
    }
}
```

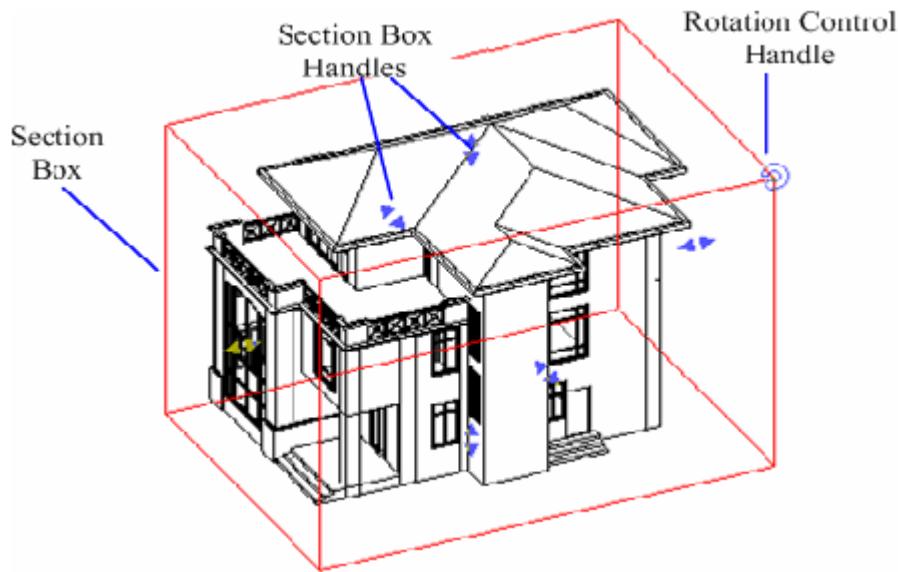


Figure 100: Section box

The View3D.GetSectionBox() and View3D.SetSectionBox() methods are used to get and change the box extents. In some cases, calling View3D.SetSectionBox() can have a side effect.

Setting the property to certain values can change the box capacity and display it in the view. To avoid displaying the section box, set the IsSectionBoxActive property to false.

The following code sample illustrates how to change the extents of the section box.

Code Region: Changing the extents of the section box

```
private void ExpandSectionBox(View3D view)
{
    // The original section box
    BoundingBoxXYZ sectionBox = view.GetSectionBox();

    // Expand the section box (doubling in size in all directions while preserving the same center and orientation)
    Autodesk.Revit.DB.XYZ deltaXYZ = sectionBox.Max - sectionBox.Min;
    sectionBox.Max += deltaXYZ / 2;
    sectionBox.Min -= deltaXYZ / 2;

    //After resetting the section box, it will be shown in the view.
    //It only works when the Section Box is active
    view.SetSectionBox(sectionBox);
}
```

The coordinates of Max and Min points of BoundingBoxXYZ returned from the GetSectionBox() method are not in global coordinates. To convert the coordinates of Max and Min to global coordinates, you need to convert each point via the transform obtained from BoundingBoxXYZ.Transform property.

Code Region: Convert Max and Min to global coordinates

```
private void ConvertMaxMinToGlobal(View3D view, out XYZ max, out XYZ min)
```

```
{
    BoundingBoxXYZ sectionbox = view.GetSectionBox();

    Transform transform = sectionbox.Transform;
    max = transform.OfPoint(sectionbox.Max);
    min = transform.OfPoint(sectionbox.Min);
}
```

View locking

The View3D class has methods and properties corresponding to the locking feature available in the Revit user interface.



The View3D.SaveOrientationAndLock() method will save the orientation and lock the view while View3D.RestoreOrientationAndLock() will restore the view's orientation and lock it. View3D.Unlock() will unlock the view if it is currently locked. The IsLocked property will return whether the 3D view is currently locked.

Grid Visibility

Grid visibility in 3D Views can be accessed with:

- View3D.GetLevelsThatShowGrids()
- View3D.ShowGridsOnLevel(ElementId levelId)
- View3D.HideGridsOnLevel(ElementId levelId)
- View3D.ShowGridsOnLevels(ElementIdset levelIds)

2.6.3.3 ViewPlan

Plan views are level-based. There are three types of plan views: floor plan view, ceiling plan view, and area plan view.

Creating 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. Use the static `ViewPlan.Create()` method to create new floor and ceiling plan views. Use the static `ViewPlan.CreateAreaPlan()` method to create a new area plan view.

Code Region: Creating Plan Views

```
public static ViewPlan ViewPlan.Create(Document document, ElementId viewFamilyTypeId, ElementId levelId);

public static ViewPlan ViewPlan.CreateAreaPlan(Document document, ElementId areaSchemeId, ElementId levelId);
```

The `viewFamilyTypeId` parameter in `ViewPlan.Create()` needs to be a `FloorPlan`, `CeilingPlan`, `AreaPlan`, or `StructuralPlan` `ViewType`. The `levelId` parameter represents the Id of the 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: Creating a floor plan and ceiling plan

```
private void CreateViewPlan(Autodesk.Revit.DB.Document document)

{
    FilteredElementCollector collector = new FilteredElementCollector(document);

    IList<Element> viewFamilyTypes = collector.OfClass(typeof(ViewFamilyType)).ToElements();

    ElementId floorPlanId = ElementId.InvalidElementId;

    foreach (Element e in viewFamilyTypes)
    {
        ViewFamilyType v = e as ViewFamilyType;
```

```
if (v != null && v.ViewFamily == ViewFamily.FloorPlan)

{
    floorPlanId = e.Id;
    break;
}

}

ElementId ceilingPlanId = ElementId.InvalidElementId;

foreach (Element e in viewFamilyTypes)

{
    if (e.Name == "Ceiling Plan")

    {
        ceilingPlanId = e.Id;
        break;
    }
}

// Create a Level and a Floor Plan based on it

double elevation = 10.0;

Level level1 = Level.Create(document, elevation);

ViewPlan floorView = ViewPlan.Create(document, floorPlanId, level1.Id);

// Create another Level and a Ceiling Plan based on it

elevation += 10.0;

Level level2 = Level.Create(document, elevation);

ViewPlan ceilingView = ViewPlan.Create(document, ceilingPlanId, level2.Id);
```

{}

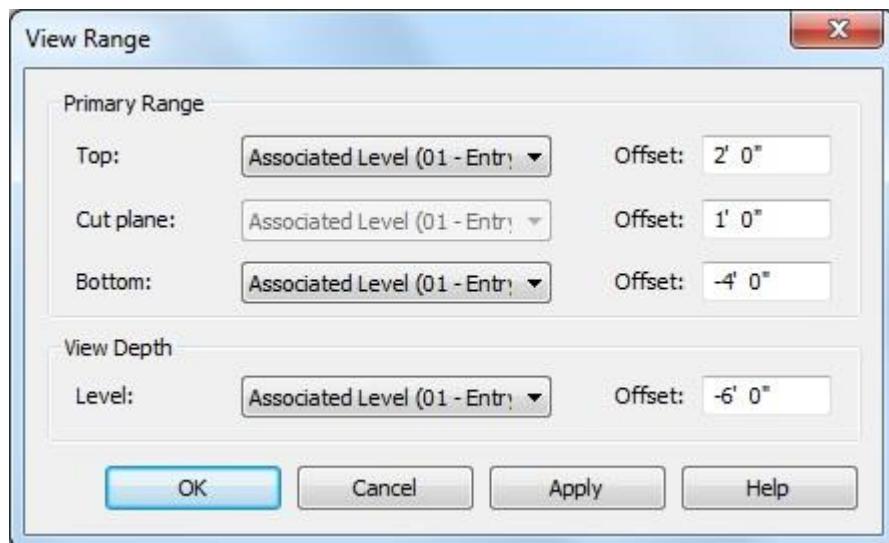
Plan view properties

After creating a new plan view, the Discipline for the view can be set using the Discipline parameter which is type ViewDiscipline. Options include Architectural, Structural, Mechanical, Electrical, Plumbing and Coordination.

For structural plan views, the view direction can be set to either Up or Down using the ViewFamilyType.PlanViewDirection property. Although it is a property of the ViewFamilyType class, an exception will be thrown if the property is accessed for views other than StructuralPlan views.

View range

The view range for plan views can be retrieved via the ViewPlan.GetViewRange() method. The returned PlanViewRange object can be used to find the levels which a plane is relative to and the offset of each plane from that level. It is the same information that is available in the View Range dialog in the Revit user interface:



The following example shows how to get the top clip plane and the associated offset for a plan view

Code Region: Getting information on the view range

```
private void ViewRange(Document doc, View view)
{

```

```

if (view is ViewPlan)
{
    ViewPlan viewPlan = view as ViewPlan;
    PlanViewRange viewRange = viewPlan.GetViewRange();

    ElementId topClipPlane = viewRange.GetLevelId(PlanViewPlane.TopClipPlane);
    double dOffset = viewRange.GetOffset(PlanViewPlane.TopClipPlane);

    if (topClipPlane.Value > 0)
    {
        Element levelAbove = doc.GetElement(topClipPlane);
        TaskDialog.Show(view.Name, "Top Clip Plane: " + levelAbove.Name +
"\r\nTop Offset: " + dOffset + " ft");
    }
}
}

```

Plan view underlay

The top and base levels of the underlay range can be retrieved and set from ViewPlan. Use GetUnderlayBaseLevel() and SetUnderlayBaseLevel() to access the base level for the underlay range. If the base level id is InvalidElementId then the underlay base level is not set and no elements will be displayed as underlay. When setting the base level for the underlay range, the elevation of the next highest level will be used to determine the top of the underlay range. If the level specified for the base level is the highest level, the underlay range will be unbounded and will consist of everything above the specified level.

Use GetUnderlayTopLevel() and SetUnderlayRange() to access the top level for the underlay range. If GetUnderlayTopLevel() returns InvalidElementId and the underlay base level is a valid level, then the underlay range is unbounded, and consists of everything above the underlay base level. To set the top level, you must use SetUnderlayRange() which takes ElementIds for both the base and top levels. This method will throw an exception if the elevation of the top level is not greater than the elevation of the base level.

Use the GetUnderlayOrientation() and SetUnderlayOrientation() methods to control how elements in the underlay are viewed. The UnderlayOrientation is either LookingDown (underlay

elements are viewed as if looking down from above) or LookingUp (underlay elements are viewed as if looking up from below).

The following code sets the underlay range if the current orientation is LookingDown and the top level Id is different than the new value. Then the orientation is changed to LookingUp.

Code Region: Change the view underlay range

```
private void ViewUnderlay(ViewPlan planView, ElementId topLevelId, ElementId baseLevelId)
{
    if (planView.GetUnderlayOrientation() == UnderlayOrientation.LookingDown)
    {
        if (planView.GetUnderlayTopLevel() != topLevelId)
        {
            planView.SetUnderlayRange(baseLevelId, topLevelId);
        }
    }

    planView.SetUnderlayOrientation(UnderlayOrientation.LookingUp);
}
```

2.6.3.4 *ViewDrafting*

Views to create unassociated, view-specific details that are not part of the modeled design.

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:

- | | |
|--|--|
| <ul style="list-style-type: none"> ○ Detail lines | <ul style="list-style-type: none"> ○ Reference planes |
|--|--|

- Detail regions
- Detail components
- Insulation
- Dimensions
- Symbols
- Text

These tools are the same tools used to create a detail view.

- Drafting views do not display model elements.

Use the static `ViewDrafting.Create()` method to create a drafting view. Model elements are not displayed in the drafting view.

ImageView

The `ImageView` class is derived from `ViewDrafting`. It can be used to create rendering views containing images imported from disk. Use the static `ImageView.Create(Document, ImageTypeOptions)` method to create new rendering views.



2.6.3.5 ViewSection

Represents section, detail, callout and elevation views, as well as reference callouts and reference sections.

The `ViewSection` class can be used to create section views, detail views, callout views, reference callouts and reference sections. It also represents elevation views.

Section Views and Reference Sections

Section views cut through the model to expose the interior structure.

The `ViewSection.CreateSection()` method creates the section view.

The `viewFamilyTypeId` parameter is the Id for the `ViewFamilyType` which will be used by the new `ViewSection`. The type needs to be a `Section ViewFamily`. The `sectionBox` parameter is the section view crop box. It provides the direction and extents 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 direction and extents.

The following code shows how to create a section view. A bounding box for the section view is created at the center of a wall. The resulting section view will be located in the Sections (Building Section) node in the Project Browser. Note that the far clip distance will be equal to the difference of the z-coordinates of bounding box's min and max values on creation.

Code Region: Creating a section view

```
public void CreateSection(Wall wall)
{
    Document document = wall.Document;

    // Find a section view type

    IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in new FilteredElementCollector(document).OfClass(typeof(ViewFamilyType))
        let type = elem as ViewFamilyType
        where type.ViewFamily == ViewFamily.Section
        select type;

    // Create a BoundingBoxXYZ instance centered on wall

    LocationCurve lc = wall.Location as LocationCurve;
    Transform curveTransform = lc.Curve.ComputeDerivatives(0.5, true);
    // using 0.5 and "true" (to specify that the parameter is normalized)
    // places the transform's origin at the center of the location curve

    XYZ origin = curveTransform.Origin; // mid-point of location curve
    XYZ viewDirection = curveTransform.BasisX.Normalize(); // tangent vector along the location curve
    XYZ normal = viewDirection.CrossProduct(XYZ.BasisZ).Normalize(); // location curve normal @ mid-point

    Transform transform = Transform.Identity;
```

```
transform.Origin = origin;
transform.BasisX = normal;
transform.BasisY = XYZ.BasisZ;

// can use this simplification because wall's "up" is vertical.

// For a non-vertical situation (such as section through a sloped floor the surface normal would be needed)

transform.BasisZ = normal.CrossProduct(XYZ.BasisZ);

BoundingBoxXYZ sectionBox = new BoundingBoxXYZ();
sectionBox.Transform = transform;
sectionBox.Min = new XYZ(-10,0,0);
sectionBox.Max = new XYZ(10,12,5);

// Min & Max X values (-10 & 10) define the section line length on each side of the wall

// Max Y (12) is the height of the section box// Max Z (5) is the far clip offset

// Create a new view section.

ViewSection viewSection = ViewSection.CreateSection(document, viewFamilyTypes.First().Id, sectionBox);
}
```

Reference sections are sections that reference an existing view. Revit does not add a new view when you create a new reference section.

Code Region: ViewSection.CreateReferenceSection()

```
public ViewSection ViewSection.CreateReferenceSection(Document document, ElementId parentViewId, ElementId viewIdToReference, XYZ headPoint, XYZ tailPoint);
```

The parentViewId parameter is the Id of the view in which the new reference section marker will appear. Reference sections can be created in FloorPlan, CeilingPlan, StructuralPlan, Section, Elevation, Drafting, and Detail views. The viewIdToReference can be the Id of a Detail, Drafting or Section view. The ViewFamilyType of the referenced view will be used by the new reference section. The two XYZ points will determine the location of the section marker's head in the parent view.

Detail Views

A detail view is a view of the model that appears as a callout or section in other views. This type of view typically represents the model at finer scales of detail than in the parent view. It is used to add more information to specific parts of the model. The static ViewSection.CreateDetail() method is used to create a new detail ViewSection.

Code Region: ViewSection.CreateDetail()

```
public ViewSection ViewSection.CreateDetail(Document document, ElementId viewFamilyTypeId, BoundingBoxXYZ sectionBox);
```

The viewFamilyTypeId parameter is the Id for the ViewFamilyType which will be used by the new ViewSection. The type needs to be a Detail ViewFamily. Just as for a standard section view, the sectionBox parameter is the section view crop box. It provides the direction and extents which are required for the section view.

When a new detail ViewSection is added, it will appear in the Detail Views (Detail) node in the Project Browser.

Elevation Views

An elevation view is a cross-section of the model where level lines are displayed. An elevation view is represented by the ViewSection class. However, unlike the other types of section views, you cannot create elevation views using a static method on the ViewSection class. To create an elevation view, first create an elevation marker, then use the marker to generate the elevation view. The newly created elevation view will appear in the Elevations (Building Elevation) node in the Project Browser. It will be assigned a unique name.

The following example creates an elevation view based on the location of a beam.

Code Region: Creating an Elevation View

```

ViewSection CreateElevationView(Document document, FamilyInstance beam)
{
    // Find an elevation view type

    IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in new FilteredElementCollector(document).OfClass(typeof(ViewFamilyType))
                                                    let type = elem as ViewFamilyType
                                                    where type.ViewFamily == ViewFamily.Elevation
                                                    select type;

    LocationCurve lc = beam.Location as LocationCurve;
    XYZ xyz = lc.Curve.GetEndPoint(0);

    ElevationMarker marker = ElevationMarker.CreateElevationMarker(document,
        viewFamilyTypes.First().Id, xyz, 1);

    ViewSection elevationView = marker.CreateElevation(document, document.ActiveView.Id, 1);

    return elevationView;
}

```

The `ElevationMarker.CreateElevation()` method takes an id of a `ViewPlan` as a parameter. That is the `ViewPlan` in which the `ElevationMarker` is visible. The new elevation `ViewSection` will derive its extents and inherit settings from the `ViewPlan`. The last parameter is the index on the `ElevationMarker` where the new elevation view will be placed. The index on the `ElevationMarker` must be valid and unused. The view's direction is determined by the index.

Split Sections

These methods identify if a section view is split and return the offset for the specified split crop region:

- `DBViewSection.IsSplitSection()`

ViewCropRegionShapeManager.GetSplitRegionOffset()

Callouts and Reference Callouts

A callout shows part of another view at a larger scale. Callout views can be created using the static method `ViewSection.CreateCallout()`. Callouts can be created in FloorPlan, CeilingPlan, StructuralPlan, Section, Elevation, Drafting and Detail views. The resulting view will be either a `ViewSection`, `ViewPlan` or `ViewDetail` depending on the `ViewFamilyType` used and will appear in the corresponding node in the Project Browser.

- `View.GetCalloutParentId()` returns the ID of a view which this callout references, or `InvalidElementId` if there is not parent.
- `View.IsCallout` identifies if the view is a callout view

Code Region: ViewSection.CreateCallout()

```
public ViewSection ViewSection.CreateCallout(Document document,
                                             ElementId parentViewId,
                                             ElementId viewFamilyTypeId,
                                             XYZ point1,
                                             XYZ point2);
```

The parent view Id parameter can be the Id of any type of View on which callouts can be created. The point parameters determine the extents of the callout symbol in the parent view. A reference callout is a callout that refers to an existing view. When you add a reference callout, Revit does not create a view in the project. Instead, it creates a pointer to a specified, existing view. Multiple reference callouts can point to the same view.

Code Region: ViewSection.CreateReferenceCallout()

```
public ViewSection ViewSection.CreateReferenceCallout(Document document,
                                                      ElementId parentViewId,
                                                      ElementId viewIdToRefer-
                                                      ence,
```

```
XYZ point1,
XYZ point2);
```

Creation of a reference callout is similar to creation of a callout. But rather than having the Id of the ViewFamilyType for the callout as a parameter, the CreateReferenceCallout() method takes the Id of the view to reference. The ViewFamilyType of the referenced view will be used by the new reference callout.

Only cropped views can be referenced, unless the referenced view is a Drafting view. Drafting views can always be referenced regardless of the parent view type. Elevation views can be referenced from Elevation and Drafting parent views. Section views can be referenced from Section and Drafting parent views. Detail views can be referenced from all parent views except for in FloorPlan, CeilingPlan and StructuralPlan parent views where only horizontally-oriented Detail views can be referenced. FloorPlan, CeilingPlan and StructuralPlan views can be referenced from FloorPlan, CeilingPlan and StructuralPlan parent views.

The following example creates a new callout using a Detail ViewFamilyType and then uses the new callout view to create a reference callout.

Code Region: Creating a callout and reference callout

```
public void CreateCalloutView(Document document, View parentView)
{
    // Find a detail view type
    IEnumerable<ViewFamilyType> viewFamilyTypes = from elem in new FilteredElementCollector(document).OfClass(typeof(ViewFamilyType))
                                                    let type = elem as ViewFamilyType
                                                    where type.ViewFamily == ViewFamily.Detail
                                                    select type;

    ElementId viewFamilyTypeId = viewFamilyTypes.First().Id;
    XYZ point1 = new XYZ(2, 2, 2);
    XYZ point2 = new XYZ(30, 30, 30);
```

```

ElementId parentViewId = parentView.Id; // a ViewPlan

View view = ViewSection.CreateCallout(document, parentViewId, viewFamilyType,
    point1, point2);

ViewSection.CreateReferenceCallout(document, parentViewId, view.Id, point1, point2);

}

```

2.6.3.6 ViewSheet

A sheet contains views and a title block. When creating a sheet view with the `ViewSheet.Create()` method, a title block family symbol Id is a required parameter for the method. A title block family symbol can be found using a `FilteredElementCollector`.

Code Region: `ViewSheet.Create()`

```

public static ViewSheet ViewSheet.Create(Document document, ElementId titleBlockTypeId);

```

The newly created sheet has no views. The `Viewport.Create()` method is used to add views. The `Viewport` class is used to add regular views to a view sheet, i.e. plan, elevation, drafting and three dimensional. To add schedules to a view, use `ScheduleSheetInstance.Create()` instead.

The `View.GetPlacementOnSheetStatus` method returns a `ViewPlacementOnSheetStatus` enum that describes if the view is placed on a sheet. Some views, such as schedules, can be partially placed on a sheet by divided them into segments and placing only some of those segments on a sheet.

The property `Viewport.LabelOffset` controls the two-dimensional label offset from left bottom corner of the viewport (as established with `Rotation` set to `None`) to the left end of the viewport label line. The property `Viewport.LabelLineLength` controls the length of the viewport label line in sheet space.

Code Region: Add two views aligned at left corner

```

public static void PlaceAlignedViewsAtLeftCorner(Document doc)

```

```
{  
  
    FilteredElementCollector fec = new FilteredElementCollector(doc);  
  
    fec.OfClass(typeof(ViewPlan));  
  
    var viewPlans = fec.Cast<ViewPlan>().Where<ViewPlan>(vp => !vp.IsTemplate  
&& vp.ViewType == ViewType.CeilingPlan);  
  
  
    ViewPlan vp1 = viewPlans.ElementAt(0);  
  
    ViewPlan vp2 = viewPlans.ElementAt(1);  
  
  
    using (Transaction t = new Transaction(doc, "Place on sheet"))  
    {  
  
        t.Start();  
  
  
        // Add two viewports distinct from one another  
  
        ViewSheet vs = ViewSheet.Create(doc, ElementId.InvalidElementId);  
  
        Viewport viewport1 = Viewport.Create(doc, vs.Id, vp1.Id, new XYZ(0,  
0, 0));  
  
        Viewport viewport2 = Viewport.Create(doc, vs.Id, vp2.Id, new XYZ(0,  
5, 0));  
  
  
        doc.Regenerate();  
  
  
        // Calculate the necessary move vector to align the lower left corner  
        Outline outline1 = viewport1.GetBoxOutline();  
  
        Outline outline2 = viewport2.GetBoxOutline();  
  
        XYZ boxCenter = viewport2.GetBoxCenter();  
  
        XYZ vectorToCenter = boxCenter - outline2.MinimumPoint;  
  
        XYZ newCenter = outline1.MinimumPoint + vectorToCenter;  
    }  
}
```

```

    // Move the viewport to the new location
    viewport2.SetBoxCenter(newCenter);

    t.Commit();

}
}

```

- The XYZ 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.

Viewports placed on sheets can have the associated view swapped for another view in the model by editing the `Viewport.ViewId` property. When this swap is done, the `Viewport.ViewportPositioning` property specifies how the viewport will be positioned on the sheet when swapped to another view by maintaining either the viewport center or the view origin.

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 (using a filter in this case) and use it to create the sheet view. Next, add a 3D view. The view is placed with its lower left-hand corner at the center of the sheet. Finally, print the sheet by calling the `View.Print()` method.

Code Region: Creating a sheet view

```

private void CreateSheetView(Autodesk.Revit.DB.Document document, View3D view
3D)
{
    // Get an available title block from document
}

```

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

    collector.OfClass(typeof(FamilySymbol));
    collector.OfCategory(BuiltInCategory.OST_TitleBlocks);

    FamilySymbol fs = collector.FirstElement() as FamilySymbol;
    if (fs != null)
    {
        using (Transaction t = new Transaction(document, "Create a new ViewSheet"))
        {
            t.Start();
            try
            {
                // Create a sheet view
                ViewSheet viewSheet = ViewSheet.Create(document, fs.Id);
                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);
            }
        }
    }
}
```

```
        Viewport.Create(document, viewSheet.Id, view3D.Id, new XYZ(location.U, location.V, 0));  
  
        // 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();  
            }  
        }  
  
        t.Commit();  
    }  
    catch  
    {  
        t.Rollback();  
    }  
}
```

{}

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.

Duplicating a sheet

When duplicating a sheet, the `SheetDuplicateOption` enum allows you to indicate what information should be copied when duplicating a sheet. Its values are:

- `DuplicateEmptySheet` - Only copies the title block.
- `DuplicateSheetWithDetailing` - Copies the title block and details.
- `DuplicateSheetWithViewsOnly` - Copies the title block, details, viewports and contained views. The newly created sheet will reference the newly duplicated views.
- `DuplicateSheetWithViewsAndDetailing` - Copies the title block, details, and viewports. Duplicates the sheet's contained views with detailing. The newly created sheet will reference the newly duplicated views.
- `DuplicateSheetWithViewsAsDependent` - Copies the title block, details, and viewports. Duplicates the sheet's contained views as dependent. The newly created sheet will reference the newly duplicated dependent views.

Revisions on Sheets

The `ViewSheet` class has several methods for working with revision and revision clouds on sheets.

- **GetAllRevisionIds()**- Gets the ordered array of Revisions which participate in the sheet's revision schedules.
- **GetAdditionalRevisionIds()**- Gets the Revisions that are additionally included in the sheet's revision schedules.
- **SetAdditionalRevisionIds()**- Sets the Revisions to additionally include in the sheet's revision schedules.
- **GetCurrentRevision()**- Returns the most recent numbered Revision shown on this `ViewSheet`.
- **GetRevisionCloudNumberOnSheet()**- Gets the Revision Number for a `RevisionCloud` on this sheet when the numbering in the project is by sheet.
- **GetRevisionNumberOnSheet()** - Gets the Revision Number for a particular Revision as it will appear on this sheet when the numbering in the project is by sheet. The Revisions are ordered according to the revision sequence in the project. Additionally included Revisions will always participate in the sheet's revision schedules. Normally a Revision is scheduled in the revision schedule because one of its associated `RevisionClouds` is present on the sheet.

The following code sample demonstrates how to add additional revisions to the sheet that match a given criteria.

Code Region: Add additional revisions to sheet

```
public static void AddAdditionalRevisionsToSheet(ViewSheet viewSheet, String  
toMatch)  
{  
    Document doc = viewSheet.Document;  
  
    ICollection<ElementId> revisions = viewSheet.GetAdditionalRevisionIds();  
  
    // Find revisions whose description matches input string  
    FilteredElementCollector collector = new FilteredElementCollector(doc);  
    collector.OfCategory(BuiltInCategory.OST_Revisions);  
    collector.WhereElementIsNotElementType();  
    if (revisions.Count > 0)  
        collector.Excluding(revisions);  
  
    // Check if revision should be added  
    foreach (Element revision in collector)  
    {  
        Parameter descriptionParam = revision.GetParameter(ParameterTypeId.Pr  
ojectRevisionDescription);  
        String description = descriptionParam.AsString();  
        if (description.Contains(toMatch))  
            revisions.Add(revision.Id);  
    }  
  
    if (revisions.Count > 0)
```

```

    {

        // Apply the new list of revisions

        using (Transaction t = new Transaction(doc, "Add revisions to sheet
"))
        {

            t.Start();

            viewSheet.SetAdditionalRevisionIds(revisions);

            t.Commit();
        }
    }
}

```

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.

Sheet Collections

The class `Autodesk.Revit.DB.SheetCollection` represents a sheet collection in Autodesk Revit.

- `Autodesk.Revit.DB.SheetCollection.Create(document, name)` Creates a new instance of sheet collection with a **specified name** and adds it to the document. Returns the newly created sheet collection element.
- `Autodesk.Revit.DB.SheetCollection.Create(document)` Creates a new instance of sheet collection with an **auto-generated name** and adds it to the document. Returns the newly created sheet collection element.
- `Autodesk.Revit.DB.ViewSheet.SheetCollectionId` represents the Id of the sheet collection that a sheet is associated with.

2.6.3.7 *TableView*

This class represents a view that shows a table of data.

TableView is the base class for ViewSchedule and PanelScheduleView.

2.6.3.7.1 Schedule Classes

Schedule views use several supporting classes.

TableView is a class that represents a view that shows a table and it is the base class for ViewSchedule and PanelScheduleView. It has an associated TableData class which contains one or more sections. For ViewSchedule, there is only one header and one body section.

The TableSectionData class represents a contiguous set of cells arranged in rows and columns. For a ViewSchedule the cell contents of TableSectionData are generated from the ScheduleDefinition and parameters. Also, for ViewSchedules, while the header section has read/write permissions, the body section is read-only.

Working with data in a schedule

The actual data for a table is contained in the `TableData` class. Although the `TableData` object cannot be obtained directly from the `TableView` class, both child classes have a `GetTableData()` method. For `ViewSchedule`, this method returns a `TableData` object. For a `PanelScheduleView`, `GetTableData()` returns a `PanelScheduleData` object, which derives from the `TableData` base class. The `TableData` class holds most of the data that describe the style of the rows, columns, and cells in a table. `PanelScheduleData` provides additional methods related specifically to panel schedules.

Working with rows, columns and cells

Data in a table is broken down into sections. To work with the rows, columns and cells of the `TableData`, it is necessary to get the a `TableSectionData` object. `TableData.GetSectionData()` can be called with either an integer to the requested section data, or using the `SectionType` (i.e. Header or Body).

The `TableSectionData` class can be used to insert or remove rows or columns, format cells, and to get details of the cells that make up that section of the schedule, such as the cell type (i.e. Text or Graphic) or the cell's category id.

In the following example, a new row is added to the header section of the schedule and the text is set for the newly created cell. Note that the first row of the header section defaults to the title when created with the UI.

Code Region: Inserting a row

```
public void CreateSubtitle(ViewSchedule schedule)
{
    TableData colTableData = schedule.GetTableData();

    TableSectionData tsd = colTableData.GetSectionData(SectionType.Header);
    tsd.InsertRow(tsd.FirstRowNumber + 1);
    tsd.SetCellText(tsd.FirstRowNumber + 1, tsd.FirstColumnNumber, "Schedule
of column top and base levels with offsets");
}
```

Also note, in the code example above, it uses the properties `FirstRowNumber` and `FirstColumnNumber`. In some sections the row or column numbers might start with 0, or they might start with 1. These properties should always be used in place of a hardcoded 0 or 1.

In the following example, a new single-category schedule is created with a custom header section.

Code Region: Customizing the header section

```
public static void CreateSingleCategoryScheduleWithSimpleHeaderSection(Document doc)
{
    using (Transaction t = new Transaction(doc, "Create single-category with
custom headers"))

    {
        // Build schedule

        t.Start();

        ViewSchedule vs = ViewSchedule.CreateSchedule(doc, new ElementId(Built
InCategory.OST_Windows));

        AddRegularFieldToSchedule(vs, new ElementId(BuiltInParameter.WINDOW_H
EIGHT));
        AddRegularFieldToSchedule(vs, new ElementId(BuiltInParameter.WINDOW_W
IDTH));
        AddRegularFieldToSchedule(vs, new ElementId(BuiltInParameter.ALL_MODE
L_MARK));
        AddRegularFieldToSchedule(vs, new ElementId(BuiltInParameter.ALL_MODE
L_COST));

        doc.Regenerate();

        // Get header section

        TableSectionData data = vs.GetTableData().GetSectionData(SectionType.
Header);

        int rowCount = data.RowCount;
        int columnCount = data.ColumnCount;
    }
}
```

```
// Get the overall width of the table so that the new columns can be
// resized properly

double tableWidth = data.GetColumnWidth(columnNumber);

data.InsertColumn(columnNumber);
data.InsertColumn(columnNumber);

// Refresh data to be sure that schedule is ready for text insertion
vs.RefreshData();

//Set text to the first header cell

data.SetCellText(rowNumber, data.FirstColumnNumber, "Special Window S
chedule Text");

// Set width of first column

data.SetColumnWidth(data.FirstColumnNumber, tableWidth / 3.0);

//Set a different parameter to the second cell - the project name

data.SetCellParamIdAndCategoryId(rowNumber, data.FirstRowNumber + 1,
new ElementId(BuiltInParameter.PROJECT_NAME),
new ElementId(BuiltInCategory.OST
_ProjectInformation));

data.SetColumnWidth(data.FirstColumnNumber + 1, tableWidth / 3.0);

//Set the third column as the schedule view name - use the special ca
tegory for schedule parameters for this

data.SetCellParamIdAndCategoryId(rowNumber, data.LastColumnNumber, ne
w ElementId(BuiltInParameter.VIEW_NAME),
```

```
new ElementId(BuiltInCategory.OST  
_ScheduleViewParamGroup));  
  
data.SetColumnWidth(data.LastColumnNumber, tableWidth / 3.0);  
  
t.Commit();  
}  
}  
  
public static void AddRegularFieldToSchedule(ViewSchedule schedule, ElementId  
paramId)  
{  
    ScheduleDefinition definition = schedule.Definition;  
  
    // Find a matching SchedulableField  
    SchedulableField schedulableField =  
        definition.GetSchedulableFields().FirstOrDefault<SchedulableField>(sf  
=> sf.ParameterId == paramId);  
  
    if (schedulableField != null)  
    {  
        // Add the found field  
        definition.AddField(schedulableField);  
    }  
}
```

The style of rows, columns, or individual cells can be customized for schedules. This includes the ability to set the border line style for all four sides of cells, as well as cell color and text appearance (i.e. color, font, size). For regular schedules, this can only be done in the header section of the table.

In the example below, the font of the subtitle of a ViewSchedule (assumed to be the second row of the header section) is set to bold and the font size is set to 10.

Code Region: Formatting cells

```
public void FormatSubtitle(ViewSchedule colSchedule)
{
    TableData colTableData = colSchedule.GetTableData();

    TableSectionData tsd = colTableData.GetSectionData(SectionType.Header);
    // Subtitle is second row, first column
    if (tsd.AllowOverrideCellStyle(tsd.FirstRowNumber + 1, tsd.FirstColumnNumber))
    {
        TableCellStyle tcs = new TableCellStyle();
        TableCellStyleOverrideOptions options = new TableCellStyleOverrideOptions();
        options.FontSize = true;
        options.Bold = true;
        tcs.SetCellStyleOverrideOptions(options);
        tcs.IsFontBold = true;
        tcs.TextSize = 10;
        tsd.SetCellStyle(tsd.FirstRowNumber + 1, tsd.FirstColumnNumber, tcs);
    }
}
```

2.6.3.7.2 ViewSchedule

A schedule is a tabular representation of data. A typical schedule shows all elements of a category (doors, rooms, etc.) with each row representing an element and each column representing a parameter.

The ViewSchedule class represents schedules and other schedule-like views, including single-category and multi-category schedules, key schedules, material takeoffs, view lists, sheet lists, keynote legends, revision schedules, and note blocks.

The ViewSchedule.Export() method will export the schedule data to a text file.

TableData.ZoomLevel allows the user to set the zoom level of a schedule in a tabular view. This setting will not change the size of text, rows, or columns in a sheet view. Additionally, the setting is temporary and only applies to the current session.

Placing Schedules on Sheets

The static ScheduleSheetInstance.Create() method creates an instance of a schedule on a sheet. It requires the ID of the sheet where the schedule will be placed, the ID of the schedule view, and the XYZ location on the sheet where the schedule will be placed. The ScheduleSheetInstance object has properties to access the ID of the "primary" schedule that generates this ScheduleSheetInstance, the rotation of the schedule on the sheet, the location on the sheet where the schedule is placed (in sheet coordinates), as well as a flag that identifies if the ScheduleSheetInstance is a revision schedule in a titleblock family.

The `GetScheduleHeightsOnSheet` method returns the height of the column headers, row heights, and schedule title.

Striped rows

These properties provide access to read or set if a given schedule is using a striped row display, and whether that display will be used on a sheet that displays this schedule:

- `ViewSchedule.HasStripedRows`
- `ViewSchedule.UseStripedRowsOnSheets`

These methods get and set the color applied to the indicated part of the pattern for a schedule with striped rows:

- `ViewSchedule.GetStripedRowsColor()`
- `ViewSchedule.SetStripedRowsColor()`

Row Height

`ViewSchedule.RowHeightOverride` allows users to define the override that is applied to the row height. `ViewSchedule.RowHeight` Allows users to define the schedule body rows height. The property is applied only for when the schedule is placed on sheet as a ScheduleSheetInstance. The new enum:

`Autodesk.Revit.DB.RowHeightOverrideOptions` describes the options for overriding schedule body row heights. It has the following values

- None - No override would be applied for the row height.
- All - Override row height for any body rows in the schedule.

- `ImageRows` - Override row height for body rows containing images in the schedule.

Frozen header

`ViewSchedule.IsHeaderFrozen` provides access to read or set if the header is frozen on a given schedule.

Split Schedules

Schedules can be split, and the resulting segments can be split, merged, and deleted. An overload of `ScheduleSheetInstance.Create` exists to place a schedule segment on a sheet.

Filtering By Sheet

With the `ScheduleDefinition.IsFilteredBySheet` property, you can define if the `ScheduleSheetInstance` will list only the elements visible in the viewports on that sheet.

2.6.3.7.2.1 Creating a Schedule

The `ViewSchedule` class has several methods for creating new schedules depending on the type of schedule. All of the methods have a `Document` parameter that is the document to which the new schedule or schedule-like view will be added. The newly created schedule views will appear under the `Schedules/Quantities` node in the Project Browser.

A standard single-category or multi-category schedule can be created with the static `ViewSchedule.CreateSchedule()` method.

Code Region: Create a single-category schedule with 2 fields

```
public static void CreateSingleCategorySchedule(Document doc)
{
    using (Transaction t = new Transaction(doc, "Create single-category"))
    {
        t.Start();

        // Create schedule
        ViewSchedule vs = ViewSchedule.CreateSchedule(doc, new ElementId(BuiltInCategory.OST_Windows));

        doc.Regenerate();
    }
}
```

```
// Add fields to the schedule
AddRegularFieldToSchedule(vs, new ElementId(BuiltInParameter.WINDOW_HEIGHT));
AddRegularFieldToSchedule(vs, new ElementId(BuiltInParameter.WINDOW_WIDTH));

t.Commit();

}

}

/// <summary>
/// Adds a single parameter field to the schedule
/// </summary>
public static void AddRegularFieldToSchedule(ViewSchedule schedule, ElementId paramId)
{
    ScheduleDefinition definition = schedule.Definition;

    // Find a matching SchedulableField
    SchedulableField schedulableField =
        definition.GetSchedulableFields().FirstOrDefault<SchedulableField>(sf => sf.ParameterId == paramId);

    if (schedulableField != null)
    {
        // Add the found field
        definition.AddField(schedulableField);
```

```

    }
}
```

The second parameters the ID of the category whose elements will be included in the schedule, or InvalidElementId for a multi-category schedule.

A second CreateSchedule() method can be used to create an area schedule and takes an additional parameter that is the ID of an area scheme for the schedule.

Code Region: Creating an area schedule

```

public void CreateAreaSchedule(Document doc)
{
    FilteredElementCollector collector = new FilteredElementCollector(doc);
    collector.OfCategory(BuiltInCategory.OST_AreaSchemes);

    //Get first ElementId of AreaScheme.
    ElementId areaSchemeId = collector.FirstElementId();

    if (areaSchemeId != null && areaSchemeId != ElementId.InvalidElementId)
    {
        // If you want to create an area schedule, you must use CreateSchedule
        // method with three arguments.

        // The value of the second argument must be ElementId of BuiltInCate
        // gory.OST_Areas category

        // and the value of third argument must be ElementId of an AreaSchem
        // e.

        ViewSchedule areaSchedule = Autodesk.Revit.DB.ViewSchedule.CreateSche
        dule(doc, new ElementId(BuiltInCategory.OST_Areas), areaSchemeId);
    }
}
```

A key schedule displays abstract "key" elements that can be used to populate parameters of ordinary model elements and can be created with the static ViewSchedule.CreateKeySchedule() method whose second parameter is the ID of the category

of elements with which the schedule's keys will be associated. A material takeoff is a schedule that displays information about the materials that make up elements in the model. Unlike regular schedules where each row (before grouping) represents a single element, each row in a material takeoff represents a single <element, material> pair. The `ViewSchedule.CreateMaterialTakeoff()` method has the same parameters as the `ViewSchedule.CreateSchedule()` method and allows for both single- and multi-category material takeoff schedules.

View lists, sheet lists, and keynote legends are associated with a designated category and therefore their creation methods do take a category ID as a parameter. A view list is a schedule of views in the project. It is a schedule of the Views category and is created using `ViewSchedule.CreateViewList()`.

A sheet list is a schedule of sheets in the project. It is a schedule of the Sheets category and is created using the `ViewSchedule.CreateSheetList()` method.

A keynote legend is a schedule of the Keynote Tags category and is created using `ViewSchedule.CreateKeynoteLegend()`.

Revision schedules are added to titleblock families and become visible as part of titleblocks on sheets. The `ViewSchedule.CreateRevisionSchedule()` method will throw an exception if the document passed in is not a titleblock family.

A note block is a schedule of the Generic Annotations category that shows elements of a single family rather than all elements in a category.

The second parameter is the ID of the family whose elements will be included in the schedule.

Code Region: Creating a note block schedule

```
public void CreateNoteBlock(Document doc)
{
    using (Transaction transaction = new Transaction(doc, "Creating Note Block"))
    {
        //Get first ElementId of a Note Block family.
        ICollection<ElementId> noteblockFamilies = ViewSchedule.GetValidFamiliesForNoteBlock(doc);

        ElementId symbolId = noteblockFamilies.First<ElementId>();

        ViewSchedule noteBlockSchedule = null;

        if (!symbolId.Equals(ElementId.InvalidElementId))
```

```
{  
    transaction.Start();  
  
    //Create a note-block view schedule.  
    noteBlockSchedule = ViewSchedule.CreateNoteBlock(doc, symbolId);  
}  
  
if (null != noteBlockSchedule)  
{  
    transaction.Commit();  
}  
else  
{  
    transaction.Rollback();  
}  
}  
}  
}
```

2.6.3.7.2.2 Working with ViewSchedule

The ScheduleDefinition class helps define the ViewSchedule.

The ScheduleDefinition class contains various settings related to the contents of a schedule view, including:

- The schedule's category and other basic properties that determine the type of schedule.
- A set of fields that become the columns of the schedule.
- Sorting and grouping criteria.
- Filters that restrict the set of elements visible in the schedule.
- Settings to control visibility of the title, headers, and grid lines.

Most schedules contain a single ScheduleDefinition which is retrieved via the ViewSchedule.Definition property. In Revit, schedules of certain categories can contain an "embedded schedule" containing elements associated with the elements in the primary schedule, for example a room schedule showing the elements inside each room or a duct system schedule showing the elements associated with each system. An embedded schedule has its own category, fields, filters, etc. Those settings are stored in a second ScheduleDefinition object. When present, the embedded ScheduleDefinition is obtained from the ScheduleDefinition.EmbeddedDefinition property.

Adding Fields

Once a ViewSchedule is created, fields can be added. The ScheduleDefinition.GetSchedulableFields() method will return a list of SchedulableField objects representing the non-calculated fields that may be included in the schedule. A new field can be added from a SchedulableField object or using a ScheduleFieldType. The following table describes the options available from the ScheduleFieldType enumeration.

Member name	Description
Instance	An instance parameter of the scheduled elements. All shared parameters also use this type, regardless of whether they are instance or type parameters.
ElementType	A type parameter of the scheduled elements.
Count	The number of elements appearing on the schedule row.
ViewBased	<p>A specialized type of field used for a few parameters whose displayed values can change based on the settings of the view:</p> <ul style="list-style-type: none"> • ROOM_AREA and ROOM_PERIMETER in room and space schedules. • PROJECT_REVISION_REVISION_NUM in revision schedules. • KEYNOTE_NUMBER in keynote legends that are numbered by sheet.
Formula	A formula calculated from the values of other fields in the schedule.
Percentage	A value indicating what percent of the total of another field each element represents.
Room	A parameter of the room that a scheduled element belongs to.
FromRoom	A parameter of the room on the "from" side of a door or window.

ToRoom	A parameter of the room on the "to" side of a door or window.
ProjectInfo	A parameter of the Project Info element in the project that the scheduled element belongs to, which may be a linked file. Only allowed in schedules that include elements from linked files.
Material	In a material takeoff, a parameter of one of the materials of a scheduled element.
MaterialQuantity	In a material takeoff, a value representing how a particular material is used within a scheduled element. The parameter ID can be MATERIAL_AREA, MATERIAL_VOLUME, or MATERIAL_ASPAIN.
RevitLinkInstance	A parameter of the RevitLinkInstance that an element in a linked file belongs to. Currently RVT_LINK_INSTANCE_NAME is the only supported parameter. Only allowed in schedules that include elements from linked files.
RevitLinkType	A parameter of the RevitLinkType that an element in a linked file belongs to. Currently RVT_LINK_FILE_NAME_WITHOUT_EXT is the only supported parameter. Only allowed in schedules that include elements from linked files.
StructuralMaterial	A parameter of the structural material of a scheduled element.
Space	A parameter of the space that a scheduled element belongs to.

Using one of the `ScheduleDefinition.AddField()` methods will add the field to the end of the field list. To place a new field in a specific location in the field list, use one of the `ScheduleDefinition.InsertField()` methods. Fields can also be ordered after the fact using `ScheduleDefinition.SetFieldOrder()`.

Scheduling Revision Clouds

Revision clouds and the view name and sheet info for revision clouds can be scheduled using the following values for the `Autodesk.Revit.DB.ScheduleFieldType` enum

- Revision - Represents the parameter of the revision element that the scheduled revision cloud element belongs to.
- Views - Represents the parameter of the primary view owning an annotation element (e.g. Revision Cloud.)
- Sheets - Represents the parameter of the sheet view owning an annotation element (e.g. Revision Cloud.)

The following is a simple example showing how to add fields to a view if they are not already in the view schedule.

Code Region: Adding fields to a schedule

```
/// <summary>
/// Add fields to view schedule.
/// </summary>
/// <param name="schedules">List of view schedule.</param>
public void AddFieldToSchedule(List<ViewSchedule> schedules)
{
    IList<SchedulableField> schedulableFields = null;

    foreach (ViewSchedule vs in schedules)
    {
        //Get all schedulable fields from view schedule definition.
        schedulableFields = vs.Definition.GetSchedulableFields();

        foreach (SchedulableField sf in schedulableFields)
        {
            bool fieldAlreadyAdded = false;
            //Get all schedule field ids
            IList<ScheduleFieldId> ids = vs.Definition.GetFieldOrder();
            foreach (ScheduleFieldId id in ids)
            {
                //If the GetSchedulableField() method of gotten schedule field
                // returns same schedulable field,
                // it means the field is already added to the view schedule.
            }
        }
    }
}
```

```
if (vs.Definition.GetField(id).GetSchedulableField() == sf)
{
    fieldAlreadyAdded = true;
    break;
}

//If schedulable field doesn't exist in view schedule, add it.

if (fieldAlreadyAdded == false)
{
    vs.Definition.AddField(sf);
}
}
```

The ScheduleField class represents a single field in a ScheduleDefinition's list of fields. Each (non-hidden) field becomes a column in the schedule.

Most commonly, a field represents an instance or type parameter of elements appearing in the schedule. Some fields represent parameters of other related elements, like the room to which a scheduled element belongs. Fields can also represent data calculated from other fields in the schedule, specifically Formula and Percentage fields.

The ScheduleField class has properties to control column headings, both the text as well as the orientation. Column width and horizontal alignment of text within a column can also be defined.

The ScheduleField.IsNullOrHidden property can be used to hide a field. A hidden field is not displayed in the schedule, but it can be used for filtering, sorting, grouping, and conditional formatting and can be referenced by Formula and Percentage fields.

DisplayType

The ScheduleField has a DisplayType property which indicates the display type for the field. Possible values are:

- **Standard** - Nothing is displayed if the values of the elements are different, otherwise, the common value will be displayed
- **Totals** - Calculates and displays the total value
- **MinMax** - Calculates and displays the minimum and maximum values
- **Min** - Calculates and displays the maximum value
- **Max** - Calculates and displays the minimum value The ScheduleField.CanDisplayMinMax() method indicates whether this field can display minimum and maximum values. In a non-itemized schedule, values for the non-Standard display types are displayed in regular rows when multiple elements appear on the same row.

Style and Formatting of Fields

ScheduleField.GetStyle() and ScheduleField.SetStyle() use the TableCellStyle class to work with the style of fields in a schedule. Using SetStyle(), various attributes of the field can be set, including the line style for the border of the cell as well as the text font, color and size.

ScheduleField.SetFormatOptions() and ScheduleField.GetFormatOptions() use the FormatOptions class to work with the formatting of a field's data. The FormatOptions class contains settings that control how to format numbers with units as strings. It contains those settings that are typically chosen by an end user in the Format dialog and stored in the document.

In the following example, all length fields in a ViewSchedule are formatted to display in feet and fractional inches.

Code Region: Formatting a field

```
// format length units to display in feet and inches format

public void FormatLengthFields(ViewSchedule schedule)

{
    int nFields = schedule.Definition.GetFieldCount();

    for (int n = 0; n < nFields; n++)

    {
        ScheduleField field = schedule.Definition.GetField(n);

        if (field.GetSpecTypeId() == SpecTypeId.Length)

        {
            FormatOptions formatOpts = new FormatOptions();

```

```
        formatOpts.UseDefault = false;

        formatOpts.SetUnitTypeId(UnitTypeId.FeetFractionalInches);

        field.SetFormatOptions(formatOpts);

    }

}

}
```

The example below applies both formatting and style overrides to a given field.

Code Region: Apply formatting and style overrides to field

```
public static void ApplyFormattingToField(ViewSchedule schedule, int fieldIndex)

{
    // Get the field.

    ScheduleDefinition definition = schedule.Definition;

    ScheduleField field = definition.GetField(fieldIndex);

    // Build unit formatting for the field.

    FormatOptions options = field.GetFormatOptions();

    options.UseDefault = false;

    options.SetUnitTypeId(UnitTypeId.SquareInches);

    options.SetSymbolTypeId(SymbolTypeId.InCaret2);

    // Build style overrides for the field

    // Use override options to indicate fields that are overridden and apply
    changes

    TableCellStyle style = field.GetStyle();
```

```
    TableCellStyleOverrideOptions overrideOptions = style.GetCellStyleOverrideOptions();

    overrideOptions.BackgroundColor = true;
    style.BackgroundColor = new Color(0x00, 0x00, 0xFF);

    overrideOptions.FontColor = true;
    style.TextColor = new Color(0xFF, 0xFF, 0xFF);

    overrideOptions.Italics = true;
    style.IsFontItalic = true;

    style.SetCellStyleOverrideOptions(overrideOptions);

    double width = field.GridColumnWidth;

    using (Transaction t = new Transaction(schedule.Document, "Set style etc"))
    {
        t.Start();
        field.SetStyle(style);
        field.SetFormatOptions(options);
        // Change column width (affects width in grid and on sheet) - units are in Revit length units - ft.
        field.GridColumnWidth = width + 0.5;
        t.Commit();
    }
}
```

Title and Headers

Display of the schedule title and/or headers is optional. Whether the title or headers are shown can be controlled with the ScheduleDefinition properties ShowTitle and ShowHeaders.

Grouping and Sorting in Schedules

A schedule may be sorted or grouped by one or more of the schedule's fields. Several methods can be used to control grouping and sorting of fields. The ScheduleSortGroupField class represents one of the fields that the schedule is sorted or grouped by. Sorting and grouping are related operations. In either case, elements appearing in the schedule are sorted based on their values for the field by which the schedule is sorted/grouped, which automatically causes elements with identical values to be grouped together. By enabling extra header, footer, or blank rows, visual separation between groups can be achieved.

If the ScheduleDefinition.IsItemized property is false, elements having the same values for all of the fields used for sorting/grouping will be combined onto the same row. Otherwise the schedule displays each element on a separate row

A schedule can be sorted or grouped by data that is not displayed in the schedule by marking the field used for sorting/grouping as hidden using the ScheduleField.isHidden property.

Code Region: Add grouping/sorting to schedule

```
public static void AddGroupingToSchedule(ViewSchedule schedule, BuiltInParameter paramEnum, bool withTotalsAndDecoration, ScheduleSortOrder order)
{
    // Find field
    ScheduleField field = FindScheduleField(schedule, paramEnum);

    if (field == null)
        throw new Exception("Unable to find field.");

    // Build sort/group field.
    ScheduleSortGroupField sortGroupField = new ScheduleSortGroupField(field.FieldId, order);
    if (withTotalsAndDecoration)
    {
        sortGroupField.ShowFooter = true;
        sortGroupField.ShowFooterTitle = true;
    }
}
```

```
sortGroupField.ShowFooterCount = true;
sortGroupField.ShowHeader = true;
sortGroupField.ShowBlankLine = true;

}

// Add the sort/group field

ScheduleDefinition definition = schedule.Definition;

using (Transaction t = new Transaction(schedule.Document, "Add sort/group
field"))

{
    t.Start();
    definition.AddSortGroupField(sortGroupField);
    t.Commit();
}

}

public static ScheduleField FindScheduleField(ViewSchedule schedule, BuiltInP
arameter paramEnum)
{
    ScheduleDefinition definition = schedule.Definition;
    ScheduleField foundField = null;
    ElementId paramId = new ElementId(paramEnum);

    foreach (ScheduleFieldId fieldId in definition.GetFieldOrder())
    {
        foundField = definition.GetField(fieldId);
    }
}
```

```
    if (foundField.ParameterId == paramId)

    {
        return foundField;
    }

}

return null;
}
```

Headers can also be grouped. ViewSchedule.GroupHeaders() method can be used to specify which rows and columns to include in a grouping of the header section. The last parameter is a string for the caption of the grouped rows and columns.

In the following example, columns are grouped for a newly created single-category schedule.

Code Region: Grouping headers

```
public static void CreateSingleCategoryScheduleWithGroupedColumnHeaders(Document doc)
{
    using (Transaction t = new Transaction(doc, "Create single-category with
grouped column headers"))

    {
        // Build the schedule

        t.Start();

        ViewSchedule vs = ViewSchedule.CreateSchedule(doc, new ElementId(Built
InCategory.OST_Windows));

        AddRegularField(vs, new ElementId(BuiltInParameter.WINDOW_HEIGHT));
        AddRegularField(vs, new ElementId(BuiltInParameter.WINDOW_WIDTH));
    }
}
```

```
        AddRegularField(vs, new ElementId(BuiltInParameter.ALL_MODEL_MARK));
        AddRegularField(vs, new ElementId(BuiltInParameter.ALL_MODEL_COST));

        doc.Regenerate();

        // Group the headers in the body section using ViewSchedule methods
        vs.GroupHeaders(0, 0, 0, 1, "Size");
        vs.GroupHeaders(0, 2, 0, 3, "Other");
        vs.GroupHeaders(0, 0, 0, 3, "All");

        t.Commit();
    }

}

public static void AddRegularField(ViewSchedule schedule, ElementId paramId)
{
    ScheduleDefinition definition = schedule.Definition;

    // Find a matching SchedulableField
    SchedulableField schedulableField =
        definition.GetSchedulableFields().FirstOrDefault<SchedulableField>(sf
=> sf.ParameterId == paramId);

    if (schedulableField != null)
    {
        // Add the found field
        definition.AddField(schedulableField);
    }
}
```

```
    }  
}
```

Filtering

A ScheduleFilter can be used to filter the elements that will be displayed in a schedule. A filter is a condition that must be satisfied for an element to appear in the schedule. All filters must be satisfied for an element to appear in the schedule.

A schedule can be filtered by data that is not displayed in the schedule by marking the field used for filtering as hidden using the ScheduleField.isHidden property.

Code Region: Add filter to schedule

```
public static void AddFilterToSchedule(ViewSchedule schedule, ElementId level  
Id)  
{  
    // Find level field  
  
    ScheduleDefinition definition = schedule.Definition;  
  
    ScheduleField levelField = FindField(schedule, BuiltInParameter.ROOM_LeVE  
L_ID);  
  
    // Add filter  
  
    using (Transaction t = new Transaction(schedule.Document, "Add filter"))  
    {  
        t.Start();  
  
        // If field not present, add it  
  
        if (levelField == null)  
        {
```

```
        levelField = definition.AddField(ScheduleFieldType.Instance, new
ElementId(BuiltInParameter.ROOM_LEVEL_ID));

    }

    // Set field to hidden

    levelField.Hidden = true;

    ScheduleFilter filter = new ScheduleFilter(levelField.FieldId, Schedu
leFilterType.Equal, levelId);

    definition.AddFilter(filter);

    t.Commit();

}

}

/// <summary>
/// Finds an existing ScheduleField matching the given parameter
/// </summary>
/// <param name="schedule"></param>
/// <param name="paramEnum"></param>
/// <returns></returns>
public static ScheduleField FindField(ViewSchedule schedule, BuiltInParameter
paramEnum)

{
    ScheduleDefinition definition = schedule.Definition;

    ScheduleField foundField = null;

    ElementId paramId = new ElementId(paramEnum);

    foreach (ScheduleFieldId fieldId in definition.GetFieldOrder())
}
```

```

    {

        foundField = definition.GetField(fieldId);

        if (foundField.ParameterId == paramId)

        {

            return foundField;

        }

    }

    return null;

}

```

Multiple Value Formatting

The `ScheduleField` class has properties which allow customization of the multiple value indication per field:

- `ScheduleField.MultipleValuesDisplayType`
- `ScheduleField.MultipleValuesText` - The text to be used when a field has multiple values
- `ScheduleField.MultipleValuesCustomText` - The text to be used when a field has multiple values and the display type is set to `ScheduleFieldMultipleValuesDisplayType.Custom`

The Enum `ScheduleFieldMultipleValuesDisplayType` defines how the schedule field's multiple value indication is displayed (using the project setting, a custom text, or a predefined text "<varies>").

Working with Schedule Data

The following example shows how to determine the list of elements in a schedule.

Code Region: Get a schedule's contents

```

public static void GetScheduleContents(ViewSchedule viewSchedule)
{
    // Collect types displayed in the schedule
}

```

```

    FilteredElementCollector typeCollector = new FilteredElementCollector(vie
wSchedule.Document, viewSchedule.Id);

    typeCollector.WhereElementIsElementType();

    int numberOfTypes = typeCollector.Count();

    // Collect instances displayed in the schedule

    FilteredElementCollector instCollector = new FilteredElementCollector(vie
wSchedule.Document, viewSchedule.Id);

    instCollector.WhereElementIsNotElementType();

    int numberOfInstances = instCollector.Count();

    TaskDialog.Show("Elements in schedule", String.Format("Types {0} instance
s {1}", numberOfTypes, numberOfInstances));
}

```

To work with the actual data in the schedule, the `ViewSchedule.GetTableData()` returns a `TableData` object that holds most of the data that describe the style and contents of the rows, columns, and cells in a table. More information can be found under [TableView](#).

2.6.3.7.3 PanelScheduleView

`PanelScheduleView` represents a panel schedule which displays information about a panel, the circuits connected to the panel, and their corresponding loads.

You can create a schedule that lists the circuits connected to a panel, and displays information about each circuit such as location on the panel, circuit name and apparent loads. Panel schedules display four main information sections: a header, circuit table, a loads summary and a footer. A new Panel Schedule view for the selected panel is displayed in the drawing area, and panel schedules are added to the project browser under the `Panel Schedules` folder. A panel schedule shows the following data:

- Panel Name
- Distribution System supported by the panel
- Number of phases available from the panel

- Number of wires specified for the distribution system assigned to this panel
- Rating of the mains feeding the panel
- Type of mounting (Surface or Recessed)
- Type of case enclosing the panel
- Room where the panel is installed
- Name assigned to a load circuit
- Rated trip current for a circuit breaker
- Number of poles on the circuit breaker
- Circuit number
- Phases
- Apparent load (VA) for each of the phases
- Total apparent load for all three phases
- Manufacturer
- Notation of any changes made to the panel
- Root Means Square amperage Additional circuit and panel information to display can be specified in the panel schedule templates, represented in the Revit API by the PanelScheduleTemplate class.

PanelScheduleView is derived from the TableView class as is ViewSchedule. Some of the common functionality between schedules and panel schedules can be found in the [Schedule Classes](#) topic.

Panel schedule creation

There are two static overloads for creating a PanelScheduleView. One overload of PanelScheduleView.CreateInstanceView() only requires the document in which to create the panel schedule and the id of the electrical panel element associated with the schedule. This method uses the default panel schedule template to create the new view. The other overload takes the id of a specific PanelScheduleTemplate to use.

The following example creates a new panel schedule from a user-selected electrical panel using the default template and switches the active view to the new panel schedule view.

Code Region: Create a panel schedule

```
// Create a new panel schedule and switch to that view
public void CreatePanelSchedule(UIDocument uiDocument)
{
```

```
Document doc = uiDocument.Document;

Reference selected = uiDocument.Selection.PickObject(ObjectType.Element,
"Select an electrical panel");

Element panel = doc.GetElement(selected);

if (null != panel)

{

    PanelScheduleView psv = null;

    using (Transaction trans = new Transaction(doc, "Create a new panel s
chedule"))

    {

        trans.Start();

        psv = PanelScheduleView.CreateInstanceView(doc, panel.Id);

        trans.Commit();

    }

    if (null != psv)

    {

        uiDocument.ActiveView = psv;      // make new view the active view

    }

    else

    {

        TaskDialog.Show("Revit", "Please select one electrical panel.");

    }

}
```

{}

Working with panel schedules

After a schedule is created, you may want to modify it. Several methods are helpful for moving data in the schedule. To move data around, use `PanelScheduleView.GetCellsBySlotNumber()` to get the range of cells for a specified slot number. `PanelScheduleView.MoveSlotTo()` moves the circuits in the source slot to the specific slot. Prior to moving the circuits, call `PanelScheduleView.CanMoveSlotTo()` to ensure the move is allowable.

If the moving circuit is in a group, all circuits in the group will be moved accordingly. The `IsSlotGrouped()` method will check if the slot is in a group. This method returns 0 if the slot is not in a group. If it is in a group, the returned value will be the group number (a value greater than 0).

2.6.4 Revisions

The Revit API provides several classes and members for accessing project Revisions, their settings and associated Revision Clouds.

Settings

The `RevisionSettings` class allows an application to read and modify the project-wide settings that affect Revisions and Revision Clouds. The static `RevisionSettings.GetRevisionSettings()` method returns the `RevisionSettings` object for the given project document. The following properties can be used to access project-wide Revision settings:

- `RevisionCloudSpacing` - determines the size in paper space of revision clouds drawn in a project.
- `RevisionNumbering` - determines whether revision numbers for the project are determined on a per sheet or a whole project basis. The `AlphanumericRevisionSettings` class contains settings that apply to Revisions with the Alphanumeric RevisionNumberType. The `RevisionSettings` methods `GetAlphanumericRevisionSettings()` and `SetAlphanumericRevisionSettings()` provide read and write access to the `AlphanumericRevisionSettings`. `AlphanumericRevisionSettings` offers the following members:
 - `Prefix` - a prefix to be prepended to each revision number with alphanumeric type.
 - `Suffix` - a suffix to be appended to each revision number with alphanumeric type.
 - `GetSequence()` - gets the list of strings to be used as the numbering sequence for revisions with the alphanumeric type.
 - `SetSequence()` - sets the list of strings for numbering revisions of this type. Similarly, the `NumericRevisionSettings` class contains settings that apply to Revisions with the Numeric RevisionNumberType. The `RevisionSettings` methods `GetNumericRevisionSettings()` and `SetNumericRevisionSettings()` provide read and write access to these settings. `NumericRevisionSettings` offers the following members:

- Prefix - a prefix to be prepended to each revision number with numeric type.
- Suffix - a suffix to be appended to each revision number with numeric type.
- StartNumber property - the value to be used as the first number in the sequence of numeric revisions.

When revision clouds appear on a sheet, the revision number of each revision can be displayed either by tagging the revision cloud or by a revision schedule within the sheet's titleblock. There are two ways the number can be determined:

- **Per project:** The value of the Revision numbers will always correspond to the project-wide Revision Sequence Number assigned to the revision. For example, if revision clouds for revisions with sequence numbers 5, 7, and 8 are placed on a sheet then revision tags and schedules on that sheet would display 5, 7, and 8.
- **Per sheet:** Revision numbers will be assigned consecutive numbers based on the revision clouds visible on that sheet. For example, if revision clouds for revisions assigned project-wide Revision Sequence Numbers 5, 7, and 8 are placed on a sheet then revision tags and schedules on that sheet would display 1, 2, and 3. The sequence on the sheet will still follow the relative ordering of the Revision Sequence Numbers, so in this example revision 5 would be displayed as 1 on the sheet, revision 7 would be displayed as 2, etc.

The Revision class allows an application to read and modify the existing revisions in a project and to create new revisions. The Revision object represents the data related to a single revision in the project. It has properties such IssuedBy, IssuedTo, RevisionNumber, SequenceNumber and RevisionDate. Revision clouds and tags can be associated with a particular Revision object to display its properties on sheets.

The revisions in the project are stored in a specific order called the revision sequence. The revision sequence represents the conceptual sequence in which revisions will be issued. The static method Revision.GetAllRevisionIds() will return the ids of all Revisions in this order. The static method Revision.ReorderRevisionSequence() can be used to change the sequence of revisions with the project. Note that the newly specified sequence must include every Revision in the project exactly once and that changing the sequence of Revisions can change the SequenceNumber and RevisionNumber of Revisions that have already been issued.

The `RevisionNumberingSequence` class defines the sequences by which numbers are assigned to Revisions. Revision numbering is either numeric or alphanumeric. Alphanumeric from the API corresponds to the UI concept of "Custom". Members of this class provide the ability to create, read and modify the settings related to Revision numbering sequences.

`Revision.GetRevisionNumberingSequenceId()` and `Revision.SetRevisionNumberingSequenceId()` provide access to the sequence which controls a revision's numbering.

The static Create() method will create a new Revision in the specified document. In the sample below, multiple revisions are added and their properties are set.

Code Region: Creating new revisions

```
public IList<Revision> AddRevisions(Document document)
{
    IList<Revision> newRevisions = new List<Revision>();

    using (Transaction createRevision = new Transaction(document, "createRevision"))
    {
        createRevision.Start();

        newRevisions.Add(AddNewRevision(document, "Include door tags", "manager1", "employee1", 1, DateTime.Now));
        newRevisions.Add(AddNewRevision(document, "Add a section view", "manager1", "employee1", 2, DateTime.Now));
        newRevisions.Add(AddNewRevision(document, "Make callout view larger", "manager1", "employee1", 3, DateTime.Now));

        createRevision.Commit();
    }

    return newRevisions;
}

private Revision AddNewRevision(Document document, string description, string issuedBy, string issuedTo, int sequenceNumber, DateTime date)
{
    Revision newRevision = Revision.Create(document);

    newRevision.Description = description;
    newRevision.IssuedBy = issuedBy;
    newRevision.IssuedTo = issuedTo;
}
```

Code Region: Creating new revisions

```

    AlphanumericRevisionSettings ars = new AlphanumericRevisionSettings();

    RevisionNumberingSequence sequence = RevisionNumberingSequence.CreateAlph
    numericSequence(document, "name", ars);

    newRevision.RevisionNumberingSequenceId = sequence.Id;

    newRevision.RevisionDate = date.ToShortDateString();

    return newRevision;
}

```

Two methods, `CombineWithNext()` and `CombineWithPrevious()` allow an application to combine a specified Revision with the next or previous Revision in the model. Combining the Revisions means that the RevisionClouds and revision tags associated with the specified Revision will be reassociated with the next Revision and the specified Revision will be deleted from the model. This method returns the ids of the RevisionClouds that were reassociated. However, these operations can only be implemented if neither Revision has been issued.

The following example demonstrates use of the `CombineWithNext()` method. It also uses the `GetAllRevisionIds()` method to find the next revision to be sure the `CombineWithNext()` method will be successful.

Code Region: Combining revisions

```

private bool CombineRevision(Document document, Revision revision)
{
    bool combined = false;

    // Can only combine two revisions if neither have been issued

    if (revision.Issued == false)
    {
        ElementId revisionId = revision.Id;

        Revision nextRevision = GetNextRevision(document, revisionId);

```

Code Region: Combining revisions

```
if (nextRevision != null && nextRevision.Issued == false)

{
    ISet<ElementId> revisionCloudIds = Revision.CombineWithNext(document, revisionId);

    combined = true;

    int movedClouds = revisionCloudIds.Count;

    if (movedClouds > 0)

    {
        RevisionCloud cloud = document.GetElement(revisionCloudIds.ElementAt(0)) as RevisionCloud;

        if (cloud != null)

        {
            string msg = string.Format("Revision {0} deleted and {1} revision clouds were added to Revision {2}",

                revisionId.ToString(), movedClouds, cloud.RevisionId.ToString());

            TaskDialog.Show("Revision Combined", msg);
        }
    }

}

return combined;
}

private Revision GetNextRevision(Document document, ElementId currentRevisionId)
```

Code Region: Combining revisions

```
{  
    Revision nextRevision = null;  
  
    IList<ElementId> revisionIds = Revision.GetAllRevisionIds(document);  
  
    int currentRevisionIndex = -1;  
  
    for (int n = 0; n < revisionIds.Count; n++)  
  
    {  
        if (revisionIds[n] == currentRevisionId)  
  
        {  
            currentRevisionIndex = n;  
  
            break;  
        }  
  
    }  
  
    // if the current revision id was found and is not the last index  
  
    if (currentRevisionIndex >= 0 && currentRevisionIndex < revisionIds.Count - 1)  
  
    {  
        ElementId nextRevisionId = revisionIds[currentRevisionIndex + 1];  
  
        nextRevision = document.GetElement(nextRevisionId) as Revision;  
  
    }  
  
    return nextRevision;  
}
```

Revision clouds

A RevisionCloud is a graphical "cloud" that can be displayed on a view or sheet to indicate where revisions in the model have occurred. The RevisionCloud class allows an application to

access information about the revision clouds that are present within a model and to create new revision clouds.

RevisionClouds are view specific and can be created in most graphical views, except 3D.

Note also that when a RevisionCloud is created in a ViewLegend, it is treated as a legend representation of what a RevisionCloud looks like rather than as an actual indication of a change to the model. As a result, RevisionClouds in ViewLegends will not affect the contents of revision schedules.

Creating revision clouds

The static Create() method allows an application to create a new RevisionCloud in a specified view based on a series of lines and curves. RevisionClouds can only be created if the associated Revision has not yet been issued.

RevisionClouds can be created in most graphical Views, excepting 3D views and graphical column schedules. Unlike most other Elements, RevisionClouds can be created directly on a ViewSheet.

RevisionClouds are created based on a series of sketched curves. There is no requirement that the curves form closed loops and self-intersections are also permitted. The curves will be automatically projected onto the appropriate plane for the View. The list of curves cannot be empty and no lines can be perpendicular to the View's plane. If the View is a model View, the coordinates specified for the curves will be interpreted in model space. If the View is a non-model View (such as a ViewSheet) then the coordinates will be interpreted in the View's space.

Each curve will have a series of "cloud bumps" drawn along it to form the appearance of a cloud. The cloud graphics will be attached to the curves under the assumption that each curve is oriented in a clockwise direction. For lines, this means that the outside of the cloud is in the direction of the line's normal vector within the View's plane. Any closed loops should therefore be oriented clockwise to create the typical cloud shape.

Code Region: Create revision cloud

```
private void CreateRevisionCloudInActiveView(Document document, Revision revision, IList<Curve> curves)
{
    using (Transaction newRevisionCloud = new Transaction(document, "Create Revision Cloud"))
    {
        newRevisionCloud.Start();
        // Can only create revision cloud for revision that is not issued
```

Code Region: Create revision cloud

```
if (revision.Issued == false)
{
    RevisionCloud.Create(document, document.ActiveView, revision.Id,
curves);

    newRevisionCloud.Commit();

}
else
{
    newRevisionCloud.RollBack();
}

}
```

Revision cloud geometry

RevisionCloud is derived from the Element class. The Element.Geometry property for revision clouds will return the actual curved lines that make up the cloud. The RevisionCloud.GetSketchCurves() method on the other hand, will return the sketched curves that define the basic outline of the cloud and not the arcs that Revit attaches to these curves to create the cloud appearance.

Revision Associated with RevisionCloud

Each RevisionCloud is associated with one Revision. The associated revision id is specified when calling Create() and can be retrieved from the RevisionCloud.RevisionId property. The RevisionId property for a RevisionCloud can be changed if it is not associated with a Revision that has already been issued. It can only be changed to the id of another Revision that has also not been issued. RevisionCloud.IsRevisionIssued() returns whether the associated Revision has been issued.

ViewSheets

When a RevisionCloud is visible on a ViewSheet (either because it is directly placed on that ViewSheet or because it is visible in a View placed on the ViewSheet), any revision schedules displayed on the ViewSheet will automatically include the Revision associated with the RevisionCloud.

The RevisionCloud.GetSheetIds() method returns the ids of the ViewSheets where it may appear and contribute to the sheet's revision schedule. A RevisionCloud can appear on a ViewSheet because it is drawn directly on the ViewSheet or because its owner view is placed on the ViewSheet. If the RevisionCloud is owned by a view that is a dependent view or has associated dependent views, then the RevisionCloud can also be visible on the sheets where the related dependent or primary views have been placed.

This RevisionCloud may not be visible in all ViewSheets reported by this method. Additional factors, such as the visibility settings or annotation crop of the Views or the visibility settings of the associated Revision may still cause this RevisionCloud to not appear on a particular ViewSheet.

If this RevisionCloud is owned by a ViewLegend, no sheets will be returned because the RevisionCloud will not participate in revision schedules. The ViewSheet class includes methods for working with Revisions and RevisionClouds on sheets. See the [ViewSheet](#) topic for more information.

2.6.5 View Filters

Filters are elements independent of views. They can be applied to Views using the ParameterFilterElement class or a SelectionFilterElement class.

ParameterFilterElement

A ParameterFilterElement filters elements based on the elements' categories and an ElementFilter. The ElementFilter must be either an ElementParameterFilter or an ElementLogicalFilter containing only ElementParameterFilters and other ElementLogicalFilters. This allows the view filter to be constructed from a combination of AND and OR filters that are then gathered by an AND filter, an OR filter, or an ElementParameterFilter as an input to ParameterFilterElement.Create().

Once a filter has been defined (with one or more categories and zero or more filter rules), it can be applied to a View using one of several methods. The View.AddFilter() method will apply the filter to the view, but with default overrides, meaning the view's display will not change. View.SetFilterOverrides() sets graphical overrides associated with a filter. Setting elements that pass a filter to not be visible in a view is done with View.SetFilterVisibility(ElementId filterElementId, bool visibility). AddFilter() and SetFilterVisibility() will both apply the filter to the view if it is not already applied, making it unnecessary to call AddFilter() separately.

Validation

Validation restrictions for the ElementFilter stored by a ParameterFilterElement (the class that represents a View Filter) support flexible creation of OR filters where criteria can reference parameters only associated to specific categories. The ElementFilter must be either an ElementParameterFilter or an ElementLogicalFilter representing a Boolean combination of ElementParameterFilters.

In addition, Revit checks that each ElementParameterFilter satisfies the following conditions:

- Its array of FilterRules is not empty and contains:

- Any number of FilterRules of type FilterValueRule, FilterInverseRule, and SharedParameterApplicableRule or
- Exactly one FilterCategoryRule containing only one category from categories stored by this ParameterFilterElement or
- Exactly two rules: the first one is a FilterCategoryRule containing only one category from categories stored by this ParameterFilterElement and the second one is a FilterRule of type FilterValueRule, FilterInverseRule, or SharedParameterApplicableRule.

Cases in the second and third bullet are currently allowed only if the parent node of ElementParameterFilter is LogicalOrFilter.

The method `ParameterFilterElement.GetElementFilterParametersForCategory()` retrieves a list of the parameters associated with all rules in the filter that are combined (using logical AND) with a FilterCategoryRule corresponding to single categoryId.

These methods identify or set if the filter is enabled in this view:

- `View.GetIsFilterEnabled()`
- `View.SetIsFilterEnabled()`

`View.GetOrderedFilters()` gets the filters applied to the view in the order they are applied.

The following example creates a new view filter matching multiple criteria and then hides those elements in the view.

Code Region: Applying a parameter filter to a view

```
public static void CreateViewFilter(Document doc, View view)
{
    List<ElementId> categories = new List<ElementId>();
    categories.Add(new ElementId(BuiltInCategory.OST_Walls));
    List<ElementFilter> elementFilterList = new List<ElementFilter>();

    using (Transaction t = new Transaction(doc, "Add view filter"))
    {
        t.Start();

        // Criterion 1 - wall type Function is "Exterior"
    }
}
```

```
ElementId exteriorParamId = new ElementId(BuiltInParameter.FUNCTION_PARAMETER);

elementFilterList.Add(new ElementParameterFilter(ParameterFilterRuleFactory.CreateEqualsRule(exteriorParamId, (int)WallFunction.Exterior)));

// Criterion 2 - wall length > = 28 or < = 14

ElementId lengthId = new ElementId(BuiltInParameter.CURVE_ELEM_LENGTH);

LogicalOrFilter wallHeightFilter = new LogicalOrFilter(
    new ElementParameterFilter(ParameterFilterRuleFactory.CreateGreaterOrEqualRule(lengthId, 28.0, 0.00001)),
    new ElementParameterFilter(ParameterFilterRuleFactory.CreateLessOrEqualRule(lengthId, 14.0, 0.00001)));
    
elementFilterList.Add(wallHeightFilter);

// Criterion 3 - custom shared parameter value matches string pattern

// Get the id for the shared parameter - the ElementId is not hardcoded, so we need to get an instance of this type to find it

Guid spGuid = new Guid("96b00b61-7f5a-4f36-a828-5cd07890a02a");

FilteredElementCollector collector = new FilteredElementCollector(doc);

collector.OfClass(typeof(Wall));

Wall wall = collector.FirstElement() as Wall;

if (wall != null)
{
    Parameter sharedParam = wall.get_Parameter(spGuid);
    ElementId sharedParamId = sharedParam.Id;
```

```
        elementFilterList.Add(new ElementParameterFilter(ParameterFilterRuleFactory.CreateBeginsWithRule(sharedParamId, "15.")));

    }

    // Create filter element associated to the input categories

    LogicalAndFilter andFilter = new LogicalAndFilter(elementFilterList);

    if (ParameterFilterElement.ElementFilterIsAcceptableForParameterFilterElement(doc, new HashSet<ElementId>(categories), andFilter))

    {

        ParameterFilterElement parameterFilterElement = ParameterFilterElement.Create(doc, "Example view filter", categories, andFilter);

        // Apply filter to view

        view.AddFilter(parameterFilterElement.Id);

        view.SetFilterVisibility(parameterFilterElement.Id, false);

    }

    else

    {

        TaskDialog.Show("Error", "Filter cannot be used");

    }

    t.Commit();

}

}
```

SelectionFilterElement

A SelectionFilterElement is a special view filter not based on rules, but on a group of possibly unrelated elements. Specific elements can be added to the filter as required and the resulting selection can be overridden just like ParameterFilterElement.

The following example creates a new selection filter and applies an override to it.

Code Region: Applying a selection filter to a view

```
public static void CreateSelectionFilter(Document doc, View view)
{
    // find room tags in this view

    FilteredElementCollector collector = new FilteredElementCollector(doc, vi
ew.Id);

    collector.WherePasses(new RoomTagFilter());

    // collect tags whose room number matches criteria
    List<ElementId> tagIds = new List<ElementId>();

    foreach (RoomTag tag in collector.Cast<RoomTag>())
    {
        int number = Int32.Parse(tag.Room.Number);

        if (number % 3 == 0)
        {
            tagIds.Add(tag.Id);
        }
    }

    using (Transaction t = new Transaction(doc, "Create SelectionFilterElemen
t"))
    {
        t.Start();

        // Create selection filter and assign ids
    }
}
```

```
    SelectionFilterElement filterElement = SelectionFilterElement.Create
(doc, "Room tags filter");

    filterElement.SetElementIds(tagIds);

    ElementId filterId = filterElement.Id;

    // Add the filter to the view
    view.AddFilter(filterId);

    doc.Regenerate();

    // Use the existing graphics settings, and change the color to Blue
    OverrideGraphicSettings overrideSettings = view.GetFilterOverrides(
filterId);

    overrideSettings.SetProjectionLineColor(new Color(0x00, 0x00, 0xFF));

    view.SetFilterOverrides(filterId, overrideSettings);

    t.Commit();
}

}
```

Modifying filters

All filters applied to a view can be retrieved using the `View.GetFilters()` method which will return a list of filter ids. Filter visibility and graphic overrides can be checked for a specific filter using the `View.GetFilterVisibility()` and `View.GetFilterOverrides()` methods respectively. `View.RemoveFilter` will remove a filter from the view.

The following example demonstrates how to get the filters in a view and then modifies the overrides associated with any filter currently setting the cut color to red.

Code Region: Modify existing filter

```
public static void ModifyExistingFilter(Document doc, View view)

{
    // Find any filter with overrides setting cut color to Red

    Dictionary<ElementId, OverrideGraphicSettings> filterIdsToChange = new Dictionary<ElementId, OverrideGraphicSettings>();

    foreach (ElementId filterId in view.GetFilters())
    {
        OverrideGraphicSettings overrideSettings = view.GetFilterOverrides(filterId);

        Color lineColor = overrideSettings.CutLineColor;

        if (lineColor == Color.InvalidColorValue)
            continue;

        // Save overrides setting the cut color to green

        if (lineColor.Red == 0xFF && lineColor.Green == 0x00 && lineColor.Blue == 0x00)
        {
            overrideSettings.SetCutLineColor(new Color(0x00, 0xFF, 0x00));
            filterIdsToChange[filterId] = overrideSettings;
        }
    }

    // Make the change to all found filters
}
```

```
using (Transaction t = new Transaction(doc, "Change override filters"))
{
    t.Start();

    foreach (ElementId filterId in filterIdsToChange.Keys)
    {
        view.SetFilterOverrides(filterId, filterIdsToChange[filterId]);
    }
    t.Commit();
}
```

2.6.6 View Cropping

The crop region for some views may be modified using the Revit API. The `ViewCropRegionShapeManager.CanHaveShape` property indicates whether the view is allowed to manage the crop region shape while the `ShapeSet` property indicates whether a shape has been set. The following example crops a view around the boundary of a room.

Code Region: Cropping a view

```
public void CropAroundRoom(Room room, View view)
{
    if (view != null)
    {
        IList<IList<Autodesk.Revit.DB.BoundarySegment>> segments = room.GetBoundarySegments(new SpatialElementBoundaryOptions());
        if (null != segments) //the room may not be bound
```

```
{  
    foreach (IList<Autodesk.Revit.DB.BoundarySegment> segmentList in  
segments)  
    {  
        CurveLoop loop = new CurveLoop();  
        foreach (Autodesk.Revit.DB.BoundarySegment boundarySegment in  
segmentList)  
        {  
            loop.Append(boundarySegment.GetCurve());  
        }  
  
        ViewCropRegionShapeManager vcrShapeMgr = view.GetCropRegionSh  
apeManager();  
        vcrShapeMgr.SetCropShape(loop);  
        break; // if more than one set of boundary segments for roo  
m, crop around the first one  
    }  
}  
}  
}
```

2.6.7 Displaced Views

Create a displaced view using the `DisplacementElement` class. `DisplacementElement` is a view-specific element that can be used to cause elements to appear displaced from their actual location. Displaced views are useful to illustrate the relationship model elements have to the model as a whole. The `DisplacementElement` does not actually change the location of any model elements; it merely causes them to be displayed in a different location.

For a detailed example of creating displaced views, see the `DisplacementElementAnimation` sample in the Revit SDK.

Creating a Displaced View

The static `DisplacementElement.Create()` method creates a new `DisplacementElement`. The new `DisplacementElement` may be a child of a parent `DisplacementElement` if the `parentDisplacementElement` parameter is not null. If a parent is specified, the child `DisplacementElement`'s transform will be concatenated with that of the parent, and the displacement of its associated elements will be relative to the parent `DisplacementElement`.

The `Create()` method also requires a document, a list of elements to be displaced, the owner view, and the translation to be applied to the graphics of the displaced elements. An element may only be displaced by a single `DisplacementElement` in any view. Assigning an element to more than one `DisplacementElement` will result in an exception.

Other static methods of `DisplacementElement` can be used prior to calling `Create()` to help prevent any exceptions. `CanCategoryBeDisplaced()` tests whether elements belonging to a specific category can be displaced, while the overloaded static method `CanElementsBeDisplaced()` indicates if specific elements may be assigned to a new `DisplacementElement`. `IsAllowedAsDisplacedElement()` tests a single element for eligibility to be displaced.

The static `GetAdditionalElementsToDisplace()` method will return any additional elements that should be displaced along with the specified element in a specified view. For example, when a wall is displaced, any inserts or hosted elements should also be displaced.

When creating a child `DisplacementElement`, the static `IsValidAsParentInView()` can be used to verify a specific `DisplacementElement` may be used as a parent in a specific View.

Other static methods of `DisplacementElement` can be used to find the `DisplacementElement` that includes a specific element, to get a list of all displaced elements in a View, or to get all the `DisplacementElements` owned by a specified View.

Working with Displaced Elements

Once a new `DisplacementElement` has been created, methods are available to obtain any child `DisplacementElements`, to get the ids of all elements affected by the `DisplacementElement`, or to obtain the ids of all elements affected by the `DisplacementElement` as well as any child `DisplacementElements`. The `ParentId` property will return the element id of the parent `DisplacementElement`, if there is one.

After creation, the set of elements affected by the `DisplacementElement` can be modified using `SetDisplacedElementIds()` or `RemoveDisplacedElement()`. Additionally, the relative displacement can be changed.

The method `ResetDisplacedElements()` will set the translation of the `DisplacementElement` to (0, 0, 0). The `DisplacementElement` continues to exist, but its elements are displayed in their actual location.

Creating a Displaced Path

`DisplacementPath` is a view-specific annotation related to a `DisplacementElement`. The `DisplacementPath` class creates an annotation that depicts the movement of the element from its actual location to its displaced location. The `DisplacementPath` is anchored to the `DisplacementElement` by a reference to a point on an edge of a displaced element of the

DisplacementElement. It is represented by a single line, or a series of jogged lines, originating at the specified point on the displaced element.

The static DisplacementPath.Create() method requires a document, id of the associated DisplacementElement, a reference that refers to an edge or curve of one of the elements displaced by the DisplacementElement, and a value in the range [0,1] that is a parameter along the edge specified. Once created, the path style of the DisplacementPath can get set using the PathStyle property. The anchor point can also be changed using SetAnchorPoint().

The following example creates a new displacement by moving the first wall found vertically and horizontally and then adds a displacement path for it.

Code Region: Create displacement and path

```
public static void CreateDisplacementAndPath(Document doc, View view)

{
    // Find roof

    FilteredElementCollector fec = new FilteredElementCollector(doc);

    fec.OfClass(typeof(RoofBase));

    RoofBase roof = fec.FirstElement() as RoofBase;

    // Get a geometric reference for the path

    Reference edgeRef = GetHorizontalEdgeReference(roof);

    using (Transaction t = new Transaction(doc, "CreateDisplacementAndPath"))

    {
        t.Start();

        // Create a new top level DisplacementElement

        DisplacementElement dispElem = DisplacementElement.Create(doc, new ElementId[] { roof.Id }, new XYZ(10, 0, 20), view, null);

        // Create the path associated to the element

        DisplacementPath.Create(doc, dispElem, edgeRef, 0.5);
    }
}
```

```
        t.Commit();

    }

}

private static Reference GetHorizontalEdgeReference(Element elem)
{
    //Find target edge from lower face of roof
    Options options = new Options();
    options.ComputeReferences = true;

    GeometryElement geomElem = elem.get_Geometry(options);

    foreach (var geomObj in geomElem)
    {
        if (geomObj is Solid)
        {
            Solid solid = geomObj as Solid;
            var faces = solid.Faces;

            foreach (Face face in faces)
            {
                BoundingBoxUV box = face.GetBoundingBox();
                UV midpoint = (box.Min + box.Max) / 2.0;
                if (face.ComputeNormal(midpoint).Normalize().Z < -0.1) // Downward facing, this is good enough
                {
                    var edgeLoops = face.EdgeLoops;
```

```

        foreach (EdgeArray edgeArray in edgeLoops)
    {
        foreach (Edge edge in edgeArray)
        {
            // horizontal?

            if (Math.Abs(edge.AsCurve().ComputeDerivatives(0.
0, true).BasisX.DotProduct(XYZ.BasisZ)) - 1 <= 0.00001)

            {
                return edge.Reference;
            }
        }
    }

}

}

}

return null;
}

```

The associated DisplacementElement may have a parent DisplacementElement and this parent may have its own parent DisplacementElement, producing a series of ancestors. The terminal point may be the point's original (un-displaced) location, or the corresponding point on any of the intermediate displaced locations corresponding to these ancestor DisplacementElements. The DisplacementPath . AncestordIdx property specifies the end point of the path.

2.6.8 UIView

While the View class is the base class for all view types in Revit and keeps tracks of elements in the view, the UIView class contains data about the view windows in the Revit user interface. A list of all open views can be retrieved from the UIDocument using the GetOpenUIViews()

method. The `UIView` class has methods to get information about the views drawing area as well as to pan and zoom the active view.

`UIView.GetWindowRectangle()` returns a rectangle that describes the size and placement of the `UIView` window. It does not include the window border or title bar.

Zoom Operations

`UIView` has several methods related to zooming the active view. `UIView.GetZoomCorners()` gets the corners of the view's rectangle in model coordinates and `UIView.ZoomAndCenterRectangle()` offers the ability to zoom and pan the active view to center on the input region of the model.

The `ZoomToFit()` and `ZoomSheetSize()` methods provide quick ways to adjust the zoom of the window, while the `Zoom()` method can be used to zoom in or out by a specified factor.

Closing a View

`UIView.Close()` can close a visible window. However, it cannot be used to close the last active window. Attempting to close the last active window will throw an exception.

2.6.9 View Templates

A View Template is a special kind of View that can controls parameters of views that display elements in the Revit model. View Templates themselves do not display Revit elements.

Basic functions that can be used with View Templates

- `View.CreateViewTemplate()` - Creates a new view template instance from the view instance
- `View.IsViewValidForTemplateCreation()` - Verifies that the view is valid for template creation
- `View.IsTemplate` - Test whether the view is a view template
- `View.IsValidViewTemplate(ElementId templateId)` - Verifies that the view represented by `templateId` can be set as the controlling view template for this view
- `View.ViewTemplateId` - The id of the template view that controls this view's parameters
- `IList<ElementId> View.GetTemplateParameterIds()` - Returns a list of parameter ids that may be controlled when this view is assigned as a template
- `ICollection<ElementId> View.GetNonControlledTemplateParameterIds()` - Returns a list of parameters that are not marked as included when this view is used as a template
- `View.SetNonControlledTemplateParameterIds(ICollection<ElementId> newSet)` - Sets the parameters that will not be included when this view is used as a template

Below is list of controllable built-in parameters along with references to their respective API getters/setters.

Parameter Name in View Template's Dialog	View Template Parameter Name	View Type Limitations	API getters/setters	Comments
View Scale Scale Value	VIEW_SCALE	Valid for all plan views, all 3D views, Section, Elevation, Detail, DraftingView	View.Scale property	To include/exclude in non-controlled parameters set, VIEW_SCALE has to be used together with with the following parameters: VIEW_SCALE_PULLDOWN_IMPERIAL VIEW_SCALE_PULLDOWN_METRIC
Display Model	VIEW_MODEL_DISPLAY_MODE	Valid for all plan views, Section, Elevation, Detail	Element.Parameter.AsInteger() Element.Parameter.Set(integer)	Valid values are 0 (Normal), 1 (Halftone) and 2 (Do not display)
Detail Level	VIEW_DETAIL_LEVEL	Valid for all plan views, all 3D views, Section, Elevation, Detail, DraftingView	View.DetailLevel property	Use also View.CanModifyDetailLevel() and View.HasDetailLevel()
Parts Visibility	VIEW_PARTS_VISIBILITY	Valid for all plan views, all 3D views, Section, Elevation, Detail	View.PartsVisibility property	
V/G Overrides Model	VIS_GRAPHICS_MODEL	Valid for all plan views, all 3D views, Section, Elevation, Detail, DraftingView	View.GetCategoryOverrides View.SetCategoryOverrides	
V/G Overrides Annotation	VIS_GRAPHICS_ANNOTATION	Valid for all plan views, all 3D views, Section, Elevation, Detail, DraftingView	View.GetCategoryOverrides View.SetCategoryOverrides	
V/G Overrides Analytical Model	VIS_GRAPHICS_ANALYTICAL_MODEL	Valid for all plan views, all 3D views, Section, Elevation, Detail, DraftingView	View.GetCategoryOverrides View.SetCategoryOverrides	

Parameter Name in View Template's Dialog	View Template Parameter Name	View Type Limitations	API getters/setters	Comments
V/G Overrides Import	VIS_GRAPHICS_IMPORT	Valid for all plan views, all 3D views, Section, Elevation, Detail, DraftingView	View.GetCategoryOverrides View.SetCategoryOverrides	Imported categories are not built-in, so they can be found using other API operations
V/G Overrides Filters	VIS_GRAPHICS_FILTERS	Valid for all plan views, all 3D views, Section, Elevation, Detail, DraftingView	View.GetFilters View.GetFilterOverrides View.SetFilterOverrides	
V/G Overrides Worksets	VIS_GRAPHICS_WORKSETS	Valid for all plan views, all 3D views, Section, Elevation, Detail, DraftingView	View.GetWorksetVisibility View.SetWorksetVisibility	Valid only if Document.IsWorkshared() is true Use also View.IsWorksetVisibility(♦)
V/G Overrides RVT Links	VIS_GRAPHICS_RVT_LINKS	Valid for all plan views, all 3D views, Section, Elevation, Detail	No API access at the time (Revit 2020)	Valid only if Revit links are present in the document
V/G Overrides Point Clouds	VIS_GRAPHICS_POINT_CLOUDS	Valid for all plan views, all 3D views, Section, Elevation, Detail, DraftingView	View.GetPointCloudOverrides View.SetPointCloudOverrides	Valid only if PointCloudInstance elements are present in the document Use also View.ArePointCloudsHidden()
V/G Overrides Coordination Model	VIS_GRAPHICS_COORDINATION_MODEL	Valid for all plan views, all 3D views, Section, Elevation, Detail, DraftingView	View.GetDirectContext3DHandleOverrides	Valid only if NavisWorks coordination models are linked in the document
V/G Overrides Design Options	VIS_GRAPHICS DESIGN_OPTIONS	Valid for all plan views, all 3D views, Section, Elevation,	No API access at the time (Revit 2020)	Valid only if design options other than ♦Main Model♦

Parameter Name in View Template's Dialog	View Template Parameter Name	View Type Limitations	API getters/setters	Comments
		Detail, DraftingView, all schedules		are present in the document
Model Display	GRAPHIC_DISPLAY_OPTIONS_MODEL	Valid for all plan views, all 3D views, Section, Elevation, Detail	View.GetViewDisplayModel View.SetViewDisplayModel	
Shadow	GRAPHIC_DISPLAY_OPTIONS_SHADOW	Valid for all plan views, all 3D views, Section, Elevation, Detail		No API access at the time (Revit 2020)
Sketchy Lines	GRAPHIC_DISPLAY_OPTIONS_SKETCHY_LINES	Valid for all plan views, all 3D views, Section, Elevation, Detail	View.GetSketchyLines View.SetSketchyLines	
Depth Cueing	GRAPHIC_DISPLAY_OPTIONS_FOG	Valid for all plan views, all 3D views, Section, Elevation, Detail	View.GetDepthCueing View.SetDepthCueing	Use also View.CanUseDepthCueing()
Lighting	GRAPHIC_DISPLAY_OPTIONS_LIGHTING	Valid for all plan views, all 3D views, Section, Elevation, Detail		Partial API access at the time (Revit 2020): View.ShadowIntensity property View.SunLightIntensity property
Photographic Exposure	GRAPHIC_DISPLAY_OPTIONS_PHOTO_EXPOSURE	Valid for all plan views, all 3D views, Section, Elevation, Detail		API access provided only for 3D views: View3D.GetRenderingSettings View3D.SetRenderingSettings
Background	GRAPHIC_DISPLAY_OPTIONS_BACKGROUND	Valid for all plan views, all 3D views, Section, Elevation, Detail	View.GetBackground View.SetBackground	
Far Clipping	VIEWER_BOUND_FAR_CIPPING	Section, Elevation, Detail	Element.Parameter.AsInteger() Element.Parameter.Set(integer)	

Parameter Name in View Template's Dialog	View Template Parameter Name	View Type Limitations	API getters/setters	Comments
Phase Filter	VIEW_PHASE_FILTER	Valid for all plan views, all 3D views, Section, Elevation, Detail, all schedules	Element.Parameter.AsElementId() Element.Parameter.Set(Element Id)	
Discipline	VIEW_DISCIPLINE	Valid for all plan views, all 3D views, Section, Elevation, Detail, DraftingView	View.Discipline property	
Show Hidden Lines	VIEW_SHOW_HIDDEN_LINES	Valid for all plan views, all 3D views, Section, Elevation, Detail	Element.Parameter.AsInteger() Element.Parameter.Set(integer)	Valid values are 0 (None), 1 (By Discipline), 2 (All)
Color Scheme Location	COLOR_SCHEME_LOCATION	Valid for all plan views except for ceiling, Section, Elevation, Detail	Element.Parameter.AsInteger() Element.Parameter.Set(integer)	Valid values are 0 (Foreground), 1 (Background)
Color Scheme	VIEW_SCHEMA_SETTING_FOR_BUILDING	Valid for all plan views except for ceiling, Section, Elevation, Detail	No API access at the time (Revit 2020)	
Underlay Orientation	VIEW_UNDERLAY_ORIENTATION	Valid for all plan views	ViewPlan.GetUnderlayOrientation ViewPlan.SetUnderlayOrientation	
View Range	PLAN_VIEW_RANGE	Valid for all plan views	ViewPlan.GetViewRange ViewPlan.SetViewRange	
Orientation	PLAN_VIEW_NORTH	Valid for all plan views	Element.Parameter.AsInteger() Element.Parameter.Set(integer)	Valid values are 0 (Project North), 1 (True North)
System Color Schemes	VIEW_SCHEMA_SETTING_FOR_SYSTEM_TEMPLATE	Valid for all plan views except for ceiling	No API access at the time (Revit 2020)	

Parameter Name in View Template's Dialog	View Template Parameter Name	View Type Limitations	API getters/setters	Comments
Depth Clipping	VIEW_BACK_CLIPPING	Valid for all plan views	Element.Parameter.AsInteger() Element.Parameter.Set(integer)	Valid values are 0 (No Clip), 1 (Clip With Line), 2 (Clip With No Line)
Rendering Settings	VIEWER3D_RENDER_SETTINGS	Valid for all 3D views	View3D.GetRenderingSettings View3D.SetRenderingSettings	
Visual Style	MODEL_GRAPHICS_STYLE_ANON_DRAFT	Valid for DraftingView	View.DisplayStyle property	Use also View.HasDisplayStyle()
Fields	SCHEDULE_FIELDS_PARAM	Valid for all schedules	ViewSchedule.Definition.GetField ViewSchedule.Definition.AddField ViewSchedule.Definition.RemoveField	
Filter	SCHEDULE_FILTER_PARAM	Valid for all schedules	ViewSchedule.Definition.GetFilter ViewSchedule.Definition.AddFilter ViewSchedule.Definition.RemoveFilter	
Sorting/Grouping	SCHEDULE_GROUP_PARAM	Valid for all schedules	ViewSchedule.Definition.GetSortGroupField ViewSchedule.Definition.AddSortGroupField ViewSchedule.Definition.RemoveSortGroupField	
Formatting	SCHEDULE_FORMAT_PARAM	Valid for all schedules	ViewSchedule.Definition.GetFieldId().IsHidden ViewSchedule.Definition.GetFieldId().ColumnHeading ViewSchedule.Definition.GetField	

Parameter Name in View Template s Dialog	View Template Parameter Name	View Type Limitations	API getters/setters	Comments
			Id().GetFormatOptions	
			ViewSchedule.Definition.GetFile Id().SetFormatOptions	
			etc.	
Appearance	SCHEDULE_SHEET_APPEARANCE_PARAM	Valid for all schedules	ViewSchedule.Definition.ShowTitle ViewSchedule.Definition.ShowHeaders	
			etc.	

2.6.10 Temporary Graphics

The `TemporaryGraphicsManager` class allows you to create temporary graphics in Revit views. The graphics created by this class are not subject to undo and are not saved. The Revit API developer should manage their lifetime, creation and destruction, though Revit will destroy all of them when closing the model.

The `AddControl` method creates a control in a specified view based on an instance of the `InCanvasControlData` class which defines the image path and location in model coordinates of the control. Additional functionality exists in the methods:

- `RemoveControl`
- `SetVisibility`
- `UpdateControl`

This code sample creates a temporary control at the center of a wall. The website www.autodesk.com is opened When the user clicks on the control. The `onclick` method is implemented in a server class that derives from `ITemporaryGraphicsHandler`.

Code Region: Creating a Temporary Control

```
private void TemporaryGraphicsControl(Wall wall)
{
    Document doc = wall.Document;
```

```
MultiServerService externalService = ExternalServiceRegistry.GetService(
    ExternalServices.BuiltInExternalServices.TemporaryGraphicsHandlerService)
as MultiServerService;

MyGraphicsService myGraphicsService = new MyGraphicsService();

externalService.AddServer(myGraphicsService);

externalService.SetActiveServers(
    new List<Guid> {myGraphicsService.GetServerId()});

TemporaryGraphicsManager mgr = TemporaryGraphicsManager.GetTemporaryGraphicsM
anager(doc);

XYZ controlPoint = ((LocationCurve)wall.Location).Curve.Evaluate(0.5, true);

InCanvasControlData data = new InCanvasControlData(
    @"C:/Autodesk/image32.bmp",
    controlPoint);

mgr.AddControl(data, doc.ActiveView.Id);

}

public class MyGraphicsService: ITemporaryGraphicsHandler
{
    public void OnClick(TemporaryGraphicsCommandData data)
    {
        Process.Start("https://www.autodesk.com");
    }

    public string GetName()
    {
        return "My Graphics Service";
    }

    public string GetDescription()
    {
        return "This is a graphics service";
    }

    public string GetVendorId()
```

```

    { return "ADSK"; }

    public ExternalServiceId GetServiceId()

    { return ExternalServices.BuiltInExternalServices.TemporaryGraphicsHandlerService; }

    public Guid GetServerId()

    { return new Guid("a8debc37-19fe-4198-1198-01a891ff1a7f"); }

}

```

2.7 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 section 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.

Note: An exception will be thrown if a transaction is started from an outside thread or outside modeless dialog. Transactions can only be started from supported API workflows, such as part of an external command, event, updater, or call-back.

2.7.1 Transaction Classes

All three transaction objects share some common methods:

Table 51: Common Transaction Object 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

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:

Table 52: TransactionStatus values

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 and is waiting for the end-user's response (in a modeless dialog). Once the failure processing is finished, automatically updated (to either Committed or RolledBack status).

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

```
public void CreatingModelLines(UIApplication uiApplication)
{
    Autodesk.Revit.DB.Document document = uiApplication.ActiveUIDocument.Document;
```

```
    Autodesk.Revit.ApplicationServices.Application application = uiApplication.Application;

    // Create a few geometry lines. These lines are not elements,
    // therefore they do not need to be created inside a document transaction.

    XYZ Point1 = XYZ.Zero;
    XYZ Point2 = new XYZ(10, 0, 0);
    XYZ Point3 = new XYZ(10, 10, 0);
    XYZ Point4 = new XYZ(0, 10, 0);

    Line geomLine1 = Line.CreateBound(Point1, Point2);
    Line geomLine2 = Line.CreateBound(Point4, Point3);
    Line geomLine3 = Line.CreateBound(Point1, Point4);

    // This geometry plane is also transaction and does not need a transaction
    XYZ origin = XYZ.Zero;
    XYZ normal = new XYZ(0, 0, 1);
    Plane geomPlane = Plane.CreateByNormalAndOrigin(normal, origin);

    // In order to a sketch plane with model curves in it, we need
    // to start a transaction because such operations modify the model.

    // All and any transaction should be enclosed in a 'using'
    // block or guarded within a try-catch-finally blocks
    // to guarantee that a transaction does not out-live its scope.

    using (Transaction transaction = new Transaction(document))
```

```
{  
    if (transaction.Start("Create model curves") == TransactionStatus.Started)  
    {  
        // Create a sketch plane in current document  
  
        SketchPlane sketch = SketchPlane.Create(document, geomPlane);  
  
        // Create a ModelLine elements using the geometry lines 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;  
  
        // Ask the end user whether the changes are to be committed or not  
        TaskDialog taskDialog = new TaskDialog("Revit");  
        taskDialog.MainContent = "Click either [OK] to Commit, or [Cancel] to Roll back the transaction.";  
        TaskDialogCommonButtons buttons = TaskDialogCommonButtons.Ok | TaskDialogCommonButtons.Cancel;  
        taskDialog.CommonButtons = buttons;  
  
        if (TaskDialogResult.Ok == taskDialog.Show())  
        {  
            // For many various reasons, a transaction may not be committed  
            // if the changes made during the transaction do not result a valid model.  
        }  
    }  
}
```

```
// If committing a transaction fails or is canceled by the end user,  
// the resulting status would be RolledBack instead of Committed.  
  
        if (TransactionStatus.Committed != transaction.Commit())  
        {  
            TaskDialog.Show("Failure", "Transaction could not be committed");  
        }  
    }  
    else  
    {  
        transaction.Rollback();  
    }  
}  
}  
}
```

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.

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 "Level and Grid".

Code Region 23-2: Combining multiple transactions into a TransactionGroup

```
public void CompoundOperation(Autodesk.Revit.DB.Document document)
{
    // All and any transaction group should be enclosed in a 'using' block or
    // guarded within
    // a try-catch-finally blocks to guarantee that the group does not out-ли
    // ve its scope.

    using (TransactionGroup transGroup = new TransactionGroup(document, "Leve
    l and Grid"))

    {
        if (transGroup.Start() == TransactionStatus.Started)

        {
            // We are going to call two methods, each having its own local tr
            // ansaction.

            // For our compound operation to be considered successful, both t
            // he individual

            // transactions must succeed. If either one fails, we will roll o
            // ur group back,
            // regardless of what transactions might have already been commit
            // ted.

            if (CreateLevel(document, 25.0) && CreateGrid(document, new XYZ
            (0,0,0), new XYZ(10,0,0)))
        }
    }
}
```

```
// The process of assimilating will merge the two (or any number of) committed

        // transaction together and will assign the grid's name to the one resulting transaction,
        // which will become the only item from this compound operation appearing in the undo menu.

        transGroup.Assimilate();

    }

    else

    {

        // Since we could not successfully finish at least one of the individual

        // operation, we are going to roll the entire group back, which will

        // undo any transaction already committed while this group was open.

        transGroup.Rollback();

    }

}

}

}

public bool CreateLevel(Autodesk.Revit.DB.Document document, double elevation)
{
    // All and any transaction should be enclosed in a 'using'

    // block or guarded within a try-catch-finally blocks

    // to guarantee that a transaction does not out-live its scope.

    using (Transaction transaction = new Transaction(document, "Creating Level"))
}
```

```
{  
    // Must start a transaction to be able to modify a document  
  
    if( TransactionStatus.Started == transaction.Start())  
    {  
        if (null != Level.Create(document, elevation))  
        {  
            // For many various reasons, a transaction may not be committed  
            // if the changes made during the transaction do not result a valid model.  
            // If committing a transaction fails or is canceled by the end user,  
            // the resulting status would be RolledBack instead of Committed.  
            return (TransactionStatus.Committed == transaction.Commit());  
        }  
  
        // For we were unable to create the level, we will roll the transaction back  
        // (although on this simplified case we know there weren't any other changes)  
  
        transaction.Rollback();  
    }  
}  
return false;  
}
```

```
public bool CreateGrid(Autodesk.Revit.DB.Document document, XYZ p1, XYZ p2)
{
    // All and any transaction should be enclosed in a 'using'
    // block or guarded within a try-catch-finally blocks
    // to guarantee that a transaction does not out-live its scope.

    using (Transaction transaction = new Transaction(document, "Creating Grid"))
    {
        // Must start a transaction to be able to modify a document
        if (TransactionStatus.Started == transaction.Start())
        {
            // We create a line and use it as an argument to create a grid
            Line gridLine = Line.CreateBound(p1, p2);

            if ((null != gridLine) && (null != Grid.Create(document, gridLine)))
            {
                if (TransactionStatus.Committed == transaction.Commit())
                {
                    return true;
                }
            }
        }

        // For we were unable to create the grid, we will roll the transaction back
        // (although on this simplified case we know there weren't any other changes)
    }
}
```

```

        transaction.Rollback();

    }

}

return false;
}

```

2.7.2 Transactions in Events

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.

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.

2.7.3 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.

ClearAfterRollback

This option controls whether all warnings should be cleared after a transaction is rolled back. The default value is False.

DelayedMiniWarnings

This option 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: This option is ignored in modal mode (see `ForcedModalHandling` below). **## ForcedModalHandling**

This option 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').

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 [Failure Posting and Handling](#) for more information.

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: Since the finalizer is called after the transaction has finished, the document is not modifiable from the finalizer unless a new transaction is started.

2.7.4 Getting Element Geometry and AnalyticalElement

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 AnalyticalElement 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 AnalyticalElement 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.

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 information, see [Analytical Element and Geometry](#).

The following sample program demonstrates how a transaction populates these properties:

Code Region 23-3: Transaction populating Geometry and AnalyticalPanel 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 = Line.CreateBound(start, end);

    // All and any transaction should be enclosed in a 'using'
    // block or guarded within a try-catch-finally blocks
    // to guarantee that a transaction does not out-live its scope.

    using (Transaction wallTransaction = new Transaction(document, "Creating wall"))
    {
        // To create a wall, a transaction must be first started
```

```

if (wallTransaction.Start() == TransactionStatus.Started)

{

    // Create a wall using the location line

    Wall wall = Wall.Create(document, geomLine, level.Id, true);

    // the transaction must be committed before you can

    // get the value of Geometry and AnalyticalPanel.

    if (wallTransaction.Commit() == TransactionStatus.Committed)

    {

        Autodesk.Revit.DB.Options options = uiApplication.Application.
Create.NewGeometryOptions();

        Autodesk.Revit.DB.GeometryElement geoelem = wall.get_Geometry
(options);

        Autodesk.Revit.DB.Structure.AnalyticalPanel analyticalPanel =
(AnalyticalPanel)document.GetElement(AnalyticalToPhysicalAssociationManager.G
etAnalyticalToPhysicalAssociationManager(document).GetAssociatedElementId(wal
l.Id));

    }

}

}

```

The transaction timeline for this sample is as follows:

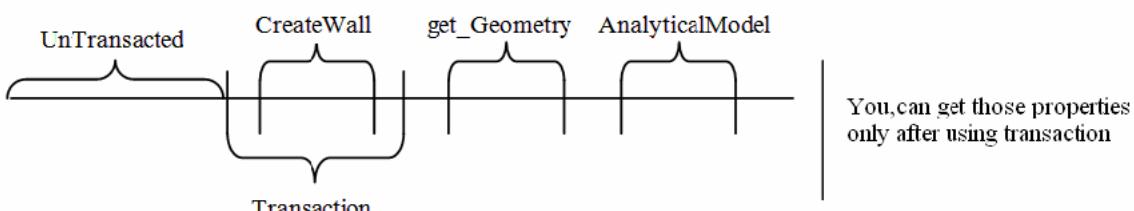


Figure 134: Transaction timeline

2.7.5 Temporary transactions

It is not always required to commit a transaction. The transaction framework also allows for Transactions to be rolled back. This is useful when there is an error during the processing of the transaction, but can also be leveraged directly as a technique to create a temporary transaction.

Using a temporary transaction can be useful for certain types of analyses. For example, an application looking to extract geometric properties from a wall or other object before it is cut by openings should use a temporary transaction in conjunction with Document.Delete(). When the application deletes the elements that cut the target elements, the cut element's geometry is restored to its original state (after the document has been regenerated).

To use a temporary transaction:

1. Instantiate the Transaction using the Transaction constructor, and assign it a name.
2. Call Transaction.Start()
3. Make the temporary change(s) to the document (element modification, deletion or creation)
4. Regenerate the document
5. Extract the desired geometry and properties
6. Call Transaction.RollBack() to restore the document to the previous state.

This technique is also applicable to SubTransactions.

3 Revit Geometric Elements

3.1 Walls, Floors, Ceilings, Roofs and Openings

Elements and the corresponding ElementTypes representing built-in place construction.

The following sections cover the classes associated with built-in place construction such as walls, floors, ceilings, roofs and openings and their corresponding properties.

3.1.1 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 33: Original wall

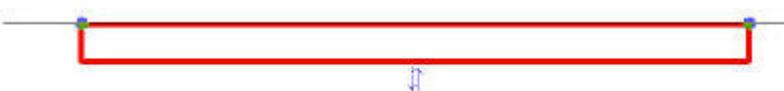


Figure 34: Wall after flip

Table 24: Wall Location Line

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

There are five static override methods in the Wall class to create a Wall:

Table 25: Create() Overrides

Name	Description
Create(Document, Curve, WallType, Level, Double, Double, Boolean, Boolean)	Creates a new rectangular profile wall within the project using the specified wall type, height, and offset.

Create(Document, IList Curve , Boolean)	Creates a non rectangular profile wall within the project using the default wall style.
Create(Document, Curve, ElementId, Boolean)	Creates a new rectangular profile wall within the project on the level specified by ElementId using the default wall style.
Create(Document, IList Curve , ElementId, ElementId, Boolean)	Creates a non rectangular profile wall within the project using the specified wall type.
Create(Document, IList Curve , ElementId, ElementId, Boolean, XYZ)	Creates a non rectangular profile wall within the project using the specified wall type and normal vector.

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:

Table 26: Wall Function

Wall Function	Interior	Exterior	Foundation	Retaining	Soffit
Value	0	1	2	3	4

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 [Structural Engineering](#).

Wall Profile Sketch

These methods provide access to add and remove profile sketches:

- `Wall.CreateProfileSketch()`
- `Wall.RemoveProfileSketch()`

`Wall.CanHaveProfileSketch()` returns True if the wall supports profile sketch. Arc walls and elliptical walls are two wall geometries that do not support having an edited profile. Once a sketch is added, the profile sketch can be edited using `SketchEditScope`

Slanted and Tapered Walls

`Wall.CrossSection` lets you get and set a wall to a value of the `WallCrossSection` enum (`SingleSlanted`, `Vertical`, `Tapered`). `Wall.IsWallCrossSectionValid()` checks if a cross section is valid for a specific wall.

`Autodesk.Revit.DB.WidthMeasuredAt` and `Autodesk.Revit.DB.InsertOrientation` define the shape of the wall and behavior of inserts for non-vertical walls.

Wall Wrapping

Wrapping for a specific wall end can be selectively enabled or disabled when the wall's end wrap is activated in the Wall Type dialog.

- `Wall.GetWrappingLocationAsReferences(int locationIndex)` - Gets an array of references to faces at the location.
- `Wall.GetWrappingLocationAsCurveParameter(int locationIndex)` - Gets the non-normalized (actual) curve parameter of the location.
- `Wall.GetValidWrappingLocationIndices()` - Gets all valid locations for end wrapping. The valid wrapping locations include the wall's two ends and locations on vertical faces of openings or profiles without joins.
- `AllowWrappingAtLocation(int locationIndex)` - Allows wrapping at either of the wall's ends (when using `locationIndex` = 0 or 1) or other wrapping locations at vertical faces of openings or profiles.
- `DisallowWrappingAtLocation(int locationIndex)` - Disallows wrapping at the specified location
- `IsWrappingAtLocationAllowed(int locationIndex)` - Indicates if wrapping is allowed at a specified location

3.1.2 Floors, Ceilings and Foundations

Classes associated with floors, ceilings and foundations.

Floor, Ceiling and Foundation-related API items include:

Table 28: Floors, Ceilings and Foundations in the API

Object	Element Type	ElementType Type	Element Creation	Other
Floor	Floor	FloorType	Floor.Create()	FloorType.IsFoundationSlab = false
Ceiling	Ceiling	CeilingType	Ceiling.Create()	Category = OST_Ceilings

Wall Foundation	WallFoundation	WallFoundationType	No	Category = OST_StructuralFoundation
Isolated Foundation	FamilyInstance	FamilySymbol	NewFamilyInstance()	Category = OST_StructuralFoundation
Foundation Slab	Floor	FloorType	Floor.Create()	Category = OST_StructuralFoundation FloorType.IsFoundationSlab = true

Note: Floor and Ceiling derive from the class CeilingAndFloor.

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 Floor.Create(), use the following methods:

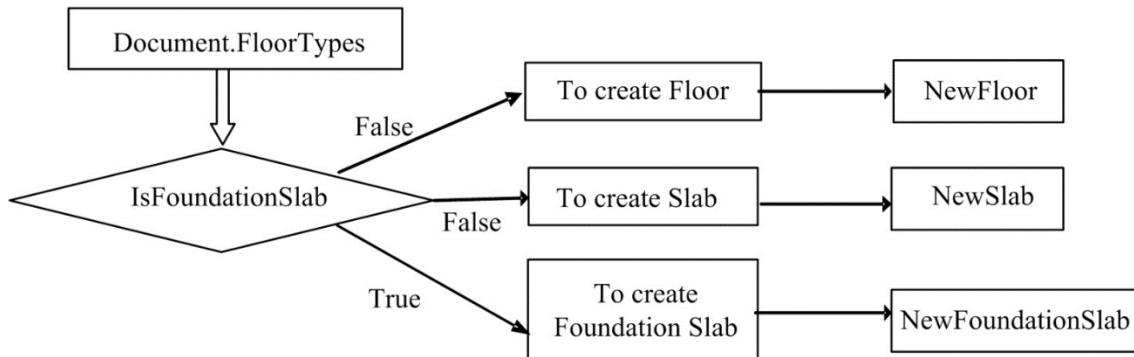


Figure 35: Create foundation and floor/slab

Currently, the API does not provide access to the Floor Slope Arrow in the Floor class. However, when using the structural features of Revit, you can create a sloped slab with Floor.Create():

Code Region 11-1: Floor.Create()

```

public static Floor Create(Document document, IList<CurveLoop> profile, ElementId floorTypeId, ElementId levelId)

public static Floor Create(Document document, IList<CurveLoop> profile, ElementId floorTypeId, ElementId levelId, bool isStructural, Line slopeArrow, double slope

)

```

The Slope Arrow is created using the `slopeArrow` parameter.

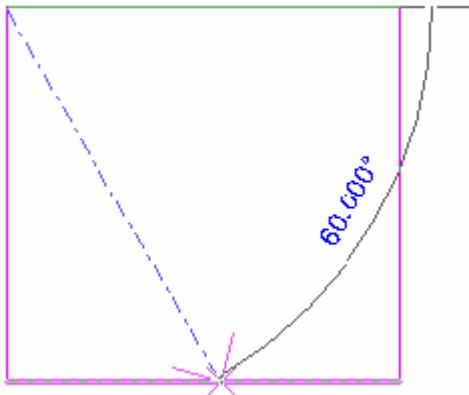


Figure 36: slopeArrow parameter in Floor.Create()

The unit for the slope parameter in `Floor.Create()` is rise/run.

The `Floor.FloorType` property is an alternative to using the `Floor.GetTypeId()` method. For more information about structure-related members such as the `GetSpanDirectionSymbolIds()` method and the `SpanDirectionAngle` property, refer to the [Structural Engineering](#) section.

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_OFFSET_PARAM`) parameter to control the foundation offset from the host Element.

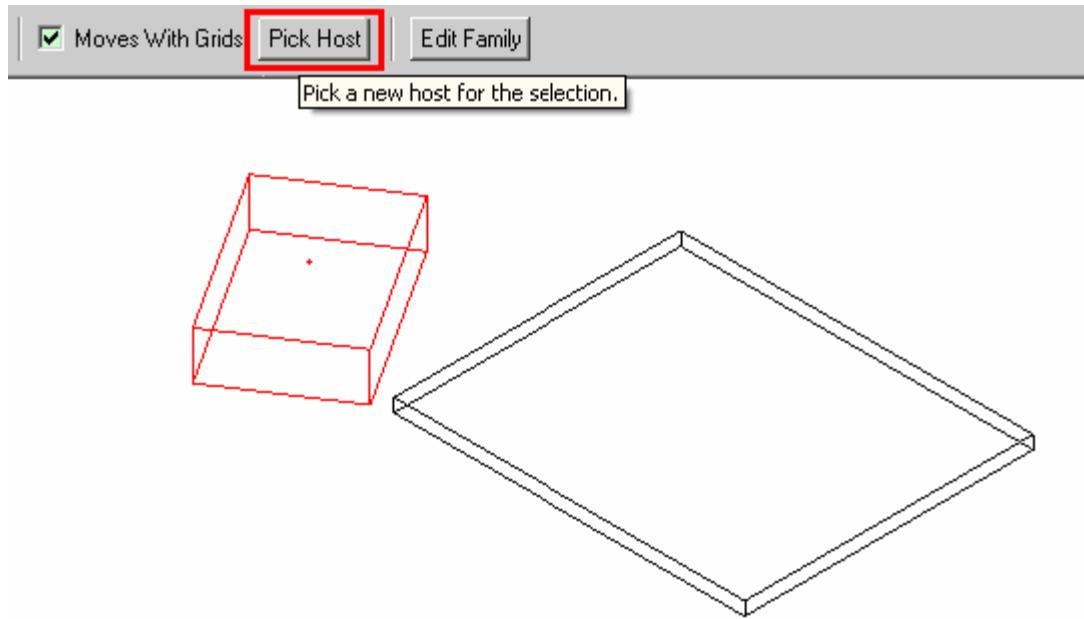


Figure 37: Pick Host for FoundationSlab (FamilyInstance)

Wall foundations are represented by the `WallFoundation` class in the API. The API provides limited access to both `WallFoundation` and `WallFoundationType` except when using the `GetAnalyticalModel()` method (refer to [Analytical Model](#) in the [Structural Engineering](#) section). For example, the attached wall is not available with the architectural features of Revit. Using the structural features of Revit, the relationship between the `Wall` class and the `WallFoundation` class is shown using the `GetAnalyticalModelSupports()` method in the `AnalyticalModel` class. For more details, refer to [Analytical Model](#) in the [Structural Engineering](#) section.

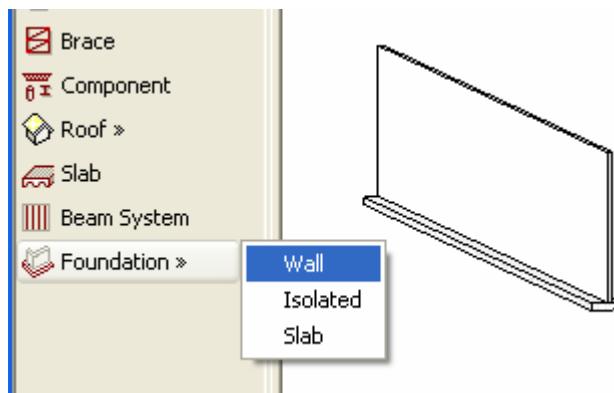


Figure 38: Wall Foundation

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;
    foreach (Autodesk.Revit.DB.Element elem in choices.GetElementIds().Select(q => document.GetElement(q)))
    {
        Floor floor = elem as Floor;
        if (floor != null)
        {
            SlabShapeEditor slabShapeEditor = floor.GetSlabShapeEditor();
            slabShapeEditor.ResetSlabShape();
        }
    }
}
```

For more detailed examples of using the SlabShapeEditor and related classes, see the SlabShapeEditing sample application included in the Revit SDK.

3.1.3 Roofs

Representation of roofs in the Revit API.

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

```
public void CreateRoof(Document document)
{
    // 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.OfClass(typeof(Level));

    var elements = from element in collector where element.Name == "Roof" select element;
    Level level = elements.Cast<Level>().ElementAt<Level>(0);

    collector = new FilteredElementCollector(document);
    collector.OfClass(typeof(RoofType));
    RoofType roofType = collector.FirstElement() as RoofType;

    // 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);

ICollection<ElementId> selectedIds = uidoc.Selection.GetElementIds();

if (selectedIds.Count != 0)

{

    foreach (ElementId id in selectedIds)

    {

        Element element = document.GetElement(id);

        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.ForwardIt
erator();

    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.

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(Document document)
```

```

{
    UIDocument uidoc = new UIDocument(document);

    foreach (Element elem in uidoc.Selection.GetElementIds().Select(q =>
document.GetElement(q)))
    {
        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");
        }
    }
}

```

3.1.4 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.

3.1.5 Other Elements

Some Elements are not HostObjects (and don't have a specific class), but are special cases that can host other objects. For example, ramp and its associated element type, do not have specific classes in the API and instead are represented as Element and ElementType in the OST_Ramps category.

3.1.6 CompoundStructure

Describes the internal structure of a wall, floor, roof or ceiling.

Walls, floors, ceilings and roofs are all children of the API class HostObject. HostObject (and its related type class HostObjAttributes) provide read only access to the CompoundStructure. A compound structure consists a collection of ordered layers, proceeding from exterior to interior for a wall, or from top to bottom for a floor, roof or ceiling. The properties of these layers

determine the thickness, material, and function of the overall structure of the associated wall, floor, roof or ceiling.

Layers can be accessed via the `GetLayers()` method and completely replaced using `SetLayers()`.

Normally these layers are parallel and extend the entire host object with a fixed layer width. However, for walls the structure can also be “vertically compound”, where the layers vary at specified vertical distances from the top and bottom of the wall. Use `CompoundStructure.IsVerticallyCompound` to identify these. For vertically compound structures, the structure describes a vertical section via a rectangle which is divided into polygonal regions whose sides are all vertical or horizontal segments. A map associates each of these regions with the index of a layer in the `CompoundStructure` which determines the properties of that region.

It is possible to use the compound structure to find the geometric location of different layer boundaries. The method `CompoundStructure.GetOffsetForLocationLine()` provides the offset from the center location line to any of the location line options (core centerline, finish faces on either side, or core sides).

With the offset to the location line available, you can obtain the location of each layer boundary by starting from a known location and obtaining the widths of each bounding layer using `CompoundStructure.GetLayerWidth()`.

Some notes about the use of `CompoundStructure`:

- The total width of the element is the sum of each `CompoundStructureLayer`'s widths. You cannot change the element's total width directly but you can change it via changing the `CompoundStructureLayer` width. The index of the designated variable length layer (if assigned) can be obtained from `CompoundStructure.VariableLayerIndex`.
- You must set the `CompoundStructure` back to the `HostObjAttributes` instance (using the `HostObjAttributes.SetCompoundStructure()` method) in order for any change to be stored.
- Changes to the `HostObjAttributes` affects every instance in the current document. If you need a new combination of layers, you will need to create a new `HostObjAttributes` (use `ElementType.Duplicate()`) and assign the new `CompoundStructure` to it.
- The `CompoundStructureLayer` `DeckProfileId`, and `DeckEmbeddingType`, properties only work with Slab in the structural features of Revit. For more details, refer to [Structural Engineering](#).

Material

Each `CompoundStructureLayer` in `HostObjAttributes` is typically displayed with some type of material. If `CompoundStructureLayer.MaterialId` returns -1, it means the Material is Category-related. For more details, refer to [Material](#). 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.GetCompoundStructure();
        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.GetLayers())
        {
            Autodesk.Revit.DB.Material layerMaterial =
                document.GetElement(structLayer.MaterialId) as Material;
        }
    }
}
```

```
        // If CompoundStructureLayer's Material is specified,  
use default  
  
        // Material of its Category  
        if (null == layerMaterial)  
  
        {  
            switch (structLayer.Function)  
            {  
                case MaterialFunctionAssignment.Finish:  
                    h1:  
                        layerMaterial =  
                            allCategories.get_Item(  
                                m(BuiltInCategory.OST_WallsFinish1).Material);  
                        break;  
  
                case MaterialFunctionAssignment.Finish:  
                    h2:  
                        layerMaterial =  
                            allCategories.get_Item(  
                                m(BuiltInCategory.OST_WallsFinish2).Material);  
                        break;  
  
                case MaterialFunctionAssignment.Member:  
                    hane:  
                        layerMaterial =  
                            allCategories.get_Item(  
                                m(BuiltInCategory.OST_WallsMembrane).Material);  
                        break;  
  
                case MaterialFunctionAssignment.Structure:  
                    ture:  
                        layerMaterial =  
                            allCategories.get_Item(  
                                m(BuiltInCategory.OST_WallsStructure).Material);  
            }  
        }  
    }  
}
```

```
        break;

    case MaterialFunctionAssignment.Substrate:
        layerMaterial =
            allCategories.get_Item(
                m(BuiltInCategory.OST_WallsSubstrate).Material);
        break;

    case MaterialFunctionAssignment.Insulation:
        layerMaterial =
            allCategories.get_Item(
                m(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);
}
```

{}

Sometimes just the material from the "structural" layer is needed. Rather than looking at each layer for the one whose function is `MaterialFunctionAssignment.Structure`, use the `CompoundStructure.StructuralMaterialIndex` property to find the index of the layer whose material defines the structural properties of the type for the purposes of analysis.

Note: When calling `SetLayers()`, the `StructuralMaterialIndex` value will be cleared and need to be reset.

3.1.7 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.

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 [Geometry](#).
- `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: 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.IntegerValue;

    //get the information whether the opening has a rect boundary
    //If the opening has a rect boundary, we can get the geometry information 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
        IList<XYZ> boundaryRect = opening.BoundaryRect;

        //get the coordinate value of the min coordinate point
        XYZ point = opening.BoundaryRect[0];
        message += "\nMin coordinate point:(" + point.X + ", "
                    + point.Y + ", " + point.Z + ")";

        //get the coordinate value of the Max coordinate point
        point = opening.BoundaryRect[1];
        message += "\nMax coordinate point: (" + point.X + ", "
                    + point.Y + ", " + point.Z + ")";
    }
}
```

```
        + 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.DB.Curve curve = opening.BoundaryCurves.get_Item(i);

            // Get curve start point

            message += "\nCurve start point: " + XYZToString(curve.GetEndPoint(0));

            // Get curve end point

            message += "; Curve end point: " + XYZToString(curve.GetEndPoint(1));

        }

        TaskDialog.Show("Revit", message);

    }

    // output the point's three coordinates

    string XYZToString(XYZ point)

    {

        return "(" + point.X + ", " + point.Y + ", " + point.Z + ")";
    }
}
```

{

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 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:

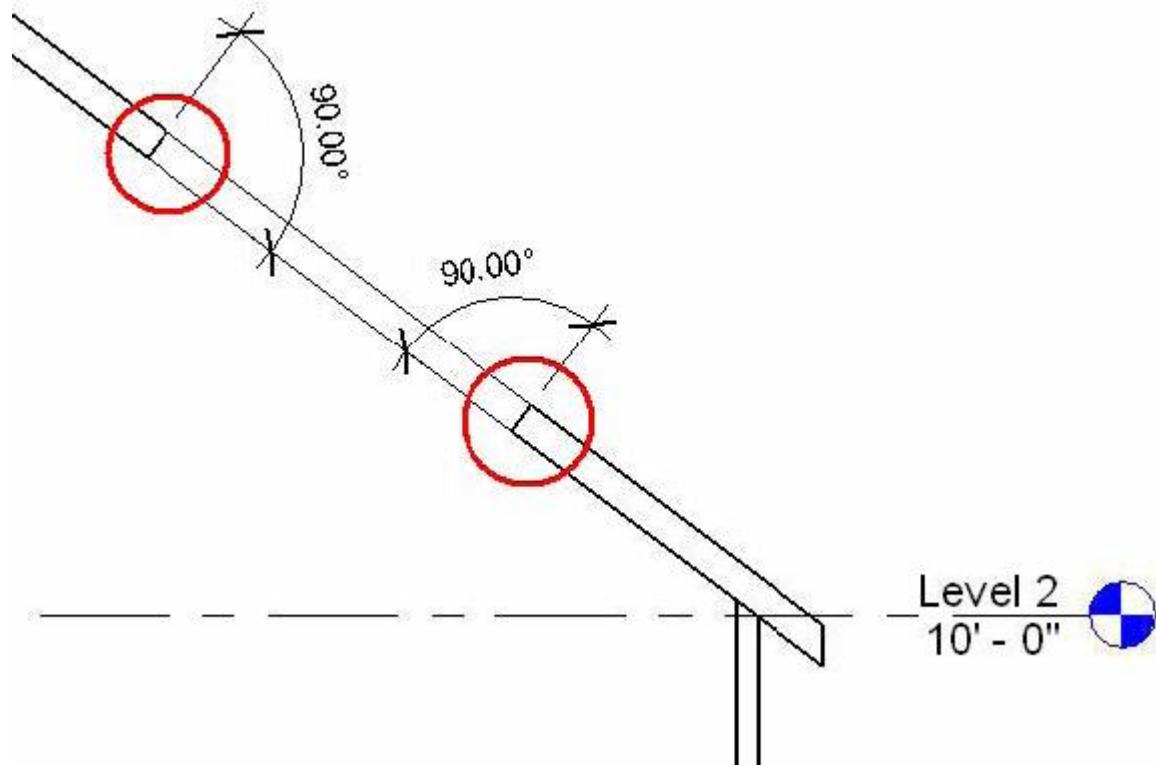


Figure 39: Opening cut perpendicular to the host element face

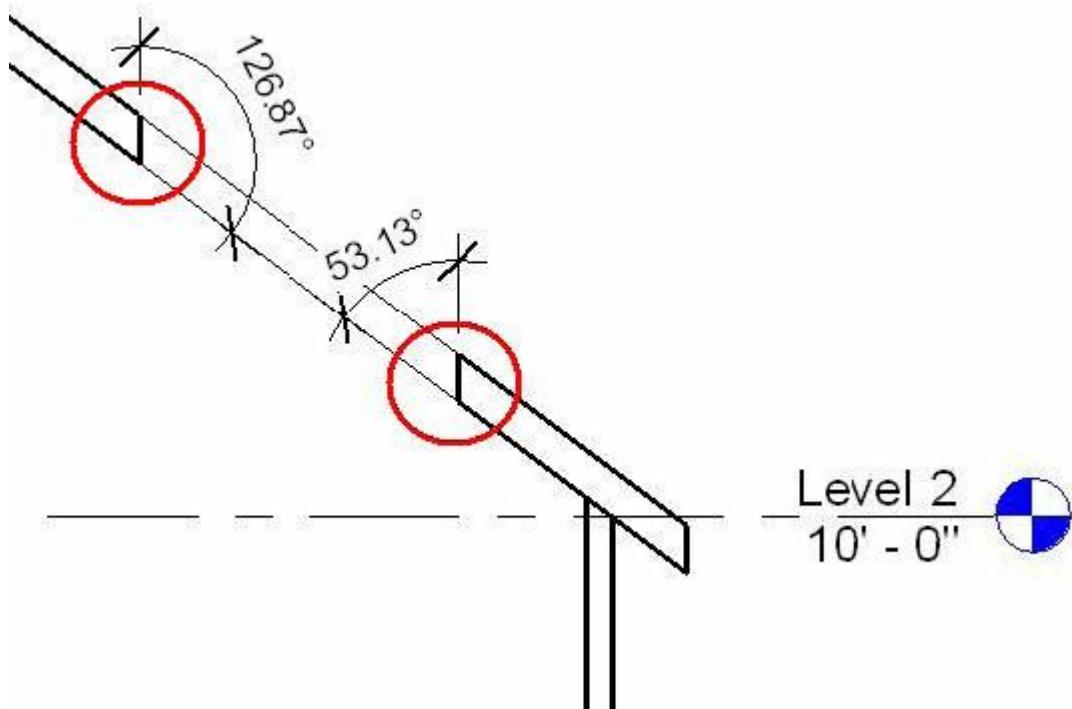


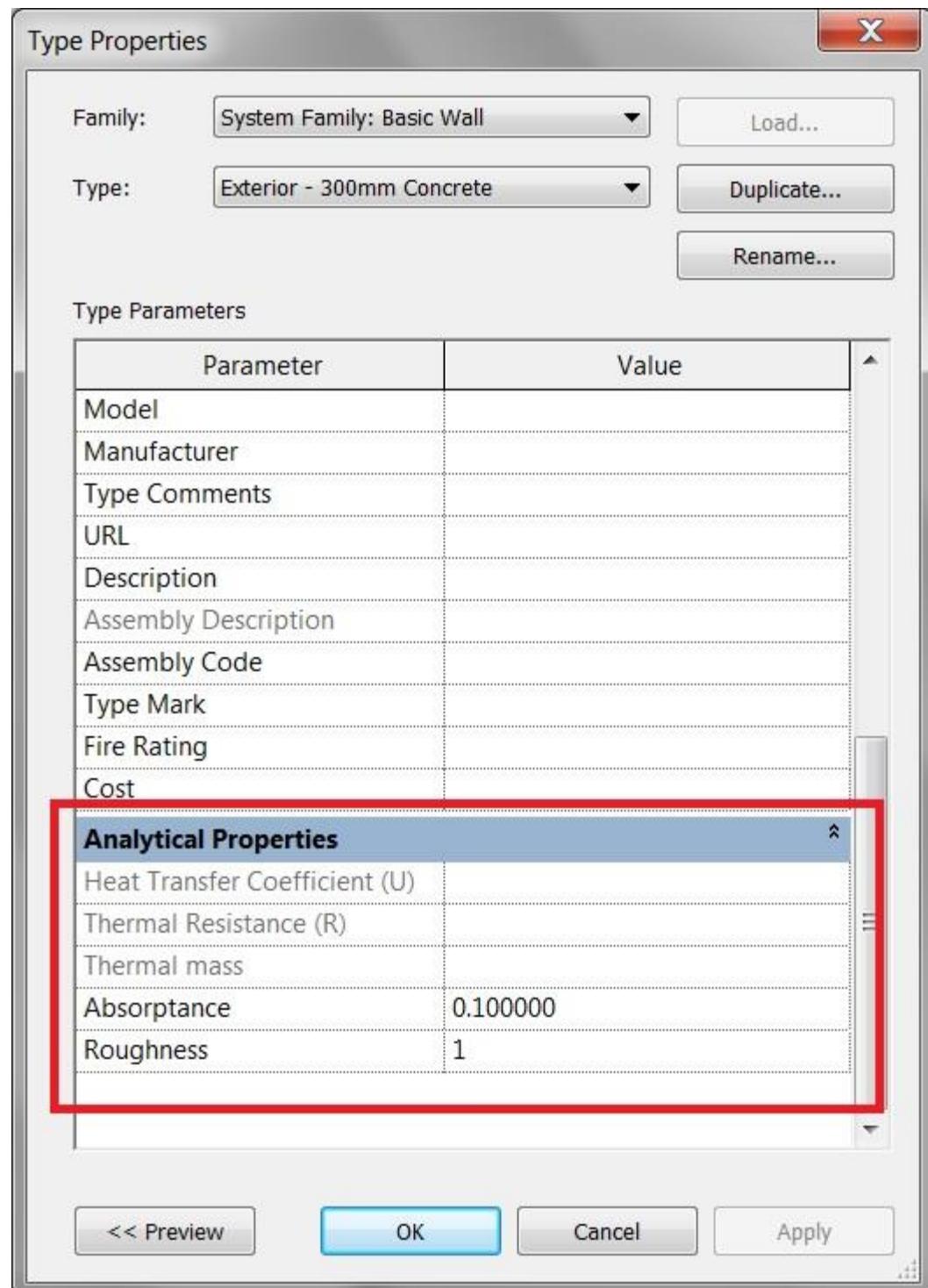
Figure 40: Opening cut vertically to the host element

- 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.

3.1.8 Thermal Properties

Certain assembly types such as Wall, Floor, Ceiling, Roof and Building Pad have calculated and settable thermal properties which are represented by the ThermalProperties class.



The ThermalProperties class has properties for the values shown above. Absorptance and Roughness are modifiable while HeatTransferCoefficient, ThermalResistance, and ThermalMass are read-only. The units for these calculated values are shown in the table below.

Property	Unit
----------	------

HeatTransferCoefficient	watts per meter-squared kelvin (W/(m ² *K))
ThermalResistance	meter-squared kelvin per watt (m ² *K)/Watt)
ThermalMass	kilogram feet-squared per second squared kelvin (kg ft ² /(s ² K))

Thermal properties can be retrieved using the ThermalProperties property on the following types:

- WallType
- FloorType
- CeilingType
- RoofType
- BuildingPadType

3.2 Family Instances

In this section, 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
- The relationship between family instance and family symbol

3.2.1 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.

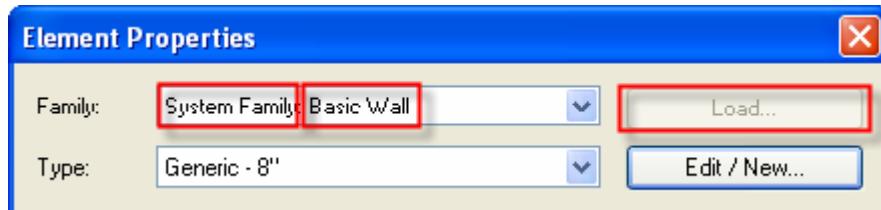


Figure 41: System Family

- 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 42: Component Family

- There are some exceptions, for example: Mass and in-place member. The Family and Type fields are blank.



Figure 43: 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" x 80".

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" x 80".
- Each FamilySymbol belongs to one Family. The 34" x 80" symbol belongs to a Single-Flush family.
- Each Family contains one or more FamilySymbols. The Single-Flush family contains a 34" x 80" symbol, a 34" x 84" symbol, a 36" x 84" and so on.

Note: 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.

3.2.2 Family

The Family class represents an entire Revit family. It contains the FamilySymbols used by FamilyInstances.

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 GetLibraryPaths() method.
- 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 [Code Samples](#).

Categories

The Family.FamilyCategory property indicates the category of the Family such as Columns, Furniture, Structural Framing, or Windows.

3.2.3 FamilyInstances

FamilyInstance objects include Beams, Braces, Columns, Furniture, Massing, and more. The FamilyInstance object provides more detailed properties so that the family instance type and appearance in the project can be changed.

Location-Related Properties

Location-related properties show the physical and geometric characteristics of FamilyInstance objects, such as orientation, rotation and location.

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.

Table 29: Compare Family Instances

Boolean Property	Door	Window (Fixed: 36"w × 72"h)	Desk
CanFlipFacing	True	True	False
CanFlipHand	True	False	False

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.

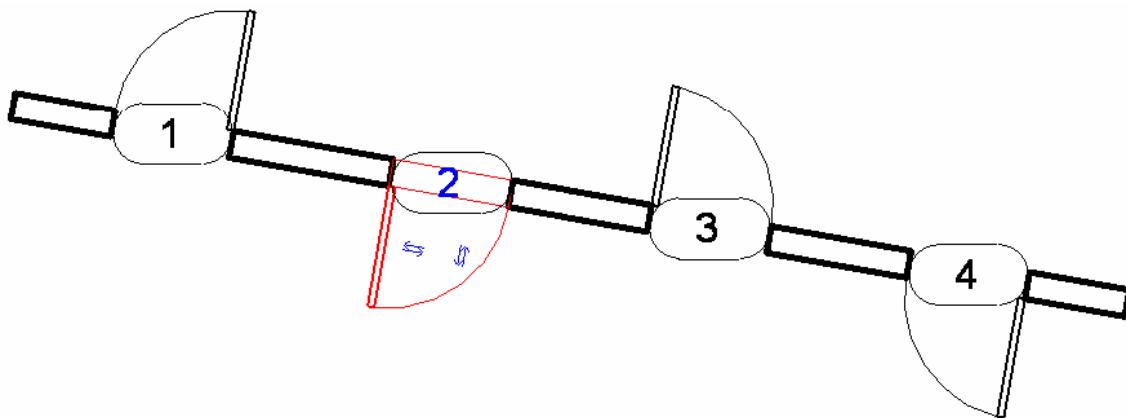


Figure 44: Doors with different Facing and Hand Orientations

Table 30: Different Instances of the Same Type

Boolean Property	Door 1	Door 2	Door 3	Door 4
FacingFlipped	False	True	False	True
HandFlipped	False	True	True	False

Orientation - Work Plane

The work plane orientation for a FamilyInstance can be changed, as well. If CanFlipWorkPlane is true, you can set the IsWorkPlaneFlipped property. Attempting to set this property for a FamilyInstance that does not allow the work plane to be flipped will result in an exception.

Rotation - Mirrored

The Mirrored property indicates whether the FamilyInstance object has been mirrored.

Table 31: Door Mirrored Property

Boolean Property	Door 1	Door 2	Door 3	Door 4
Mirrored	False	False	True	True

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.

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 3×3 with Trim: 36"×72") and Door 1 (Double-Glass 2: 72"×82") are true, while Window 2 (Fixed: 36"w × 72"h) is false.

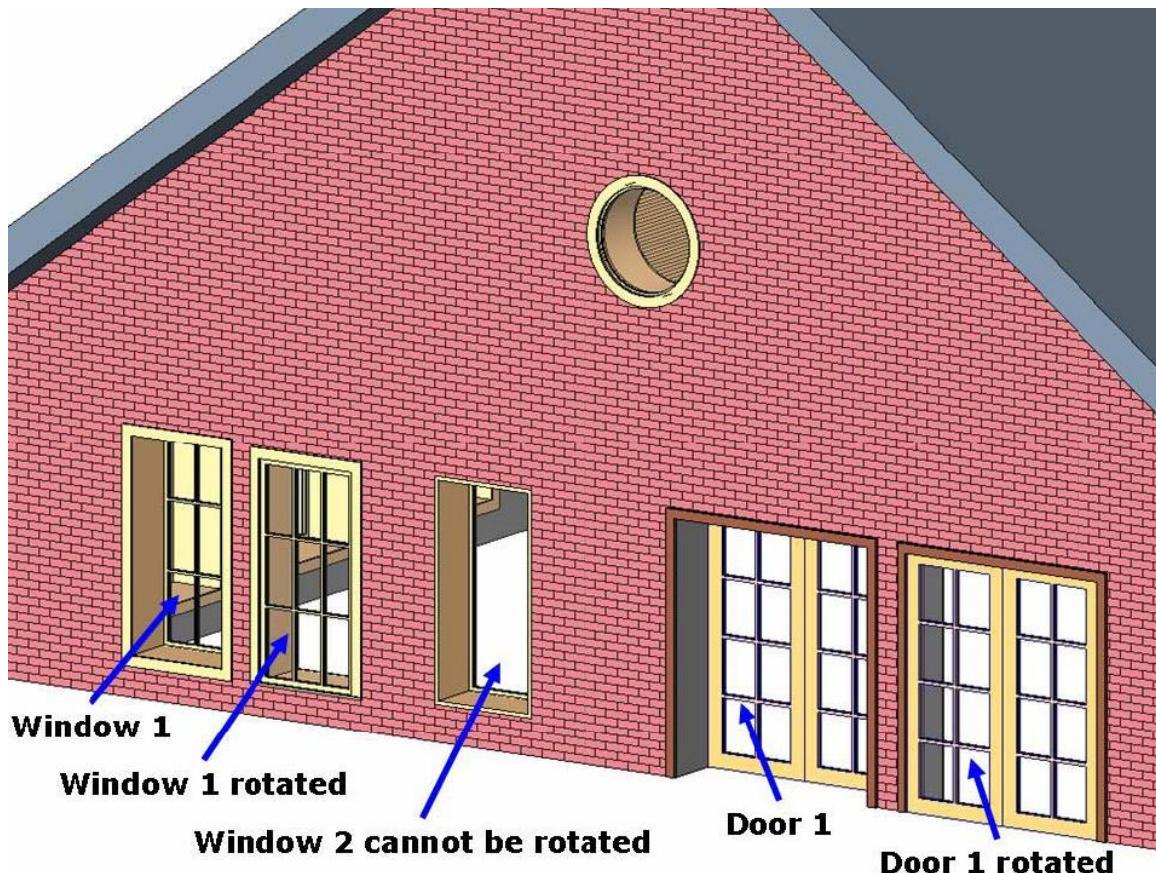


Figure 45: Changes after rotate()

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 `ElementTransformUtils.RotateElement()` and `ElementTransformUtils.RotateElements()`.

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 [Editing Elements](#).

Host and HostFace

Host and HostFace are both FamilyInstance properties.

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.

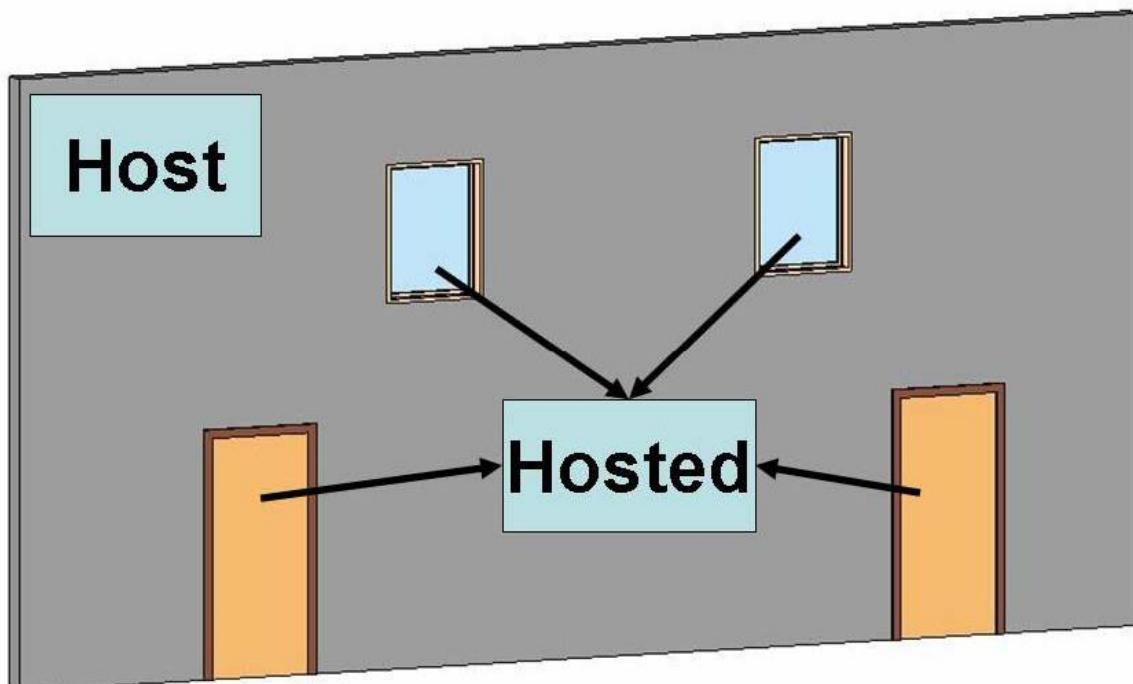


Figure 46: Doors and windows hosted in a wall

HostFace The HostFace property gets the reference to the host face of the family instance, or if the instance is placed on a work plane, the reference to the geometry face underlying the work plane. This property will return a null reference if the work plane is not referencing other geometry, or if the instance is not hosted on a face or work plane.

Subcomponent and Supercomponent

The FamilyInstance.GetSubComponentIds() method returns the ElementIds of family instances loaded into that family. When an instance of 'Table-Dining Round w Chairs.rfa' is placed in a project, the ElementIds of the set of chairs are returned by the GetSubComponentIds() method.

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)
{
    ICollection<ElementId> subElemSet = familyInstance.GetSubComponentIds();

    if (subElemSet != null)
    {
        string subElems = "";
        foreach (Autodesk.Revit.DB.ElementId ee in subElemSet)
        {
            FamilyInstance f = familyInstance.Document.GetElement(ee) as FamilyInstance;
            subElems = subElems + f.Name + "\n";
        }
        TaskDialog.Show("Revit", "Subcomponent count = " + subElemSet.Count +
"\n" + subElems);
    }

    FamilyInstance super = familyInstance.SuperComponent as FamilyInstance;
    if (super != null)
    {
        TaskDialog.Show("Revit", "SUPER component: " + super.Name);
    }
}

```

Geometry

Sometimes the geometry of a FamilyInstance object is modified by joins, cuts, coping, extensions, or other post-processing that occurs in Revit. The FamilyInstance.HasModifiedGeometry() method identifies if the geometry of this FamilyInstance has been modified from the automatically generated default. The GetOriginalGeometry() method will return the original geometry of the instance prior to any modifications that may have

occurred. To get the current geometry of the instance, use the Geometry property inherited from the Element class.

Spatial Element Calculation Points

The FamilyInstance has several members for reading the information about spatial calculation point(s) directly from the family instance. The HasSpatialElementCalculationPoint property identifies if this instance has a single SpatialElementCalculationPoint used as the search point for Revit to identify if the instance is inside a room or space. If true, the GetSpatialElementCalculationPoint() method will return the location of the calculation point for this instance as an XYZ point.

The HasSpatialElementFromToCalculationPoints property identifies if this instance has a pair of SpatialElementCalculationPoints used as the search points for Revit to identify if the instance lies between up to two rooms or spaces. The points determine which room or space is considered the "from" and which is considered the "to" for a family instance which connects two rooms or spaces, such as a door or window. When this property is true, the GetSpatialElementFromToCalculationPoints() method returns the locations for the calculation points for this instance as a list of XYZ points.

Other Properties

The properties in this section are specific to the architectural and structural engineering features of Revit. They are covered thoroughly in their respective chapters.

Room Information

FamilyInstance properties include Room, FromRoom, and ToRoom. For more information about Room, refer to [Architecture](#).

Space Information

FamilyInstance has a Space property for identifying the space that holds an instance in MEP.

Structural Analytical Model

The GetAnalyticalModel() method retrieves the family instance structural analytical model.

For more information about AnalyticalModel refer to [Structural Engineering](#).

Creating FamilyInstance Objects

Typically a FamilyInstance object is created using one of the twelve 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 32 - 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 33 - 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.

All overloads for NewFamilyInstance() check to ensure that the input FamilySymbol is active (FamilySymbol.IsActive). If the input FamilySymbol is inactive, the method will throw an ArgumentException. Symbols that are not used in the document may be deactivated to conserve memory and regeneration time. When the symbol is inactive, its geometry is empty and cannot be accessed. In order to access the geometry of a symbol that is not active in the document, the symbol should first be activated by calling FamilySymbol.Activate().

Table 32 - Options for creating instance with NewFamilyInstance()

Category	NewFamilyInstance() parameters	Comments
Air Terminals	XYZ, FamilySymbol, StructuralType	Creates the instance in an arbitrary location without reference to a level or host element.
Boundary Conditions	XYZ, FamilySymbol, Element, StructuralType	If it is to be hosted on a wall, floor or ceiling
Casework		
Communication Devices		
Data Devices		
Electrical Equipment		
Electrical Fixtures		
Entourage		
Fire Alarm Devices		
Furniture		
Furniture Systems		
Generic Models		
Lighting Devices		
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		

Category	NewFamilyInstance() parameters	Comments
	Reference, XYZ, XYZ, FamilySymbol	If it is face-based and needs to be oriented in a non- default direction, accepts a reference to a face rather than a Face
	Face, Line, FamilySymbol	If it is face-based and linear
	Reference, Line, FamilySymbol	If it is face-based and linear, but accepts a reference to a face, rather than a Face
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.
	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.
Doors Windows	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

Category	NewFamilyInstance() parameters	Comments
	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
Detail Component	Line, FamilySymbol, View	Applies only to 2D family line based detail symbols
Generic Annotations	XYZ, FamilySymbol, View	Applies only to 2D family symbols

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

```
IICollection<ElementId> BatchCreateColumns(Autodesk.Revit.DB.Document document, Level level)
{
    List<Autodesk.Revit.Creation.FamilyInstanceCreationData> fiCreationDatas = new List<Autodesk.Revit.Creation.FamilyInstanceCreationData>();

    IICollection<ElementId> 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);
```

```
        Autodesk.Revit.Creation.FamilyInstanceCreationData fi
CreationData = new Autodesk.Revit.Creation.FamilyInstanceCreationData(location, familySymbol, level,
                                                                    StructuralType.Column);

        if (null != fiCreationData)
        {
            fiCreationDatas.Add(fiCreationData);
        }
    }

    if (fiCreationDatas.Count > 0)
    {
        // Create Columns

        elementSet = document.Create.NewFamilyInstances2(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.

Table 33 - Options for creating instances with other methods

Category	Creation method	Comments
Air Terminal Tags	IndependentTag.Create(Document, ElementId, Reference, 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
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		
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	<code>IndependentTag.Create(Document, ElementId, Reference, Boolean, TagMode, TagOrientation, XYZ)</code>	TagMode should be <code>TM_ADDBY_MATERIAL</code> and there should be a material tag family loaded, otherwise exception will be thrown
Multi-Category Tags	<code>IndependentTag.Create(Document, ElementId, Reference, Boolean, TagMode, TagOrientation, XYZ)</code>	TagMode should be <code>TM_ADDBY_MULTICATEGORY</code> , and there should be a multi-category tag family loaded, otherwise exception will be thrown
Title Blocks	<code>ViewSheet.Create(Document, ElementId)</code>	The titleblock will be added to the newly created sheet.

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 [Code Samples](#).

3.2.4 Code Samples

Code samples for working with Family Instances.

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:

Table 34: The value of StructuralType argument in the NewFamilyInstance() method

Type of Family Instance	Value of StructuralType
Doors, tables, etc.	NonStructural
Beams	Beam
Braces	Brace
Columns	Column
Footings	Footing

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

```
void CreateTables(Autodesk.Revit.DB.Document document)
{
    String fileName = @"C:\ProgramData\Autodesk\RVT 2014\Libraries\US Imperial\Furniture\Tables\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
ISet<ElementId> familySymbolIds = family.GetFamilySymbolIds();
double x = 0.0, y = 0.0;
foreach (ElementId id in familySymbolIds)
{
    FamilySymbol symbol = family.Document.GetElement(id) as FamilySymbol;
    XYZ location = new XYZ(x, y, 10.0);

    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:

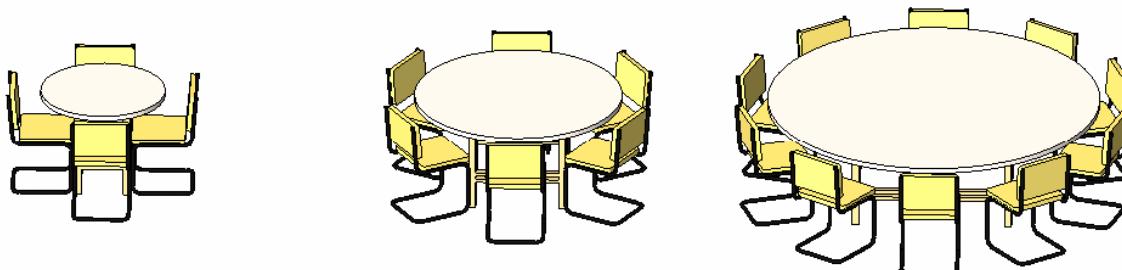


Figure 47: Load family and create tables in the Revit project

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

```
FamilyInstance CreateBeam(Autodesk.Revit.DB.Document document, View view)

{

    // get the given view's level for beam creation
    Level level = document.GetElement(view.LevelId) as Level;

    // get a family symbol
    FilteredElementCollector collector = new FilteredElementCollector(document);
    collector.OfClass(typeof(FamilySymbol)).OfCategory(BuiltInCategory.OST_StructuralFraming);

    FamilySymbol gotSymbol = collector.FirstElement() as FamilySymbol;

    // create new beam 10' long starting at origin
    XYZ startPoint = new XYZ(0, 0, 0);
    XYZ endPoint = new Autodesk.Revit.DB.XYZ(10, 0, 0);

    Autodesk.Revit.DB.Curve beamLine = Line.CreateBound(startPoint, endPoint);

    // create a new beam
    FamilyInstance instance = document.Create.NewFamilyInstance(beamLine, gotSymbol, level, StructuralType.Beam);
```

```
    return instance;  
}
```

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)  
{  
    // get wall's level for door creation  
    Level level = document.GetElement(wall.LevelId) as Level;  
  
    FilteredElementCollector collector = new FilteredElementCollector(document);  
  
    ICollection<Element> collection = collector.OfClass(typeof(FamilySymbol))  
        .OfCategory(BuiltInCategory.OST_Doors)  
        .ToElements();  
  
    IEnumrator<Element> symbolItor = collection.GetEnumerator();  
  
    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.

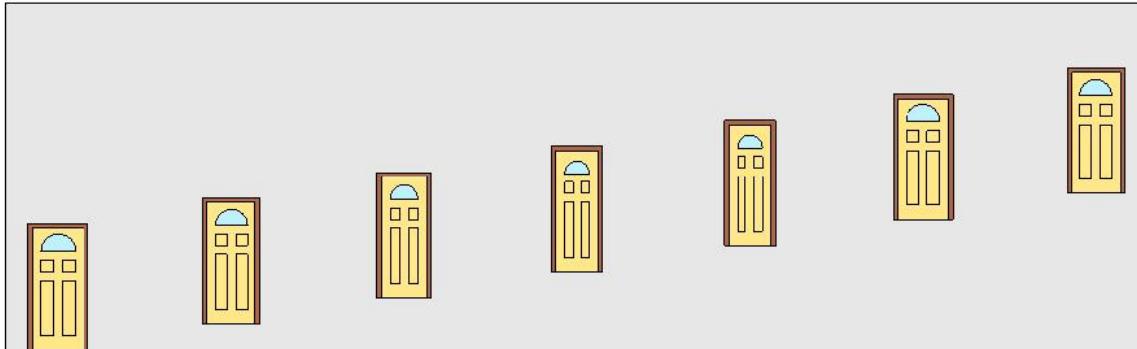


Figure 48: Insert doors into a wall

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

```

public void CreateWithRefDir(Document document)
{
    // Get a floor to place the beds
    FilteredElementCollector collector = new FilteredElementCollector(document);
    Floor floor = collector.OfClass(typeof(Floor)).FirstElement() as Floor;
}

```

```
if (floor != null)

{
    // Find a Bed-Box family

    Family family = null;

    FilteredElementCollector famCollector = new FilteredElementCollector
(document);

    famCollector.OfClass(typeof(Family));

    ICollection<Element> collection = famCollector.ToElements();

    foreach (Element element in collection)

    {

        if (element.Name.CompareTo("Bed-Box") == 0)

        {

            family = element as Family;

            break;

        }

    }

    if (family != null)

    {

        // Enumerate the beds in the Bed-Box family

        FilteredElementCollector fsCollector = new FilteredElementCollect
or(document);

        ICollection<Element> fsCollection = fsCollector.WherePasses(new F
amilySymbolFilter(family.Id)).ToElements();

        IEnumarator<Element> symbolItor = fsCollection.GetEnunimator();

        int x = 0, y = 0;
```

```
int i = 0;

while (symbolIator.MoveNext())
{
    FamilySymbol symbol = symbolIator.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++;
}

}
```

The result of the previous code appears in the following picture:

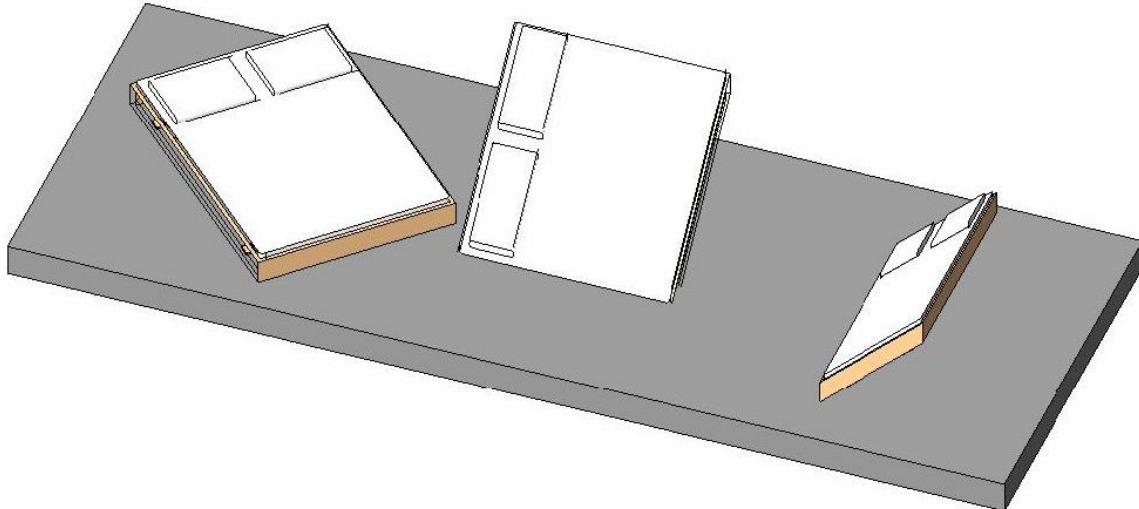


Figure 49: Create family instances using different reference directions

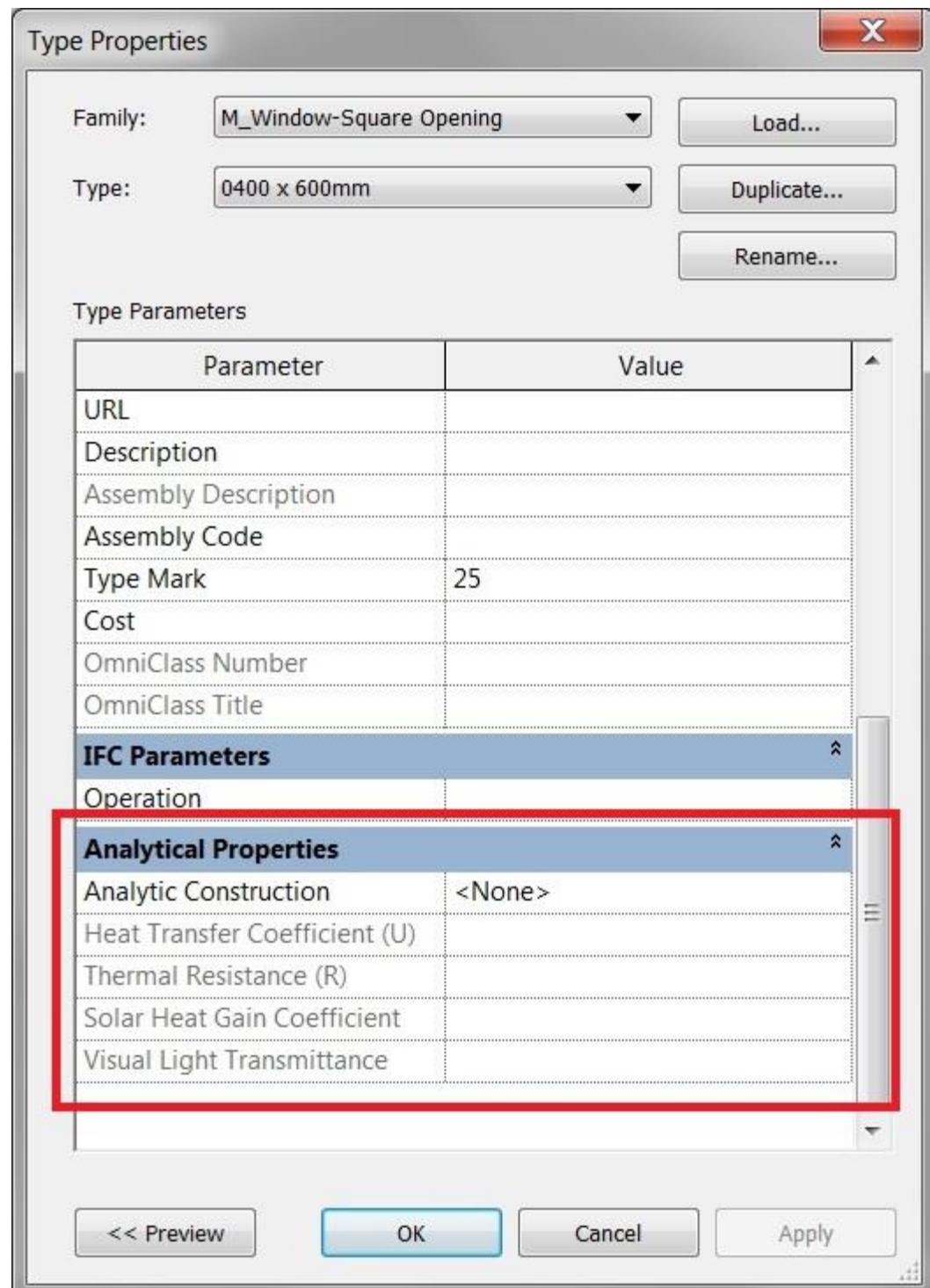
3.2.5 FamilySymbol

The FamilySymbol class represents a single Type within a Family.

Each family can contain one or more family symbols. Each FamilyInstance has an associated FamilySymbol which can be accessed from its Symbol property.

Thermal Properties

Certain types of families (doors, windows, and curtain wall panels) contain thermal properties as shown in the Type Properties window below for a window.



The thermal properties for a FamilySymbol are represented by the FamilyThermalProperties class and are retrieved using the FamilySymbol.GetThermalProperties() method. The FamilyThermalProperties for a FamilySymbol can be set using SetThermalProperties(). The properties of the FamilyThermalProperties class itself are read-only.

The units for the calculated values are shown in the table below.

Property	Unit
HeatTransferCoefficient	watts per meter-squared kelvin (W/(m ² *K))
ThermalResistance	meter-squared kelvin per watt (m ² K/Watt)

The AnalyticConstructionTypeId property is the construction gbXML type and returns the value that corresponds to the 'id' property of a constructionType node in Constructions.xml. The static FamilyThermalProperties.Find() method will find the FamilyThermalProperties by the 'id' property of a constructionType node in Constructions.xml.

FamilyType Parameters

Some parameters for a FamilySymbol may be FamilyType parameters. For these parameters, the Family.GetFamilyTypeParameterValues() method can be used to get all applicable values for the parameter. The values returned are ElementIds of all family types that match the category specified by the definition of the given parameter. The elements are either of class ElementType or NestedFamilyTypeReference. The second variant is for the types that are nested in families and therefore not accessible otherwise. The NestedFamilyTypeReference element stores only basic information about the nested FamilyType, such as the name of the Type, name of the Family, and a Category. These elements are very low-level and thus bypassed by standard element filters, so the main way to get them is via the Family.GetFamilyTypeParameterValues() method.

The following example demonstrates how to get all the family type parameter values for a FamilyType parameter of a FamilySymbol. The value for the parameter is then changed to another value. This change affects all FamilyInstances using the loaded FamilySymbol.

Code Region: Get nested FamilyTypes

```
public void GetNestedFamilyTypes(FamilyInstance instance)
{
    // find one FamilyType parameter and all values applicable to it
    Parameter aTypeParam = null;
    ISet<ElementId> values = null;

    Family family = instance.Symbol.Family;
```

```
foreach (Parameter param in instance.Symbol.Parameters)
{
    ForgeTypeId forgeTypeId = param.Definition.GetDataType();
    if (Category.IsBuiltInCategory(forgeTypeId))
    {
        aTypeParam = param;
        values = family.GetFamilyTypeParameterValues(param.Id);
        break;
    }
}

if (aTypeParam == null)
{
    TaskDialog.Show("Warning", "The selected family has no FamilyType parameter defined.");
}
else if (values == null)
{
    TaskDialog.Show("Error", "A FamilyType parameter does not have any applicable values!?");
}
else
{
    ElementId newValue = values.Last<ElementId>();
    if (newValue != aTypeParam.AsElementId())
    {

```

```
        using (Transaction trans = new Transaction(instance.Document, "Se
tting parameter value"))

    {

        trans.Start();

        aTypeParam.Set(newValue);

        trans.Commit();

    }

}

}

}
```

3.3 Family Documents

This section discusses families and how to:

- Create and modify Family documents
- Access family types and parameters

3.3.1 About family documents

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.

Categories

As noted in the previous section, the Family.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

```
public void GetName(Document familyDoc)
{
    string categoryName = familyDoc.OwnerFamily.FamilyCategory.Name;
}
```

The FamilyCategory can also be set, allowing the category of a family that is being edited to be changed.

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

```
public void GetFamilyDocumentCategory(Document familyDoc)
{
    // get the owner family of the family document.
    Family family = familyDoc.OwnerFamily;

    Parameter param = family.GetParameter(ParameterTypeId.FamilyWorkPlane
Based);

    // this param is a Yes/No parameter in UI, but an integer value in AP
I

    // 1 for true and 0 for false

    int isTrue = param.AsInteger();

    // param.Set(1); // set value to true.

}
```

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

```
public void CreateFamily(Application application)
{
    // 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");
    }
}
```

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)).ToList();
        foreach (Element e in collection)
        {
            FamilySymbol fs = e as FamilySymbol;
            loadedFamilies += "\t" + fs.Name + "\n";
        }
        TaskDialog.Show("Revit", loadedFamilies);
    }
}
```

3.3.2 Creating elements in families

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.

3.3.2.1 Create a form element

The FamilyItemFactory class provides the ability to create form elements in families, such as extrusions, revolutions, sweeps, and blends. See the section on [3D Sketch](#) 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: Creating a new Extrusion

```
private Extrusion CreateExtrusion(Autodesk.Revit.DB.Document document, Sketch
Plane 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 = Line.CreateBound(p0, p1);
Line line2 = Line.CreateBound(p1, p2);
Line line3 = Line.CreateBound(p2, p3);
Line line4 = Line.CreateBound(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, curveArrArra
y, sketchPlane, 10);

if (null != rectExtrusion)
{
    // move extrusion to proper place
    XYZ transPoint1 = new XYZ(-16, 0, 0);

    ElementTransformUtils.MoveElement(document, rectExtrusion.Id, tra
nsPoint1);
}

else
{
    throw new Exception("Create new Extrusion failed.");
}
```

```
        }

    }

    else

    {

        throw new Exception("Please open a Family document before invoking th
is command.");
    }

    return rectExtrusion;
}
```

The following sample shows how to create a new Sweep from a solid ovoid profile in a Family Document.

Code Region: Creating a new Sweep

```
private Sweep CreateSweep(Autodesk.Revit.DB.Document document, SketchPlane sk
etchPlane)

{
    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(Arc.Create(pnt2, 1.0d, 0.0d, 180.0d, XYZ.BasisX, XYZ.BasisY));
arr.Append(Arc.Create(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 = Line.CreateBound(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);
    ElementTransformUtils.MoveElement(document, sweep.Id, transPoint1);
```

```

    }

    else

    {

        throw new Exception("Failed to create a new Sweep.");

    }

}

else

{

    throw new Exception("Please open a Family document before invoking th
is command.");

}

return sweep;

}

```


Figure 50: Ovoid sweep created by previous example

The FreeFormElement class allows for the creation of non-parametric geometry created from an input solid outline. A FreeFormElement can participate in joins and void cuts with other combinable elements. Planar faces of the element can be offset interactively and programmatically in the face normal direction.

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: 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
    ElementId materialId = Material.Create(document, "Wood Material");
    subCat.Material = document.GetElement(materialId) as Material;

    // assign the subcategory to the element
    extrusion.Subcategory = subCat;
}

```

3.3.2.2 Create an annotation

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 [Annotation Elements](#) 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: Creating a Dimension

```

public Dimension CreateArcDimension(Document document, SketchPlane sketchPlane)
{

```

```

    Autodesk.Revit.Creation.Application appCreate = document.Application.Create
te;

    Line gLine1 = Line.CreateBound(new XYZ(0, 2, 0), new XYZ(2, 2, 0));
    Line gLine2 = Line.CreateBound(new XYZ(0, 2, 0), new XYZ(2, 4, 0));
    Arc arctoDim = Arc.Create(new XYZ(1, 2, 0), new XYZ(-1, 2, 0), new XYZ(0,
3, 0));
    Arc arcofDim = Arc.Create(new XYZ(0, 3, 0), new XYZ(1, 2, 0), new XYZ(0.
8, 2.8, 0));

    Autodesk.Revit.Creation.FamilyItemFactory creationFamily = document.Famil
yCreate;

    ModelCurve modelCurve1 = creationFamily.NewModelCurve(gLine1, sketchPlan
e);
    ModelCurve modelCurve2 = creationFamily.NewModelCurve(gLine2, sketchPlan
e);
    ModelCurve modelCurve3 = creationFamily.NewModelCurve(arctoDim, sketchPla
ne);

    //get their reference
    Reference ref1 = modelCurve1.GeometryCurve.Reference;
    Reference ref2 = modelCurve2.GeometryCurve.Reference;
    Reference arcRef = modelCurve3.GeometryCurve.Reference;

    Dimension newArcDim = creationFamily.NewArcLengthDimension(document.Active
eView, arcofDim, arcRef, ref1, ref2);

    if (newArcDim == null)
    {
        throw new Exception("Failed to create new arc length dimension.");
    }

    return newArcDim;
}

```



Figure 51: 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: 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 = Line.CreateBound(pt1, pt2);
    Plane plane = Plane.CreateByNormalAndOrigin(pt1.CrossProduct(pt2), pt2);
    SketchPlane skplane = SketchPlane.Create(document, plane);
    ModelCurve modelcurve1 = document.FamilyCreate.NewModelCurve(line, skplane);

    pt1 = new XYZ(10, 5, 0);
    pt2 = new XYZ(10, 10, 0);
    line = Line.CreateBound(pt1, pt2);
    plane = Plane.CreateByNormalAndOrigin(pt1.CrossProduct(pt2), pt2);
    skplane = SketchPlane.Create(document, 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 = Line.CreateBound(pt1, pt2);

Dimension dim = document.FamilyCreate.NewLinearDimension(document.ActiveView,
    line, ra);

// create a label for the dimension called "width"

FamilyParameter param = document.FamilyManager.AddParameter("width",
    GroupTypeId.Constraints,
    SpecTypeId.Length, false);

dim.FamilyLabel = param;

return dim;
}

```


Figure 52: Labeled linear dimension

3.3.3 Visibility of family elements

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 = Line.CreateBound(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, 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);

}

}
```

3.3.4 Managing family types and parameters

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.

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)

    {
        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;

}

TaskDialog.Show("Revit",types);

}

}
```

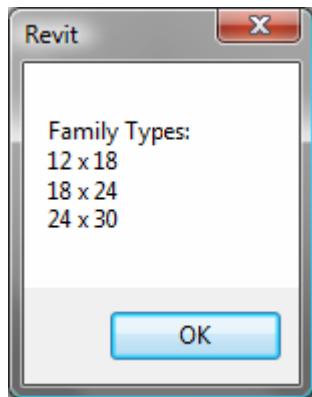


Figure 53: Family types in Concrete-Rectangular-Column family

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 == document) || (null == familyInstance.Symbol))
    {
        return;    // invalid arguments
    }

    // Get family associated with this
    Family family = familyInstance.Symbol.Family;
    if (null == family)
    {
        return;    // could not get the family
    }

    // Get Family document for family
    Document familyDoc = document.EditFamily(family);
    if (null == familyDoc)
```

```
{  
    return; // could not open a family for edit  
}  
  
FamilyManager familyManager = familyDoc.FamilyManager;  
  
if (null == familyManager)  
{  
    return; // cuould not get a family manager  
}  
  
// Start transaction for the family document  
using (Transaction newFamilyTypeTransaction = new Transaction(familyDoc,  
"Add Type to Family"))  
{  
    int changesMade = 0;  
    newFamilyTypeTransaction.Start();  
  
    // add a new type and edit its parameters  
    FamilyType newFamilyType = familyManager.NewType("2X2");  
  
    if (newFamilyType != null)  
    {  
        // 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);  
        }  
    }  
}
```

```
    changesMade += 1;

}

familyParam = familyManager.get_Parameter("h");

if (null != familyParam)
{
    familyManager.Set(familyParam, 2.0);
    changesMade += 1;
}

}

if (2 == changesMade) // set both parameters?
{
    newFamilyTypeTransaction.Commit();
}
else // could not make the change -> should roll back
{
    newFamilyTypeTransaction.Rollback();
}

// if could not make the change or could not commit it, we return
if (newFamilyTypeTransaction.GetStatus() != TransactionStatus.Committ
ed)
{
    return;
}

}
```

```
// now update the Revit project with Family which has a new type

LoadOpts loadOptions = new LoadOpts();

// This overload is necessary for reloading an edited family
// back into the source document from which it was extracted
family = familyDoc.LoadFamily(document, loadOptions);

if (null != family)

{

    // find the new type and assign it to FamilyInstance

    ISet<ElementId> familySymbolIds = family.GetFamilySymbolIds();

    foreach (ElementId id in familySymbolIds)

    {

        FamilySymbol familySymbol = family.Document.GetElement(id) as FamilySymbol;

        if ((null != familySymbol) && familySymbol.Name == "2X2")

        {

            using (Transaction changeSymbol = new Transaction(document, "Change Symbol Assignment"))

            {

                changeSymbol.Start();

                familyInstance.Symbol = familySymbol;

                changeSymbol.Commit();

            }

            break;

        }

    }

}
```

```

    }

}

class LoadOpts : IFamilyLoadOptions
{
    public bool OnFamilyFound(bool familyInUse, out bool overwriteParameterValues)
    {
        overwriteParameterValues = true;
        return true;
    }

    public bool OnSharedFamilyFound(Family sharedFamily, bool familyInUse, out FamilySource source, out bool overwriteParameterValues)
    {
        source = FamilySource.Family;
        overwriteParameterValues = true;
        return true;
    }
}

```

The FamilyManager class provides access to all the family parameters. This includes family built-in parameters, category built-in parameters and shared parameters associated to the family types. There are two ways to get the family parameters:

- Parameters property - gets all parameters in the family
- GetParameters() - gets all the parameters in the family in order in which they appear in the Revit UI When using the GetParameters() method, the Revit UI order is determined first by group and next by the order of the individual parameters.

Family parameters can be reordered (within their groups) from the API for a given family (with the exception of the Rebar Shape family which does not support reordering parameters). This allows for parameters to be presented to the user in the most logical order. Sorting only affects

visible parameters within the same parameter group. Parameters that belong to different groups will remain separated, and the groups' order will not be affected.

The simplest way to reorder parameters is using the FamilyManager.SortParameters() method, which takes a parameter indicating the desired sort order. The sample below sorts the parameters in ascending order.

Code Region: Sort family parameters

```
private void DisplayParametersInAscendingOrder(Document familyDoc)
{
    FamilyManager familyManager = familyDoc.FamilyManager;
    familyManager.SortParameters(ParametersOrder.Ascending);
}
```

Note: The sort is a one-time operation. When new parameters are added they will not be automatically sorted.

For more control over how parameters are sorted, use the FamilyManager.ReorderParameters() method which takes a list of family parameters in the new order. This list must include all the parameters returned by the GetParameters() method, including any invisible parameters, or an exception will be thrown.

3.4 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.

3.4.1 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

```
public void CreateCurve(Document document)
{
    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);
```

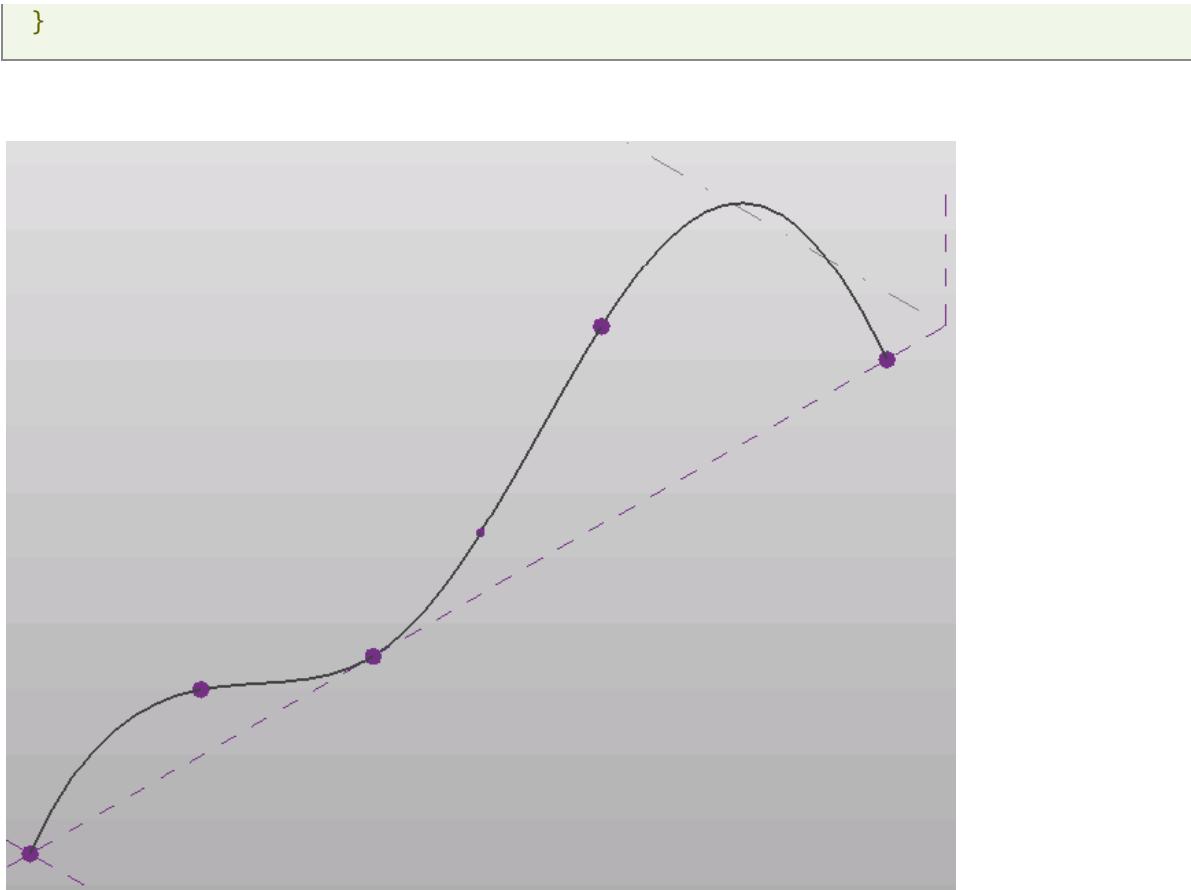


Figure 54: CurveByPoints curve

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 = Line.CreateBound(ptA, ptB);

    ModelCurve modelcurve = MakeModelCurveFromTwoPoints(document, ptA, ptB);

    ref_ar.Append(modelcurve.GeometryCurve.Reference);

    ptA = new XYZ(100, 0, 10);

    ptB = new XYZ(100, 100, 10);

    modelcurve = MakeModelCurveFromTwoPoints(document, ptA, ptB);

    ref_ar.Append(modelcurve.GeometryCurve.Reference);

    ptA = new XYZ(100, 100, 10);

    ptB = new XYZ(0, 0, 10);

    modelcurve = MakeModelCurveFromTwoPoints(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 = MakeModelCurveFromTwoPoints(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 forms from a single profile.

    revolveForms = document.FamilyCreate.NewRevolveForms(true, ref_ar, axis.GeometryCurve.Reference, 0, Math.PI / 4);

    return revolveForms;
}

public ModelCurve MakeModelCurveFromTwoPoints(Document doc, XYZ ptA, XYZ ptB)
{
    Autodesk.Revit.ApplicationServices.Application app = doc.Application;

    // Create plane by the points
    Line line = Line.CreateBound(ptA, ptB);

    XYZ norm = ptA.CrossProduct(ptB);

    if (norm.IsZeroLength()) norm = XYZ.BasisZ;

    Plane plane = Plane.CreateByNormalAndOrigin(norm, ptB);

    SketchPlane skplane = SketchPlane.Create(doc, plane);

    // Create line here
    ModelCurve modelcurve = doc.FamilyCreate.NewModelCurve(line, skplane);

    return modelcurve;
}
```

{}

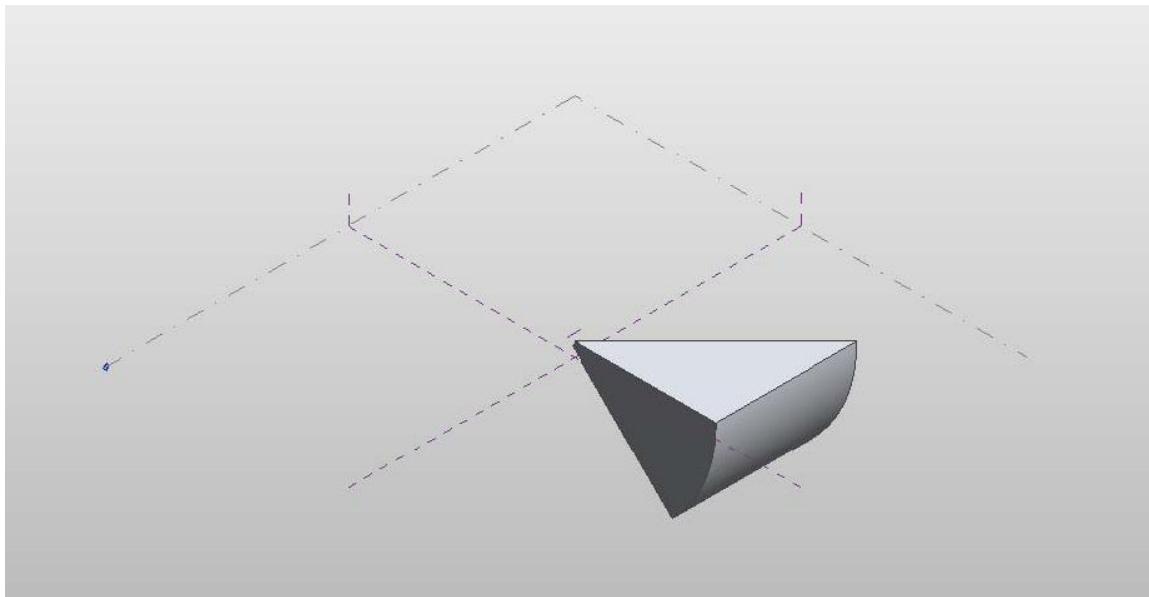


Figure 55: Resulting Revolve Form

3.4.2 Forms

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.DB.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);

    int profileCount = extrusionForm.ProfileCount;

    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 = Line.CreateBound(ptA, ptB);
    XYZ norm = ptA.CrossProduct(ptB);

    if (norm.IsZeroLength()) norm = XYZ.BasisZ;
    Plane plane = Plane.CreateByNormalAndOrigin(norm, ptB);
    SketchPlane skplane = SketchPlane.Create(doc, plane);

    // Create line here
    ModelCurve modelcurve = doc.FamilyCreate.NewModelCurve(line, skplane);
    return modelcurve;
}
```

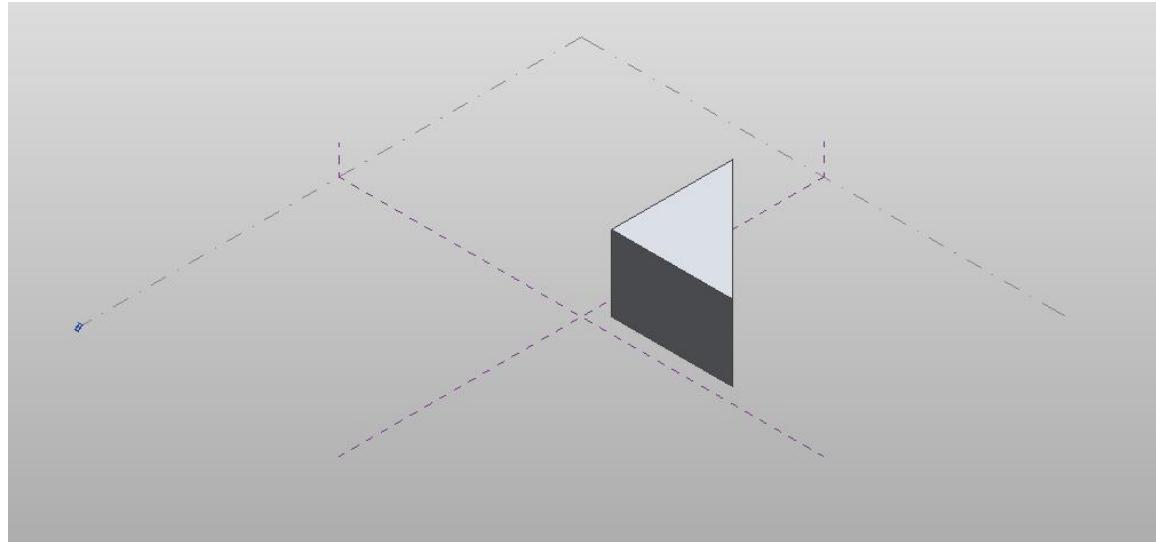


Figure 56: Resulting extrusion form

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(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;

}
```

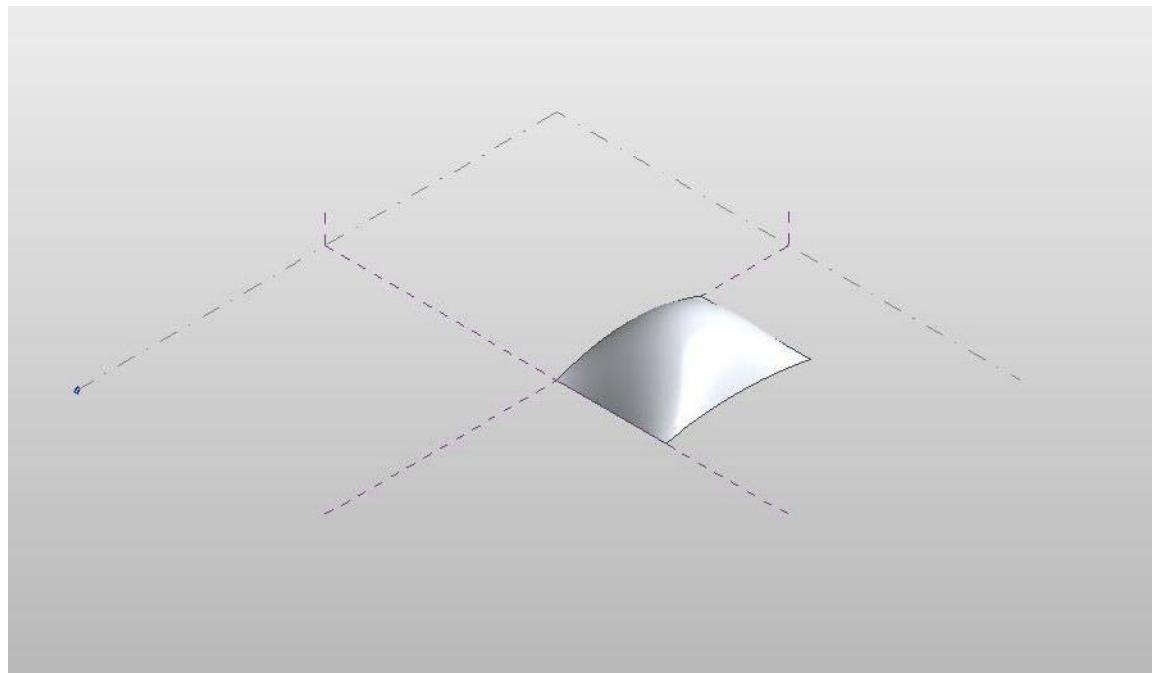


Figure 57: Resulting loft form

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);
        }
    }
}
```

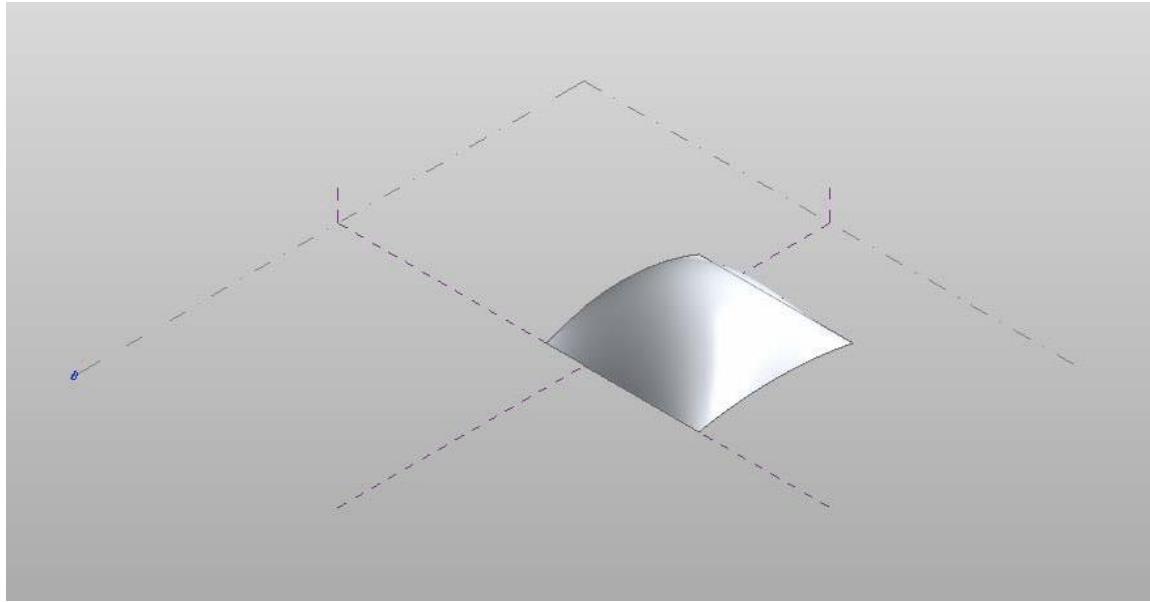


Figure 58: Modified loft form

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)
{
    Document document = form.Document;
    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 = document.GetElement(r2).GetGeometryObjectFromReference(r2) as Point;

        XYZ offset = new XYZ(0, 15, 0);
        form.MoveSubElement(r2, offset);
        break; // just move the first point
    }
}
}
}
```

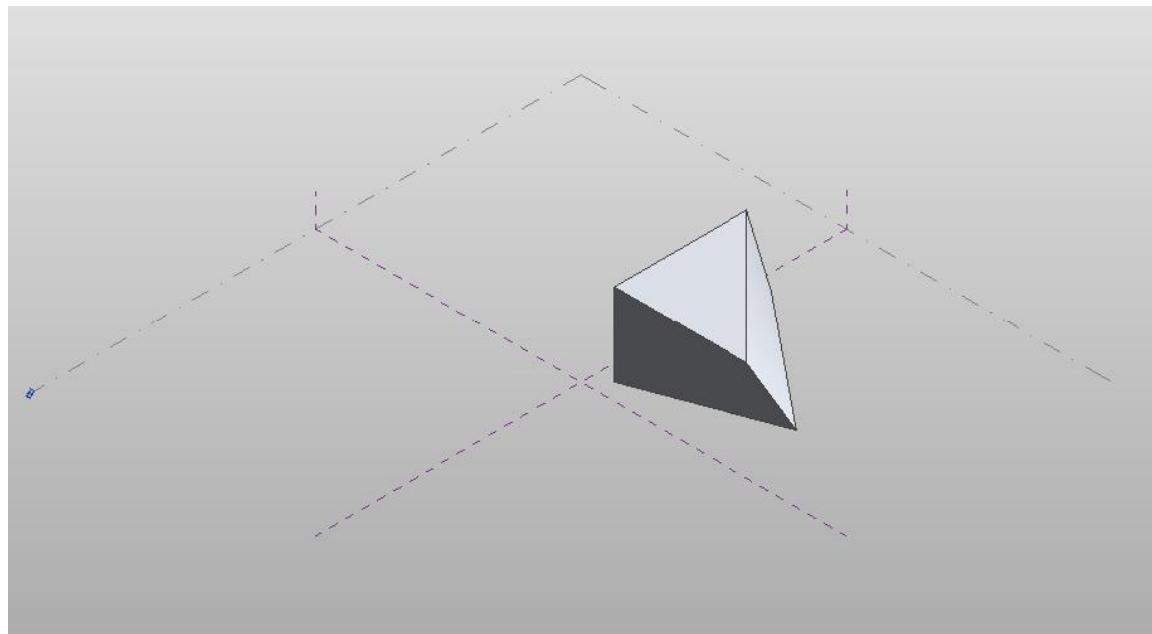


Figure 59: Modified extrusion form

3.4.3 Rationalizing a Surface

Dividing a surface

Faces of forms can be divided with UV grids. You can access the data for a divided surface using the `DividedSurface.GetReferencesWithDividedSurface()` and `DividedSurface.GetDividedSurfaceForReference()` methods (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)
{
    Autodesk.Revit.ApplicationServices.Application application = document.Application;

    Options opt = application.Create.NewGeometryOptions();
    opt.ComputeReferences = true;

    Autodesk.Revit.DB.GeometryElement geomElem = form.get_Geometry(opt);

    foreach (GeometryObject geomObj in geomElem)
    {
        Solid solid = geomObj as Solid;
        foreach (Face face in solid.Faces)
        {
            if (face.Reference != null)
            {
                DividedSurface ds = DividedSurface.Create(document, 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  
    }  
}  
}  
}
```

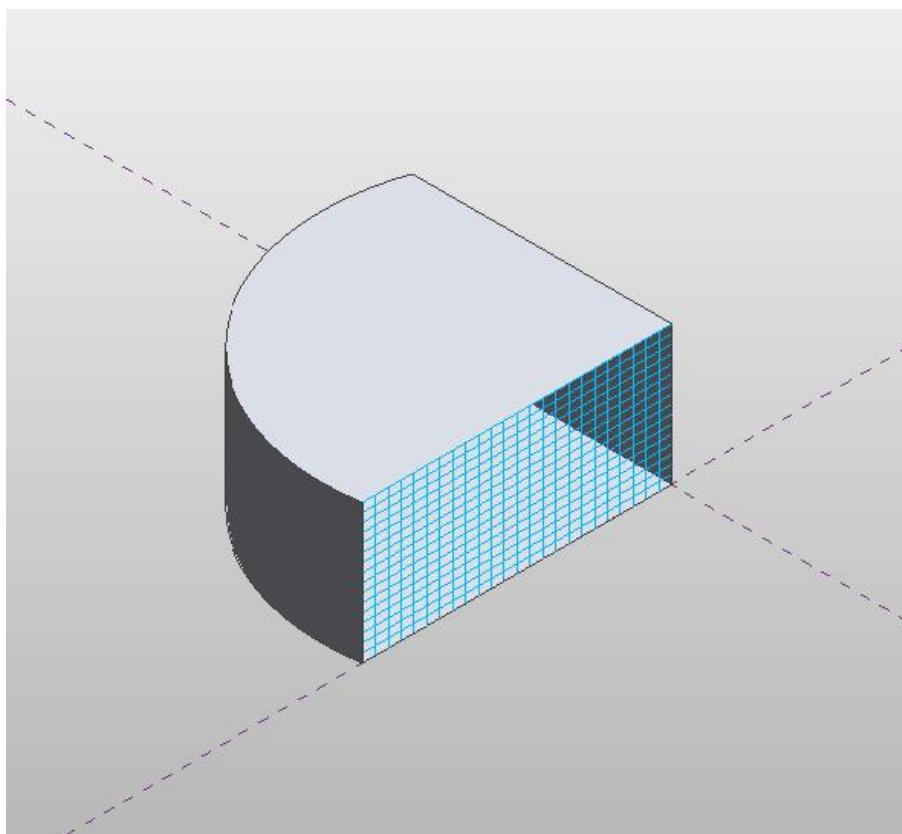


Figure 60: Face of form divided by UV grids

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.

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 tilePatterns = document.Settings.TilePatterns;
    foreach (Reference r in DividedSurface.GetReferencesWithDividedSurfaces(form))
    {
        DividedSurface ds = DividedSurface.GetDividedSurfaceForReference(document, r);
        ds.ChangeTypeId(tilePatterns.GetTilePattern(TilePatternsBuiltIn.OctagonRotate).Id);
    }
}
```

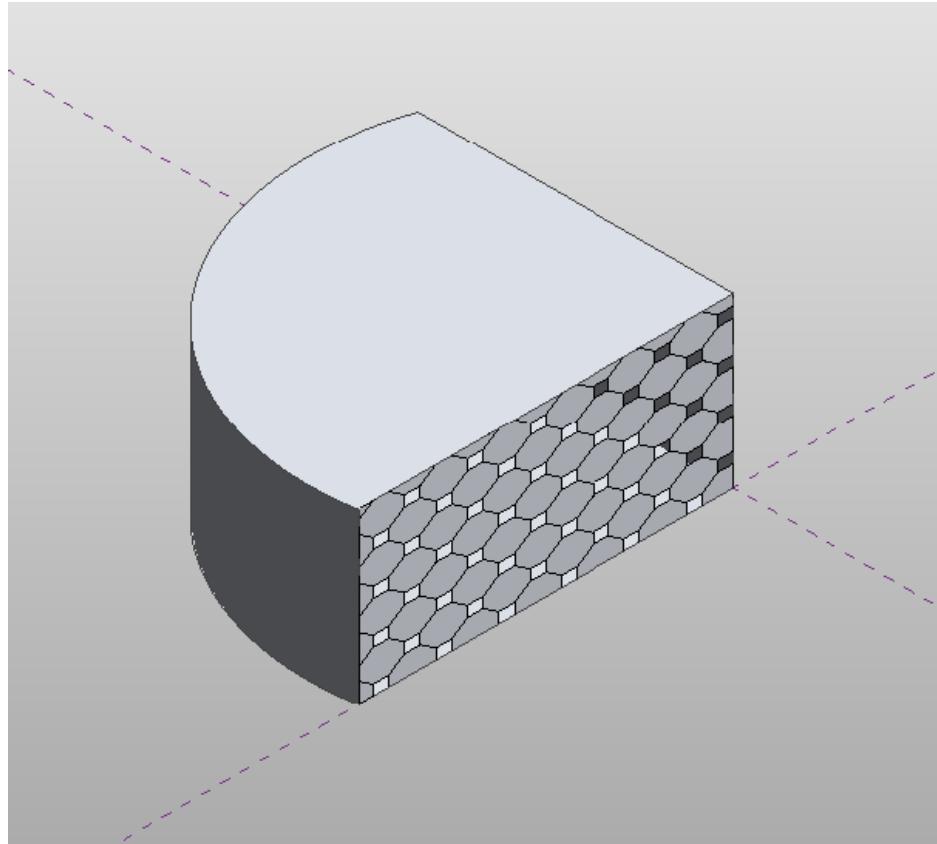


Figure 61: Tile pattern applied to divided surface

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

```
public void EditCurtainPanel(Document document)

{
    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);

PointLocationOnCurve pointLocationOnCurve25 = new PointLocationOnCurve(PointOnCurveMeasurementType.NormalizedCurveParameter, 0.25, PointOnCurveMeasureFrom.Beginning);

PointOnEdge poeA = app.Create.NewPointOnEdge(curve1.GeometryCurve.Reference, pointLocationOnCurve25);

ReferencePoint rpA = document.FamilyCreate.NewReferencePoint(poeA);

PointLocationOnCurve pointLocationOnCurve75 = new PointLocationOnCurve(PointOnCurveMeasurementType.NormalizedCurveParameter, 0.75, PointOnCurveMeasureFrom.Beginning);

PointOnEdge poeB = app.Create.NewPointOnEdge(curve1.GeometryCurve.Reference, pointLocationOnCurve75);

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, pointLocationOnCurve25);

ReferencePoint rpC = document.FamilyCreate.NewReferencePoint(poeC);

PointOnEdge poeD = app.Create.NewPointOnEdge(curve2.GeometryCurve.Reference, pointLocationOnCurve75);

ReferencePoint rpD = document.FamilyCreate.NewReferencePoint(poeD);

}

else

{

    throw new Exception("Please open a curtain family document before calling this command.");

}

}
```

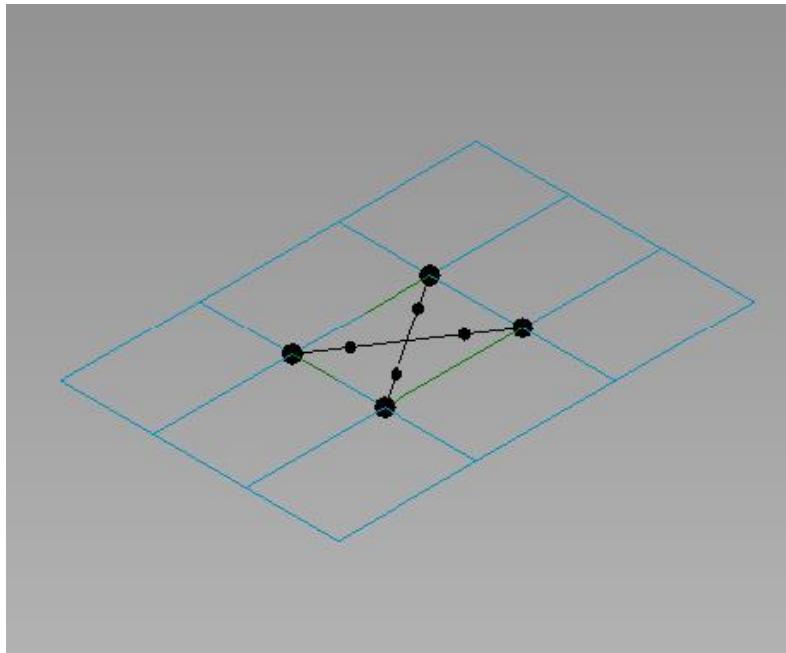


Figure 62: Curtain panel family

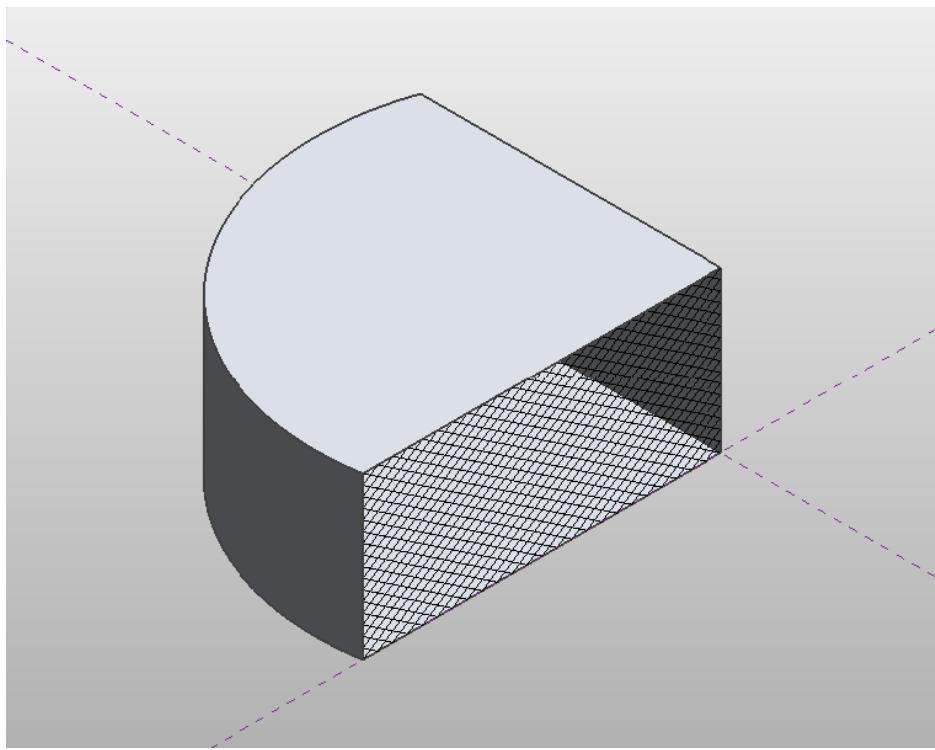


Figure 63: Curtain panel assigned to divided surface

3.4.4 Adaptive Components

Adaptive Components are designed to handle cases where components need to flexibly adapt to many unique contextual conditions. For example, adaptive components could be used in repeating systems generated by arraying multiple components that conform to user-defined constraints.

The following code shows how to create an instance of an adaptive component family into a massing family and set the position of each point mathematically.

Code Region: Creating an Instance of an Adaptive Component Family

```
private void CreateAdaptiveComponentInstance(Document document, FamilySymbol symbol)

{
    // Create a new instance of an adaptive component family

    FamilyInstance instance = AdaptiveComponentInstanceUtils.CreateAdaptiveCo
mponentInstance(document, symbol);

    // Get the placement points of this instance

    IList<ElementId> placePointIds = new List<ElementId>();

    placePointIds = AdaptiveComponentInstanceUtils.GetInstancePlacementPointE
lementRefIds(instance);

    double x = 0;

    // Set the position of each placement point

    foreach (ElementId id in placePointIds)
    {

        ReferencePoint point = document.GetElement(id) as ReferencePoint;

        point.Position = new Autodesk.Revit.DB.XYZ(10*x, 10*Math.Cos(x), 0);

        x += Math.PI/6;
    }
}
```

{}

To batch create adaptive components, you can use the overload of the FamilyInstanceCreationData constructor that takes two parameters - a FamilySymbol and a list of XYZ adaptive points where the adaptive instance is to be initialized . In conjunction with the Autodesk.Revit.Creation.ItemFactoryBase.NewFamilyInstances2() method which takes a list of FamilyInstanceCreationData objects, multiple adaptive components can be added at once. This may be more efficient than placing individual adaptive components one-by-one.

3.4.5 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.

To create a manifest file

1. 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">

  <Name>HelloWorld</Name>

  <FullClassName>HelloWorld.HelloWorld</FullClassName>

  <Text>HelloWorld</Text>

  <Description>Show Hello World.</Description>

  <VisibilityMode>AlwaysVisible</VisibilityMode>

  <Assembly>C:\Samples\HelloWorld\HelloWorld\bin\Debug\HelloWorld.dll</Assembly>

  <AddInId>239BD853-36E4-461f-9171-C5ACEDA4E723</AddInId>

  <VendorId>ADSK</VendorId>

  <VendorDescription>Autodesk, Inc, www.autodesk.com</VendorDescription>

</AddIn>
```

```
</RevitAddIns>
```

Note: The FullClassName includes the Root namespace found on the Application tab of the properties for the project.

3. Save the file as HelloWorld.addin and put it in the following location:
 - C:\ProgramData\Autodesk\Revit\Addins{{RelYear}}\

Refer to [Add-In Integration](#) for more details using manifest files.

3.5 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 [Elements Essentials](#).

If you need more information, refer to the related chapter:

- For LoadBase, LoadCase, LoadCombination, LoadNature and LoadUsage, refer to [Structural Engineering](#)
- For ModelCurve, refer to [Sketching](#)
- For Material and FillPattern, refer to [Material](#)
- For EnergyDataSettings, refer to [Energy Data](#)

3.5.1 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 DatumPlane class, which 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 an ElementClassFilter with the Level class.

Elevation

The Level class has the following properties:

- The Elevation property 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.

- The Elevation Base value 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.

The following code sample illustrates how to retrieve all levels in a project using a Level class filter.

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());  
}
```

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.DB.Document document)  
{  
    // The elevation to apply to the new level
```

```

    double elevation = 20.0;

    // Begin to create a level

    Level level = Level.Create(document, 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.

Finding levels based on elevation

You can find levels based on their elevation with a `FilteredElementCollector`. You can also use one of these static methods which return the id of the Level which is closest to the specified elevation. The level can be at, above, or below the target elevation. If there is more than one Level at the same distance from the elevation, the Level with the lowest id will be returned

- `Level.GetNearestLevelId(Document document, double elevation)`
- `Level.GetNearestLevelId(Document document, double elevation, out double offset)`

3.5.2 Grids

The Grid class represents a single grid line within Autodesk Revit.

Grids are represented by the Grid class which is derived from the DatumPlane 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.

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 [Geometry](#).

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: (" + XYZString(arc.Center);
    }
    else
```

```

    {

        // if the curve is a line, give length information

        Autodesk.Revit.DB.Line line = curve as Autodesk.Revit.DB.Line;

        message += "\nLine's Length: " + line.Length;

    }

    // Get curve start point

    message += "\nStart point: " + XYZString(curve.GetEndPoint(0));

    // Get curve end point

    message += "; End point: " + XYZString(curve.GetEndPoint(0));

    TaskDialog.Show("Revit", message);

}

// output the point's three coordinates

private string XYZString(XYZ point)

{
    return "(" + point.X + ", " + point.Y + ", " + point.Z + ")";
}

```

Creating a Grid

Two overloaded Create() methods are available in the Grid class 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: Grid.Create()

```
public Grid Create( Document document, Arc arc );
```

```
public Grid Create( Document document, 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.DB.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.CreateBound(start, end);

    // Create a grid using the geometry line
    Grid lineGrid = Grid.Create(document, 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 = Arc.Create(end0, end1, pointOnCurve);

// Create a grid using the geometry arc

Grid arcGrid = Grid.Create(document, 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.

3.5.3 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 Created Phase ID and a Demolished Phase ID property. The `Element.AllowPhases()` method indicates whether its phase ID properties can be modified.

The Created Phase ID property has the following characteristics:

- It identifies the phase in which the component was added.
- The default value is the same ID as the current view Phase value.
- Change the Created Phase ID parameter by selecting a new value corresponding to the drop-down list.

The Demolished Phase ID 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 ID value in the view where you demolished the element.
- You can demolish a component by setting the Demolished Phase ID 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 Created Phase ID parameter value for these components is changed to the next item in the drop-down list in the Properties dialog box.

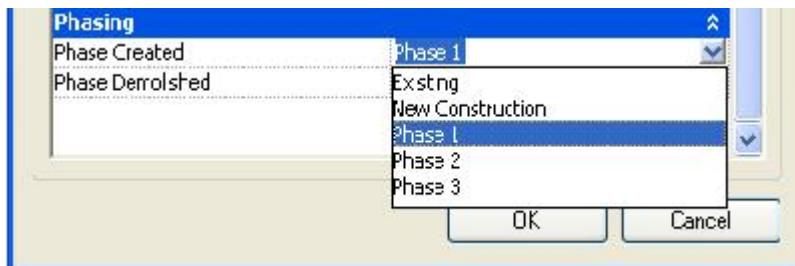


Figure 65: Phase-created component parameter value

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);
}

```

Validating phase data of an element

`Element.IsDemolishedPhaseOrderValid()` and `Element.IsCreatedPhaseOrderValid()` validate the order of phases on a given element, ensuring that an object is not assigned a phase where it is demolished before it was created.

3.5.4 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);

    ICollection<ElementId> selectedIds = uidoc.Selection.GetElementIds();

    foreach (ElementId id in selectedIds)
    {
        Element element = document.GetElement(id);
        //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.

3.6 Annotation Elements

This section covers Revit Annotation Elements, such as dimensions, text notes, keynotes, tags, and symbols.

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 [Elements Essentials](#).

3.6.1 Dimensions and Constraints

Permanent dimensions and dimension related 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.GetEndPoint(0).X + ", "
        + curve.GetEndPoint(0).Y + ", " + curve.GetEndPoint(0).Z + ")"

    ;

    // Get curve end point

    message += "; Curve end point:(" + curve.GetEndPoint(1).X + ", "
        + curve.GetEndPoint(1).Y + ", " + curve.GetEndPoint(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.Name;

// Get Dimension reference count

message += "\nDimension references count is " + dimension.References.Size;

if ((int)BuiltInCategory.OST_Dimensions == dimension.Category.Id.Value)

{

    message += "\nDimension is a permanent dimension.";

}
```

```

        else if ((int)BuiltInCategory.OST_Constraints == dimension.Category.Id.Value)
    {
        message += "\nDimension is a constraint element.";
    }

    TaskDialog.Show("Revit", message);
}

```

Dimensions

There are five kinds of permanent dimensions:

- Linear dimension
- Radial dimension
- Diameter Dimension
- Angular dimension
- Arc length dimension

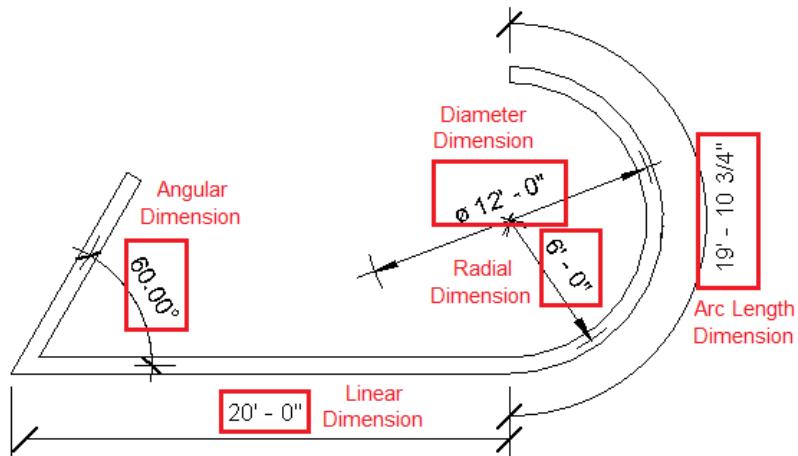


Figure 66: Permanent dimensions

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 and diameter dimensions, 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.
- A diameter dimension returns null.

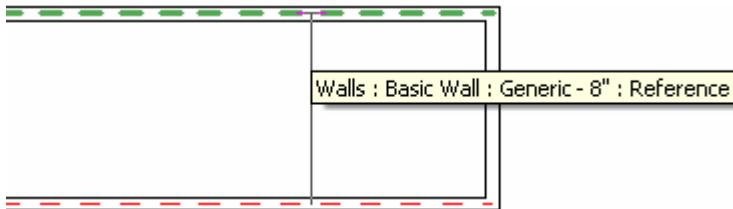


Figure 67: Dimension references

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 [Geometry Helper Classes](#) in the [Geometry](#) section.

- Radial and diameter dimensions - 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.

Dimensions can be created with the methods

- `RadialDimension.Create()`
- `ArcLengthDimension.Create()`
- `LinearDimension.Create()`

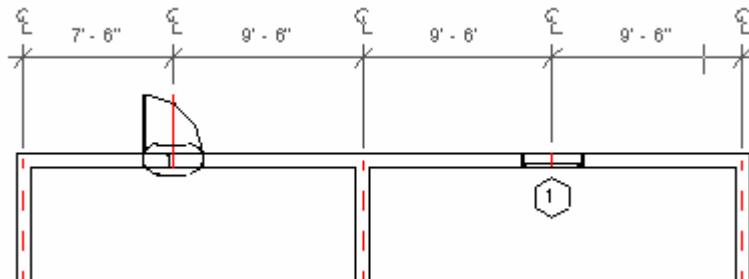


Figure 68: Linear dimension references

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.

Constraint Elements

Dimension objects with Category Constraints (`BuiltInCategory.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.

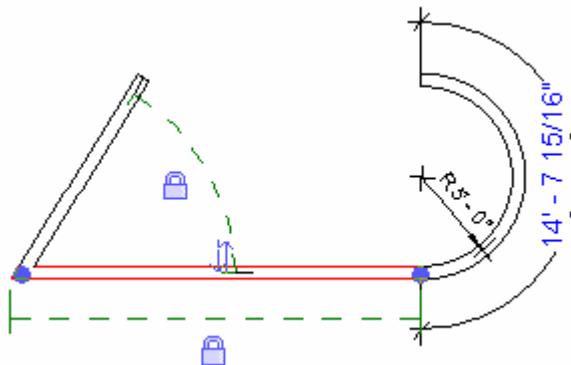


Figure 69: Linear and Radial dimension constraints

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.

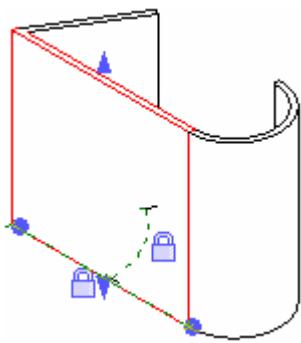
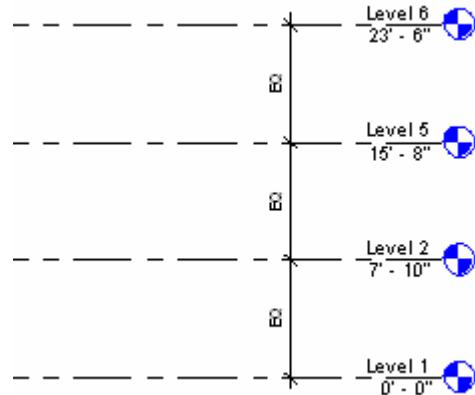


Figure 70: Linear and Radial dimension constraints in 3D view

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.

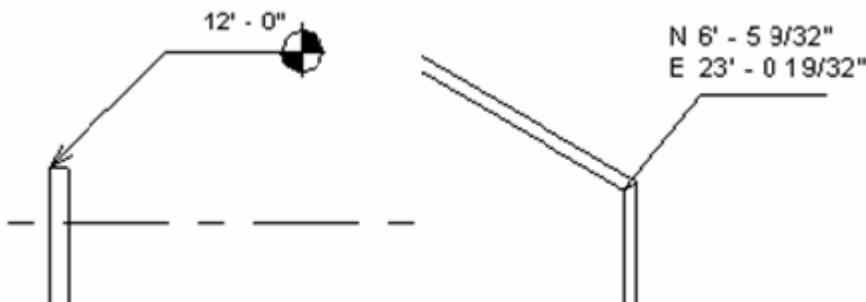
**Figure 71: Equality constraints**

Note: Not all constraint elements are represented by the Dimension class but all belong to a Constraints (OST_Constraints) category such as alignment constraint. ### 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:

Table 35: Spot dimension Type and Category

Type	Category
Spot Coordinates	OST_SpotCoordinates
Spot Elevations	OST_SpotElevations

**Figure 72: 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.

Information about the leader of a spot dimension is accessible with:

- SpotDimension.LeaderElbowPosition
- SpotDimension.LeaderHasElbow

Comparison

The following table compares different kinds of dimensions and constraints in the API:

Table 36: Dimension Category Comparison

<i>Dimension or Constraint</i>	<i>Dimension or Constraint</i>	<i>API Class</i>	<i>BuiltInCategory</i>	<i>Curve</i>	<i>Geometry Helper Classes</i>	<i>View</i>	<i>Location</i>
Permanent Dimension	linear dimension	LinearDimension	OST_Dimensions	A Line	=2	Specific view	null
Permanent Dimension	radial dimension	RadialDimension	OST_Dimensions	Nul l	1	Specific view	null
Permanent Dimension	diameter dimension	Dimension	OST_Dimensions	Nul l	1	Specific view	null
Permanent Dimension	angular dimension	Dimension	OST_Dimensions	An Arc	2	Specific view	null

Permanent Dimension	arc length dimension	ArcLengthDimension	OST_Dimensions	An Arc	2	Specific view	null
Dimension Constraint	linear dimension constraint	Dimension	OST_Constraints	An Arc	2		null
Dimension Constraint	angular dimension	Dimension	OST_Constraints	An Arc	2		null
Equality Constraint	Equality Constraint	Dimension	OST_Constraints	A Line	=3		null

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.

Manipulating Dimension Text and Leader

Access to several attributes of the dimension are available with:

- DimensionType.Prefix
- DimensionType.Suffix
- Dimension.TextPosition
- Dimension.LeaderEndPosition
- Dimension.HasLeader

Dimension and DimensionSegment classes provide similar properties and methods for querying and adjusting the position of the text relative to the dimension curve.

Dimension.Origin returns the XYZ value of the midpoint of the dimension curve, with DimensionSegment.Origin will return the midpoint of the line that makes up the segment.

Determine if text position of a Dimension or DimensionSegment is adjustable by calling the method IsTextPositionAdjustable() which will indicate whether the text and leader positions may be set.

Query or modify the position of text or the leader (of a dimension or dimension segment) by using the properties TextPosition and LeaderEndPosition.

Reset the text to its default position on a dimension by calling the method `ResetTextPosition()`.

Note: `TextPosition` and `LeaderEndPosition` are not necessarily applicable to all dimensions (e.g., spot dimensions, multi-segment dimensions using the equality constraint, when dimension style is ordinate). If these values are not applicable they will return `Null` and not allow setting a value.

Code Region: Reposition dimension text

```
// Moves all of the text in this dimension one unit in the Y direction

public bool DimensionTextReposition(Dimension dimToModify)

{
    bool modified = false;

    if (dimToModify == null)
        return false;

    // Check to see if we have a non-multisegment dimension and if text position is adjustable

    if (dimToModify.NumberOfSegments == 0 && dimToModify.IsTextPositionAdjustable())
    {

        // Get the current text XYZ position

        XYZ currentTextPosition = dimToModify.TextPosition;

        // Calculate a new XYZ position by transforming the current text position

        XYZ newTextPosition = Transform.CreateTranslation(new XYZ(0.0, 1.0, 0.0)).OfPoint(currentTextPosition);

        // Set the new text position

        dimToModify.TextPosition = newTextPosition;

        modified = true;
    }
}
```

```

else if (dimToModify.NumberOfSegments > 0)

{
    foreach (DimensionSegment currentSegment in dimToModify.Segments)
    {
        if (currentSegment != null && currentSegment.IsTextPositionAdjustable())
        {
            modified = true;

            // Get the current text XYZ position
            XYZ currentTextPosition = currentSegment.TextPosition;

            // Calculate a new XYZ position by transforming the current text position
            XYZ newTextPosition = Transform.CreateTranslation(new XYZ(0, 1, 0)).OfPoint(currentTextPosition);

            // Set the new text position for the segment's text
            currentSegment.TextPosition = newTextPosition;
        }
    }

    return modified;
}

```

3.6.2 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)| |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 [ModelCurve](#) in the [Sketching](#) section.

3.6.3 Tags

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 [Rooms](#).

Note: The IndependentTag class represents the tag element in Revit and other specific tags such as keynote, beam system tag, electronic circuit symbol, 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 IndependentTag.Create. They can be distinguished by the following categories:

Table 37: Tag Name and Category

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

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.

Note: Note that `IndependentTag.Create` only works in the 2D view or in a locked 3D 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.

The `LeadersPresentationMode` property specifies how leaders should be displayed on a tag. It has the following values:

- `ShowAll`
- `HideAll`
- `ShowOnlyOne`
- `ShowSpecificLeaders`

`IsLeaderVisible()` returns whether the leader that points to the specified reference is visible or not, and `SetIsLeaderVisible()` sets the visibility of the leader that points to the specified reference.

Code Region 16-5: Creating an `IndependentTag`

```
private IndependentTag CreateIndependentTag(Autodesk.Revit.DB.Document document, Reference reference)

{
    // make sure active view is not a 3D view
    Autodesk.Revit.DB.View view = document.ActiveView;
    if (view is View3D)
        return null;

    // define tag mode and tag orientation for new tag
    TagMode tagMode = TagMode.TM_ADDBY_CATEGORY;
    TagOrientation tagorn = TagOrientation.Horizontal;

    // Add the tag to the middle of the wall
```

```
Wall wall = document.GetElement(reference) as Wall;  
if (wall == null)  
    return null;  
  
LocationCurve wallLoc = wall.Location as LocationCurve;  
XYZ wallStart = wallLoc.Curve.GetEndPoint(0);  
XYZ wallEnd = wallLoc.Curve.GetEndPoint(1);  
XYZ wallMid = wallLoc.Curve.Evaluate(0.5, true);  
  
IndependentTag newTag = IndependentTag.Create(document, view.Id, reference, 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.LookupParameter("Type Mark");  
bool result = foundParameter.Set("Hello");  
  
// set leader mode free
```

```

// otherwise leader end point move with elbow point

newTag.LeaderEndCondition = LeaderEndCondition.Free;

XYZ elbowPnt = wallMid + new XYZ(5.0, 5.0, 0.0);

newTag.SetLeaderElbow(reference, elbowPnt);

XYZ headerPnt = wallMid + new XYZ(10.0, 10.0, 0.0);

newTag.TagHeadPosition = headerPnt;

return newTag;
}

```

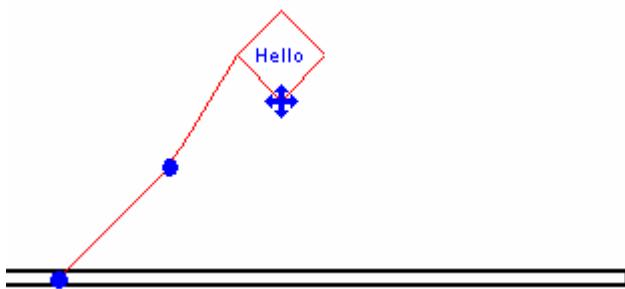


Figure 74: Create IndependentTag using sample code

3.6.4 Text

Text and associated leaders can be accessed from the `TextNote` class. Note

A `TextNote` can have plain text or formatted text. The overloaded `TextNote.Create()` method provides options for creating unwrapped and line-wrapping text note elements. The width for the area of the text content can be specified on creation, but is restricted by a minimum and maximum width that is based on properties of the text and its type. The overloaded methods `GetMinimumAllowedWidth()` and `GetMaximumAllowedWidth()`, inherited from `TextElement`, return the constraints for either a specific `TextNote` or for a given document and text type id.

The following example creates a new `TextNote` at a user specified point and with a given width and `TextNoteOptions`.

Code Region: Create a TextNote

```
public TextNote AddNewTextNote(UIDocument uiDoc)
{
    Document doc = uiDoc.Document;

    XYZ textLoc = uiDoc.Selection.PickPoint("Pick a point for sample text.");

    ElementId defaultTextTypeId = doc.GetDefaultElementType(ElementTypeGroup.TextNoteType);

    double noteWidth = .2;

    // make sure note width works for the text type
    double minWidth = TextNote.GetMinimumAllowedWidth(doc, defaultTextTypeId);
    double maxWidth = TextNote.GetMaximumAllowedWidth(doc, defaultTextTypeId);

    if (noteWidth < minWidth)
    {
        noteWidth = minWidth;
    }
    else if (noteWidth > maxWidth)
    {
        noteWidth = maxWidth;
    }

    TextNoteOptions opts = new TextNoteOptions(defaultTextTypeId);
    opts.HorizontalAlignment = HorizontalTextAlignment.Left;
    opts.Rotation = Math.PI / 4;
```

```
    TextNote textNote = TextNote.Create(doc, doc.ActiveView.Id, textLoc, noteWidth, "New sample text", opts);

    return textNote;
}
```

Whether a TextNote has plain or formatted text, the unformatted text can always be retrieved from the TextNote.Text property.

3.7 FormattedText

When first created, the TextNote will have plain text. Use the TextNote.GetFormattedText() method to get a FormattedText object for the TextNote. The FormattedText class can be used to apply various formatting to the text such as bold, underline, superscript or all caps. The TextNote is not updated until SetFormattedText() is called with the modified FormattedText.

The text in the FormattedText can be formatted in whole or in part using a TextRange. A TextRange specifies a start index and length based on the text in the FormattedText object. When the overload of a formatting method (such as SetItalicStatus() or SetAllCapsStatus()) uses a TextRange, only the characters within the range will be modified. A TextRange can be defined explicitly using its constructor, or can be retrieved using the FormattedText.Find() method to get the range for a given search string. The Find() method specifies a start index for the search, as well as whether to match the case of the search string or whether to do a whole word search. If the text in the search string is not found or if the given start index is beyond the end of the length of the entire text, an empty TextRange will be returned. Before using the returned range to set formatting on the text, ensure that it is not empty to avoid an exception.

The following example demonstrates how to format text from a TextNote and set it back to the TextNote. It uses the Find() method to bold and underline specific words in the text.

Code Region: Format text in a TextNote

```
public void FormatText(TextNote textNote)
{
    // TextNote created with "New sample text"

    FormattedText formatText = textNote.GetFormattedText();

    // italicize "New"
}
```

```
    TextRange range = new TextRange(0, 3);

    formatText.SetItalicStatus(range, true);

    // make "sample" bold

    range = formatText.Find("sample", 0, false, true);

    if (range.Length > 0)

        formatText.SetBoldStatus(range, true);

    // make "text" underlined

    range = formatText.Find("text", 0, false, true);

    if (range.Length > 0)

        formatText.SetUnderlineStatus(range, true);

    // make all text uppercase

    formatText.SetAllCapsStatus(true);

    textNote.SetFormattedText(formatText);

}
```

New text can be added to existing text in a FormattedText object. The SetPlainText() method will either replace some existing text if the overload that has a TextRange parameter is used, or will replace the entire text if not. To insert text without replacing existing text, use a TextRange with a Length of 0. The new text will be inserted at the index specified by the TextRange.Start property. Note that when inserting text, it may pick up the formatting of the adjacent text, similar to how pasting unformatted text into a Word document will result in text with the current formatting for the insertion point. If formatting has been applied to the entire FormattedText as in the case of the SetAllCapsStatus(true) call in the example above, that formatting will be applied to any new text that is inserted.

In the following example, new text is appended to the end of existing text by first finding the end of the current text and setting that as the Start of the range to be added. It also demonstrates how to create a list (which can be bulleted, numbered or lettered). Note that it also calls GetAllCapsStatus() for the range of the new text and turns off caps if the status is not FormatStatus.None (other options are All and Mixed).

Code Region: Inserting new text

```
public void AppendText(TextNote textNote)
{
    FormattedText formatText = textNote.GetFormattedText();

    TextRange range = formatText.AsTextRange();

    range.Start = range.End - 1;
    // set Length to 0 to insert
    range.Length = 0;

    string someNewText = "\rThis is a new paragraph\nThis is a new line without a paragraph break\r";
    formatText.SetPlainText(range, someNewText);

    // get range for entire text
    range = formatText.AsTextRange();
    range.Start = range.End - 1;
    range.Length = 0;

    string someListText = "\rBulleted List item 1\rItem 2\nSecond line for Item 2\rThird bullet point";
    formatText.SetPlainText(range, someListText);

    range.Start++;
    range.Length = someListText.Length;
    formatText.SetListType(range, ListType.Bullet);

    if (formatText.GetAllCapsStatus(range) != FormatStatus.None)
```

```

    {
        formatText.SetAllCapsStatus(range, false);

    }

    textNote.SetFormattedText(formatText);

}

```

The code above shows how to use `\r` to create a line break, and `\v` for a vertical tab that does not break the paragraph. In the text for the bulleted list, a "`\v`" is used to create a two line bullet point. A new bullet is only inserted when using "`\r`".

3.8 Text Editor

The `TextEditorOptions` class can be used to control the appearance and functionality of the text editor in Revit. These settings are saved in the `Revit.ini` file and are not tied to the document.

The `Revit.ini` file is in the folder returned by the property `Autodesk.Revit.ApplicationServices.Application.CurrentUsersDataFolderPath` (such as `%appdata%\Autodesk\Revit\Autodesk Revit 2019`)

Code Region: Setting text editor options

```

public void SetEditorOptions()
{
    TextEditorOptions editorOptions = TextEditorOptions.GetTextEditorOptions();
    editorOptions.ShowBorder = false;
    editorOptions.ShowOpaqueBackground = true;
}

```

3.9 Leaders

Revit supports two kinds of Leaders: straight leaders and arc leaders. Leaders can be added to a `TextNote` using the `AddLeader()` method, specifying the leader type using the `TextNoteLeaderType` enumerated type:

Table 39: Leader Types

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

Note: Straight leaders and arc leaders cannot be added to a Text type at the same time.

The TextNote.LeaderCount property returns the number of leaders and the GetLeaders() method returns all leaders currently attached to the text component. LeaderLeftAttachment and LeaderRightAttachment indicate the attachment position of the leaders on the corresponding side of the TextNote. Options for the LeaderAttachment are TopLine, MidPoint, and BottomLine. Use the RemoveLeaders() method to remove all leaders from the TextNote.

3.9.1 Annotation Symbol

An annotation symbol is a symbol applied to a family to uniquely identify that family in a project.

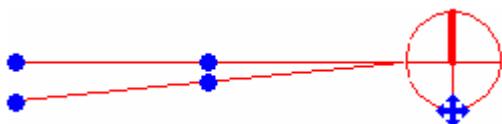


Figure 76: Annotation Symbol with two leaders

Create and Delete

Annotation symbols can be created using the following overload of the Creation.Document.NewFamilyInstance() method:

Code Region 16-6: Create a new Annotation Symbol

```
public FamilyInstance NewFamilyInstance Method (XYZ origin, FamilySymbol symbol, View specView)
```

The annotation symbol can be deleted using the Document.Delete() method.

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)
{
    // check if there are any leaders currently attached, and remove them
    IList<Leader> leaders = symbol.GetLeaders();

    if (leaders != null && leaders.Count > 0)
    {
        for (int i = leaders.Count; i > 0; i--)
        {
            symbol.removeLeader();
        }
    }

    // add one new leader instead
    symbol.addLeader();
}
```

3.9.2 Color Fill

Color Fill Legend

`Autodesk.Revit.DB.ColorFillLegend` supports creating, reading and modifying properties of color fill legend annotation elements in a particular view. The static method `ColorFillLegend.Create(document, viewId, categoryId, origin)` lets you to create a new color fill legend in a view.

Color Fill Scheme

The color fill scheme definition includes:

- The category (such as Rooms).
- The parameter whose value will be used to color the elements (such as Name, Floor Finish, or Department).
- If the coloring will be done by range of values (such as at least 40SF' and less than 80SF') or by specific values (such as equal to 40SF').
- The list of `ColorFillSchemeEntry` items.

Key members of `Autodesk.Revit.DB.ColorFillScheme` relating to these attributes are:

- `CategoryId`
- `ParameterDefinition`
- `IsByRange`
- `AddEntry()/DeleteEntry()`
- `GetEntries()/SetEntries()`

Color Fills Schemes used by a view

A view can define one `ColorFillScheme` for spatial elements (rooms, zones, spaces, and areas), one for pipes, and one for ducts. You can get and set these schemes with

- `View.GetColorFillSchemeld(categoryId)`
- `View.SetColorFillSchemeld(categoryId, schemeld)`

Color Fill Scheme Entries

The color fill scheme entry definition includes:

- The color.
- The fill pattern.
- The caption shown in the Color Fill Legend (such as 100 SF' or more).
- The parameter value that will apply to this entry.

The `Autodesk.Revit.DB.ColorFillSchemeEntry` class provides all functionality needed to create, read, and modify these entries

Code Region: Creating a Color Fill Legend

```
private void CreateColorFill(View v)
{
    Document doc = v.Document;
    ElementId roomCatId = new ElementId(BuiltInCategory.OST_Rooms);
```

```

    using (Transaction t = new Transaction(v.Document, "Create Color Fill Legend"))
    {
        t.Start();

        if (v.GetColorFillSchemeId(roomCatId) == ElementId.InvalidElementId)
        {
            v.SetColorFillSchemeId(
                roomCatId,
                new FilteredElementCollector(doc)
                    .OfClass(typeof(ColorFillScheme))
                    .Cast<ColorFillScheme>()
                    .First(q => q.CategoryId == roomCatId).Id);
        }

        ColorFillLegend.Create(v.Document, v.Id, roomCatId, XYZ.Zero);

        t.Commit();
    }
}

```

Code Sample - Create and Apply a Color Fill Scheme

```

private void NewColorScheme(View v)
{
    Document doc = v.Document;
    ColorFillScheme scheme = doc.GetElement(
        v.GetColorFillSchemeId(
            new ElementId(BuiltInCategory.OST_Rooms))) as ColorFillScheme;

    using (Transaction t = new Transaction(doc, "New Color Scheme"))
    {

```

```

        t.Start();

        ElementId newSchemeId = scheme.Duplicate("Color By Room Finish");

        ColorFillScheme newScheme = doc.GetElement(newSchemeId) as ColorFillScheme;
        newScheme.Title = "Room Finish";
        newScheme.ParameterDefinition = new ElementId(BuiltInParameter.ROOM_FINISH_BASE);

        v.SetColorFillSchemeId(new ElementId(BuiltInCategory.OST_Rooms), newSchemeId);

        t.Commit();
    }
}

```

Code Sample - Add an entry to a ColorFillScheme

```

private void AddColorFillSchemeEntry(ColorFillScheme scheme)
{
    Document doc = scheme.Document;
    ColorFillSchemeEntry entry = new ColorFillSchemeEntry(StorageType.String)
    {
        Color = new Color(25, 0, 0),
        FillPatternId = new FilteredElementCollector(doc)
            .OfClass(typeof(FillPatternElement))
            .Cast<FillPatternElement>()
            .First(a => a.GetFillPattern().IsSolidFill)
            .Id
    };
}

```

```
entry.SetString("Tile");

using (Transaction t = new Transaction(doc, "Add Scheme Entry"))

{

    t.Start();

    scheme.AddEntry(entry);

    t.Commit();

}

}
```

Code Sample - Modify entries of a ColorFillScheme

```
private void ModifyColorFillSchemeEntries(ColorFillScheme scheme)

{

    Document doc = scheme.Document;

    IList<ColorFillSchemeEntry> entries = scheme.GetEntries();

    foreach (ColorFillSchemeEntry entry in entries)

    {

        entry.FillPatternId = new FilteredElementCollector(doc)

            .OfClass(typeof(FillPatternElement))

            .Cast<FillPatternElement>()

            .First(a => a.Name == "Crosshatch")

            .Id;

    }

    using (Transaction t = new Transaction(doc, "Modify entry"))

    {

        t.Start();

        scheme.SetEntries(entries);

        t.Commit();

    }

}
```

```
    }
}
```

3.10 Geometry

The Autodesk.Revit.DB namespace contains many classes related to geometry and graphic-related types used to describe the graphical representation in the API. The geometry-related classes include:

- [GeometryObject class](#) - Includes classes derived from the GeometryObject class.
- [Geometry Helper Classes](#) - Includes classes derived from the APIObject class and value types
- [Geometry Utility Classes](#) - Includes classes to create non-element geometry and to find intersections of solids
- [Collection Classes](#) - Includes classes derived from the IEnumerable or IEnumerator interface.

In this section, you learn how to use various graphic-related types, how to retrieve geometry data from an element, how to transform an element, and more.

3.10.1 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: Retrieve Geometry Data from a Beam](#).

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

```
public void CreateOptions(Application application)
```

```

    {

        Autodesk.Revit.DB.Options geomOption = application.Create.NewGeometry
Options();

        if (null != geomOption)

        {

            geomOption.ComputeReferences = true;

            geomOption.DetailLevel = ViewDetailLevel.Fine;

            // Either the DetailLevel or the View can be set, but not bot
h

            //geomOption.View = commandData.Application.ActiveUIDocument.
Document.ActiveView;

            TaskDialog.Show("Revit", "Geometry Option created successfull
y.");

        }

    }

```

Note: For more information, refer to [Geometry Helper Classes](#).

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)
    {
        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);

    }
}

```

3.10.2 GeometryObject Class

The indexed property `Element.Geometry[]` can be used to pull the geometry of any model element (3D element). This applies both to system family instances such as walls, floors and roofs, and also to family instances of many categories, e.g. doors, windows, furniture, or masses.

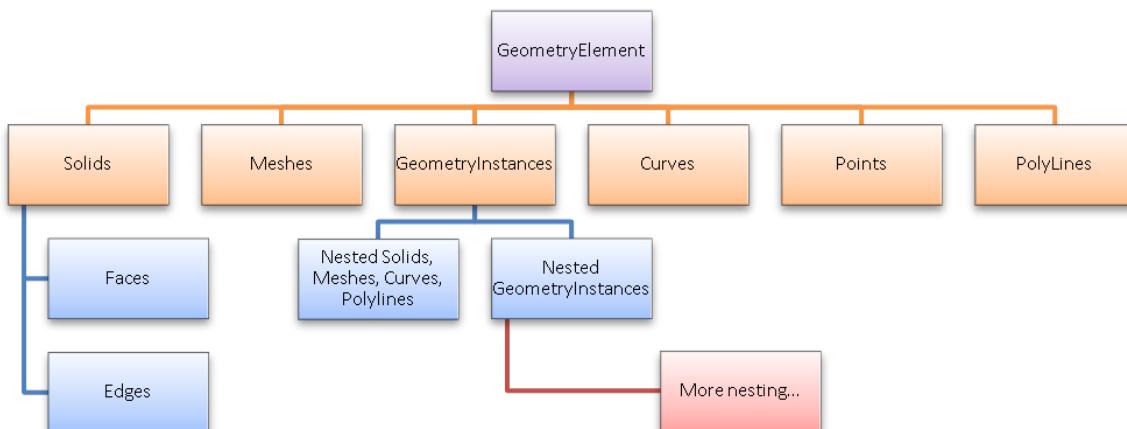
The property `GeometryObject.Id` returns an integer value which may be used to identify the `GeometryObject` in its associated `GeometryElement` when the value is non-negative and not duplicated by other `GeometryObjects` in the associated `GeometryElement`.

The extracted geometry is returned to you as `Autodesk.Revit.DB.GeometryElement`. You can iterate through the geometry members of that element by using the `GetEnumerator()` method.

Typically, the objects returned at the top level of the extracted geometry will be one of:

- [Solids, Faces and Edges](#) – a boundary representation made up of faces and edges
- [Meshes](#) – a 3D array of triangles
- [Curves](#) – a bounded 3D curve
- [Points](#) – a visible point datum at a given 3D location
- [PolyLines](#) – a series of line segments defined by 3D points
- [GeometryInstances](#) – an instance of a geometric element positioned within the element

This figure illustrates the hierarchy of objects found by geometry extraction.



3.10.2.1 Curves

A curve represents a path in 2 or 3 dimensions in the Revit model. Curves may represent the entire extent of an element's geometry (e.g. CurveElements) or may appear as a single piece of the geometry of an element (e.g. the centerline of a wall or duct). Curves and collections of curves are used as inputs in many element creation methods in the API.

3.10.2.1.1 Curve analysis

There are several Curve methods which are tools suitable for use in geometric analysis.

In some cases, these APIs do more than you might expect by a quick review of their names.

Intersect()

The Intersect method allows you compare two curves to find how they differ or how they are similar. It can be used in the manner you might expect, to obtain the point or point(s) where two curves intersect one another, but it can also be used to identify:

- Collinear lines
- Overlapping lines
- Identical curves
- Totally distinct curves with no intersections

The return value identifies these different results, and the output IntersectionSetResult contains information on the intersection point(s).

Project()

The Project method projects a point onto the curve and returns information about the nearest point on the curve, its parameter, and the distance from the projection point.

Tessellate()

This splits the curve into a series of linear segments, accurate within a default tolerance. For Curve.Tessellate(), the tolerance is slightly larger than 1/16". This tolerance of approximation is the tolerance used internally by Revit as adequate for display purposes.

Note that only lines may be split into output of only two tessellation points; non-linear curves will always output more than two points even if the curve has an extremely large radius which might mathematically equate to a straight line.

3.10.2.1.2 Working with Curves

The Curve class provides useful methods for working with curves.

In addition to methods that are useful for analysis, the Curve class provides properties and methods to modify a curve or get basic information about it.

Changing bounds

The MakeBound() method can be used to change the bounds of a curve or to create bounds for a previously unbound curve. MakeUnbound() will make the curve unbound. For both methods, if the curve is marked as read-only (because it was extracted directly from a Revit element or collection/aggregation object), calling this method causes the object to be changed to carry a disconnected copy of the original curve. The modification will not affect the original curve or the object that supplied it.

Graphics style

Curve inherits the GraphicsStyleId read-only property from GeometryObject, which provides the ElementId of the GraphicsStyle assigned to the Curve. The method Curve.SetGraphicsStyleId() can be used to set the GraphicsStyle Id of the Curve. Many methods in the Revit API will not use the graphics style associated to this curve. For example, curves used as portions of the sketch of an element will not read this property. Newly created curve elements will not use this value either, as they inherit their graphical properties from their associated category.

Curve length

Curve has two properties associated with length. The Length property will return the exact length of the curve. I computes the length of the curve using analytical or numeric integration. There is no performance hit for lines and arcs. For a faster approximation, the ApproximateLength property quickly estimates the length of the curve, but it may deviate by a factor of 2 in some cases. This computation is exact for lines and arcs.

3.10.2.1.3 Curve collections

The Revit API uses different types of collections of curves as inputs.

Note: Newer API methods use .NET collections of Curves in place of CurveArray and CurveArrArray.

CurveLoop

A CurveLoop represents a specific chain of curves joined end-to-end. It can represent a closed loop or an open one. The members of the CurveLoop may be directly iterated, as the class implements `IEnumerable<Curve>`. The iteration provides copies of the curves directly contained in the loop; modification of the curves will not affect the curves that are contained in the loop. CurveLoops can be created using:

- `CurveLoop.Create()` - creates a new CurveLoop from a list of curves.
- `CurveLoop.CreateViaCopy()` - creates a new CurveLoop as a copy of an existing CurveLoop.
- `CurveLoop.CreateViaThicken(Curve, double, XYZ)` - creates a new closed CurveLoop by thickening the input curve with respect to a given plane.

- `CurveLoop.CreateViaThicken(CurveLoop, double, XYZ)` -creates a new closed curve loop by thickening the input open curve loop with respect to a given plane.
- `CurveLoop.CreateViaTransform()` - creates a new `CurveLoop` as a transformed copy of the input `CurveLoop`. Note that the thickness parameter for the overloaded `CreateViaThicken()` method must result in a curve which exceed Revit's short curve tolerance (`Application.ShortCurveTolerance`), otherwise an exception will be thrown.

`CurveLoop.Transform()` performs similarly to `CreateViaTransform()`, but it transforms the curves contained within the `CurveLoop` rather than creating a transformed copy.

`CurveLoop.NumberOfCurves` returns the number of curves in the curve loop.

`CurveLoop.CreateViaOffset()` creates a new curve loop that is an offset of the existing curve loop. This can be done in two ways:

- With a single offset distance
- With an array of offset distances. The size of this array must match the size of the curve loop. The curve at position *i* will be offset with the double at position *i* in the array.

CurveArray

This collection class represents an arbitrary collection of curves. Create it using its constructor.

CurveArrArray

This collection class is a collection of `CurveArrays`. When this is used, the organization of the sub-elements of this array have meaning for the method this is passed to; for example, in `NewExtrusion()` multiple `CurveArrays` should represent different closed loops.

3.10.2.1.4 Curve creation

Curves are often needed as inputs to Revit API methods. They can be created a number of ways.

Curves have a number of derived types with static methods for curve creation. The base `Curve` class also has methods for creating new Curves from existing curves.

Curve creation methods prevent creation of curves that are shorter than Revit's tolerance. This tolerance is exposed through the `Application.ShortCurveTolerance` property.

Curve

The `Curve` class has several methods for creating new curves from existing curves.

- `Clone()` - creates a copy of this `Curve`.
- `CreateOffset()` - creates a new curve offset from this one.
- `CreateReversed()` - creates a new curve with the opposite orientation of the existing curve

- `Curve.CreateTransformed()` - creates a new instance of a curve as a transformation of this curve.

Line

There are two static methods for creating a new Line.

- `CreateBound()` - creates a new bound linear curve between two points.
- `CreateUnbound()` - creates a new unbound linear curve given an origin and a direction.

Code Region: Create an unbound linear curve

```
// define start point and direction for unbound line
XYZ startPoint = new XYZ(0, 0, 0);
XYZ directionPt = new XYZ(10, 10, 10);

// create line
Line line = Line.CreateUnbound(startPoint, directionPt);
```

Arc

The overloaded static `Create()` method allows for an Arc to be created in one of three ways:

- based on 3 points

Code Region: Create arc with 3 points

```
public void ThreePointArc()
{
    // Create a new arc using two ends and a point on the curve
    XYZ end0 = new XYZ(1, 0, 0);      // start point of the arc
    XYZ end1 = new XYZ(10, 10, 10); // end point of the arc
    XYZ pointOnCurve = new XYZ(10, 0, 0); // point along arc
```

```
    Arc arc = Arc.Create(end0, end1, pointOnCurve);  
}
```

- based on a plane, radius, and angles

Code Region: Create arc with plane

```
public Arc CreateArcByGivingPlane(Autodesk.Revit.ApplicationServices.Application application, Plane plane)  
{  
    // Create an arc which is placed on the plane and whose center is the  
    // plane's origin  
  
    double radius = 10;  
  
    double startAngle = 0;          // The unit is radian  
  
    double endAngle = 2 * Math.PI;    // this arc will be a circle  
  
    return Arc.Create(plane, radius, startAngle, endAngle);  
}
```

- based on center, radius, angles and two axes

Code Region: Create arc with axes

```
public void CreateArcByCenterRadius()  
{  
    // Create a new arc defined by its center, radius, angles and 2 axes  
  
    double radius = 10;  
  
    double startAngle = 0;          // In radian  
  
    double endAngle = Math.PI;      // In radian  
  
    XYZ center = new XYZ(5, 0, 0);
```

```

XYZ xAxis = new XYZ(1, 0, 0); // The x axis to define the arc plane. Mu
st be normalized

XYZ yAxis = new XYZ(0, 1, 0); // The y axis to define the arc plane. Mu
st be normalized

Arc arc = Arc.Create(center, radius, startAngle, endAngle, xAxis, yAxis);

}

```

Note for the latter two options, if the angle range is equal to or greater than $2 * \pi$, the curve will be automatically converted to an unbounded circle.

Ellipse

The static CreateCurve() method creates an Ellipse given the center, the x vector and y vector radii of the ellipse, the x and y axes to define the plane of the ellipse and the start and end parameters. If the x-radius and y-radius are almost equal it will return an arc, otherwise it will return an ellipse.

Cylindrical Helix

The static Create() method of CylindricalHelix creates a new CylindricalHelix from the base point of the axis, a radius, x vector, z vector, pitch, a start angle to specify the start point of the helix and an end angle to specify the end point of the helix. The z vector is the axis direction and should be perpendicular to the x vector. A positive pitch yields a right-handed helix while a negative pitch yields a left-handed helix.

NURBS

The NurbSpline class represents a NURBS, or Non-Uniform Rational B-Spline, curve. The overloaded static CreateCurve() method offers multiple ways to create a NURBS curve. The first way is using the same calculations that Revit uses when sketching splines in the user interface. It takes a list of control points and weights to create a new NurbSpline. Knots and degree of the spline are computed from the given control points and weights.

A second option also requires a list of control points and weights, but also a list of knots as well as the degree of the NurbSpline. The degree must be 1 or greater. There must be at least degree+1 control points. The size of knots must equal the sum of degree, the size of the control points array and 1. The first degree+1 knots should be identical, as should the last degree+1 knots. The knots in the middle of the sequence must be non-decreasing.

A third option only requires control points and weights. There must be at least 2 control points and the number of weights must be equal to the number of control points. The values of all weights must be positive.

In all cases, the created curve may be a NURBSpline or a simpler curve such as line or arc. This is consistent with Revit expectations that the simplest possible representation of curve should be used in Revit elements.

Hermite Spline

The overloaded static HermiteSpline.Create() method provides two options for creating Hermite splines. The simplest way creates a Hermite spline with default tangency at its endpoints and requires only a list of control points and a flag indicating whether or not the Hermite spline is periodic. The second option creates a Hermite spline with specified tangency at its endpoints. It has an additional parameter of a HermiteSplineTangents object to specify the tangents at the start and/or end of the curve.

3.10.2.1.5 Curve Parameterization

Curves in the Revit API can be described as mathematical functions of an input parameter “u”, where the location of the curve at any given point in XYZ space is a function of “u”.

Curves can be bound or unbound. Unbound curves have no endpoints, representing either an infinite abstraction (an unbound line) or a cyclic curve (a circle or ellipse).

In Revit, the parameter “u” can be represented in two ways:

- A ‘normalized’ parameter. The start value of the parameter is 0.0, and the end value is 1.0. For some curve types, this makes evaluation of the curve along its extents very easy, for example, the midpoint of a line is at parameter 0.5. (Note that for more complex curve equations like Splines this assumption cannot always be made).
- A ‘raw’ parameter. The start and end value of the parameter can be any value. For a given curve, the value of the minimum and maximum raw parameter can be obtained through `Curve.GetEndParameter(int)`. Raw parameters are useful because their units are the same as the Revit default units (feet). So to obtain a location 5 feet along the curve from the start point, you can take the raw parameter at the start and add 5 to it. Raw parameters are also the only way to evaluate unbound curves.

The methods `Curve.ComputeNormalizedParameter()` and `Curve.ComputeRawParameter()` automatically scale between the two parameter types. The method `Curve.IsInside()` evaluates a raw parameter to see if it lies within the bounds of the curve.

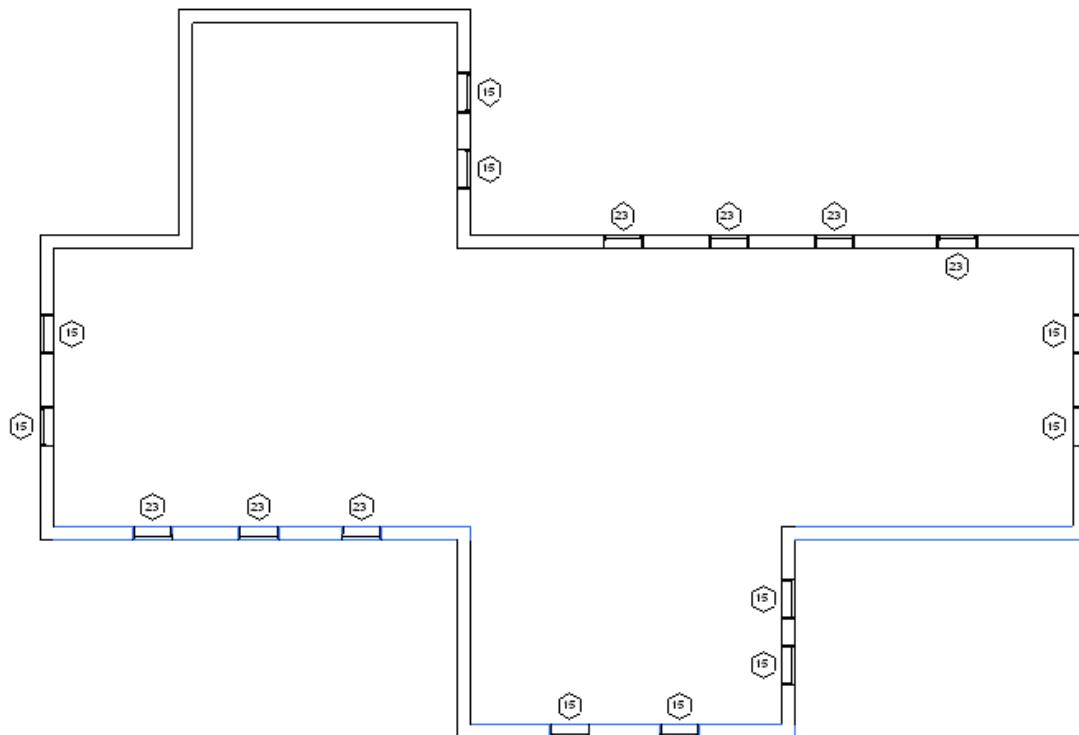
You can use the parameter to evaluate a variety of properties of the curve at any given location:

- The XYZ location of the given curve. This is returned from `Curve.Evaluate()`. Either the raw or normalized parameter can be supplied. If you are also calling `ComputeDerivatives()`, this is also the `.Origin` property of the `Transform` returned by that method.
- The first derivative/tangent vector of the given curve. This is the `.BasisX` property of the `Transform` returned by `Curve.ComputeDerivatives()`.
- The second derivative/normal vector of the given curve. This is the `.BasisY` property of the `Transform` returned by `Curve.ComputeDerivatives()`.

- The *binormal* vector of the given curve, defined as the cross-product of the tangent and normal vector. This is the .BasisZ property of the Transform returned by Curve.ComputeDerivatives().

All of the vectors returned are non-normalized (but you can normalize any vector in the Revit API with XYZ.Normalize()). Note that there will be no value set for the normal and binormal vector when the curve is a straight line. You can calculate a normal vector to the straight line in a given plane using the tangent vector.

The API sample “DirectionCalculation” uses the tangent vector to the wall location curve to find exterior walls that face south:



Finding and highlighting south facing exterior walls

3.10.2.1.6 Curve types

Revit uses a variety of curve types to represent curve geometry in a document.

Curve types in Revit include:

Curve type	Revit API class	Definition	Notes

Bound line	Line	A line segment defined by its endpoints.	Obtain endpoints from Curve.GetEndpoint()
Unbound line	Line	An infinite line defined by a location and direction	<p>Identify these with Curve.IsBound.</p> <p>Evaluate point and tangent vector at raw parameter= 0 to find the input parameters for the equation of the line.</p>
Arc	Arc	A bound circular arc	Begin and end at a certain angle. These angles can be obtained by the raw parameter values at each end of the arc.
Circle	Arc	An unbound circle	<p>Identify with Curve.IsBound.</p> <p>Use raw parameter for evaluation (from 0 to 2π)</p>
Cylindrical helix	CylindricalHelix	A helix wound around a cylinder making constant angle with the axis of the cylinder	Used only in specific applications in stairs and railings, and should not be used or encountered when accessing curves of other Revit elements and geometry.
Elliptical arc	Ellipse	A bound elliptical segment	
Ellipse	Ellipse	An unbound ellipse	Identify with Curve.IsBound. Use raw parameter for evaluation (from 0 to 2π)
NURBS	NurbSpline	A non-uniform	Used for splines sketched in various Revit tools, plus imported geometry

rational B-spline			
Hermite	HermiteSpline	A spline interpolate between a set of points	Used for tools like Curve by Points and flexible ducts/pipes, plus imported geometry
CurveUV		A curve in the 2D parameter space of a surface in 3D space.	This class helps translate geometry (BRep)s from external applications to Revit. Some geometric modelers represent the shape of an edge in a solid (or open shell) by a 3D curve, others use 2D curves in the parameter spaces of the edge's faces, and others might use both. Revit can benefit from being given the CurveUVs for an edge when translating geometry, and this class accommodates the passing of such information.

Mathematical representations of all of the Revit curve types can be found in: [Mathematical representation of face types](#).

3.10.2.1.7 Mathematical representations of curve types

This section describes the curve types encountered in Revit geometry, their properties, and their mathematical representations.

Bound lines

Bound lines are defined by their endpoints. In the Revit API, obtain the endpoints of the line from the Curve-level GetEndPoint() method.

The equation for a point on a bound line in terms of the normalized parameter "u" and the line endpoints is

$$\mathbf{P}(u) = \mathbf{P}_1 + u(\mathbf{P}_2 - \mathbf{P}_1)$$

Unbound lines

Unbound lines are handled specially in the Revit API. Most curve properties cannot be used, however, Evaluate() and ComputeDerivatives() can be used to obtain locations along the curve when a raw parameter is provided.

The equation for a point for an unbound line in terms of the raw parameter “u” and the line origin and normalized direction vector is

$$\mathbf{P}(u) = \mathbf{P}_0 + u\mathbf{V}$$

Arcs and Circles

Arcs and Circles are represented in the Revit API by the Arc class. They are defined in terms of their radius, center and vector normal to the plane of the arc, which are accessible in the Revit API directly from the Arc class as properties.

Circles have the `IsBound` property set to true. This means they can only be evaluated by using a raw parameter (range from 0 to 2π), and the equation for a point on the circle in terms of the raw parameter is

$$\mathbf{P}(u) = \mathbf{C} + r(\mathbf{n}_x \cos u + \mathbf{n}_y \sin u)$$

where the assumption is made that the circle lies in the XY plane.

Arcs begin and end at a certain angle. These angles can be obtained by the raw parameter values at each end of the arc, and angular values between these values can be plugged into the same equation as above.

Cylindrical Helices

Cylindrical helices are represented in the Revit API by the CylindricalHelix class. They are defined in terms of the base point of the axis of the cylinder around which the helix is wound, radius, x and y vectors, pitch, and start and end angles.

Ellipse and elliptical arcs

Ellipses and elliptical arcs segments are represented in the Revit API by the Ellipse class. Similar to arcs and circles, they are defined in a given plane in terms of their X and Y radii, center, and vector normal to the plane of the ellipse.

Full ellipses have the `IsBound` property set to true. Similar to circles, they can be evaluated via raw parameter between 0 and 2π :

$$\mathbf{P}(u) = \mathbf{C} + \mathbf{n}_x r_x \cos u + \mathbf{n}_y r_y \sin u$$

NurbSpline

NURBS (nonuniform rational B-splines) are used for spline segments sketched by the user as curves or portions of 3D object sketches. They are also used to represent some types of imported geometry data.

The data for the NurbSpline include:

- The control points array, of length $n+1$
- The weights array, also of length $n+1$

- The curve degree, whose value is equal to one less than the curve order (k)
- The knot vector, of length n + k + 1

$$\mathbf{P}(u) = \frac{\sum_{i=0}^n \mathbf{P}_i w_i N_{i,k}(u)}{\sum_{i=0}^n w_i N_{i,k}(u)}, \quad 0 \leq u \leq u_{max}$$

NurbSplines as used in Revit's sketching tools can be generated from the control points and degree alone using an algorithm. The calculations performed by Revit's algorithm can be duplicated externally, see this sample below:

```
public void Nurb(ModelCurve curve)
{
    NurbSpline spline = curve.GeometryCurve as NurbSpline;
    DoubleArray knots = spline.Knots;

    // Convert to generic collection
    List<double> knotList = new List<double>();
    for(int i = 0; i < knots.Size; i++)
    {
        knotList.Add(knots.get_Item(i));
    }

    // Preparation - get distance between each control point
    IList<XYZ> controlPoints = spline.CtrlPoints;
    int numControlPoints = controlPoints.Count;
    double[] chordLengths = new double[numControlPoints - 1];
    for(int iControlPoint = 1; iControlPoint < numControlPoints; ++iControlPoint)
    {
        double chordLength =
            controlPoints[iControlPoint].DistanceTo(controlPoints[iControlPoint - 1]);
        chordLengths[iControlPoint - 1] = chordLength;
    }
}
```

```
int degree = spline.Degree;

int order = degree + 1;

int numKnots = numControlPoints + order;

double[] computedKnots = new double[numKnots];

int iKnot = 0;

// Start knot with multiplicity degree + 1.

double startKnot = 0.0;

double knot = startKnot;

for(iKnot = 0; iKnot < order; ++iKnot)

{

    computedKnots[iKnot] = knot;

}

// Interior knots based on chord lengths

double prevKnot = knot;

for(/*blank*/; iKnot <= numControlPoints; ++iKnot)

    // Last loop computes end knot but does not set interior knot.

{

    double knotIncrement = 0.0;

    for (int jj = iKnot - order; jj < iKnot - 1; ++jj)

    {

        knotIncrement += chordLengths[jj];

    }

}
```

```

        knotIncrement /= degree;

        knot = prevKnot + knotIncrement;

        if (iKnot < numControlPoints)

            computedKnots[iKnot] = knot;

        else

            break; // Leave "knot" set to the end knot; do not increment "ii".

    }

// End knot with multiplicity degree + 1.

for(/*blank*/; iKnot < numKnots; ++iKnot)

{
    computedKnots[iKnot] = knot;
}

}

```

HermiteSpline

Hermite splines are used for curves which are interpolated between a set of control points, like Curve by Points and flexible ducts and pipes in MEP. They are also used to represent some types of imported geometry data. In the Revit API, the HermiteSpline class offers access to the arrays of points, tangent vectors and parameters through the ControlPoints, Tangents, and Parameters properties.

The equation for the curve between two nodes in a Hermite spline is

$\mathbf{P}(u) = h_{00}(u)\mathbf{P}_k + (u_{k+1} - u_k)h_{10}(u)\mathbf{M}_k + h_{01}(u)\mathbf{P}_{k+1} + (u_{k+1} - u_k)h_{11}(u)\mathbf{M}_{k+1}$, where
 \mathbf{P}_k and \mathbf{P}_{k+1} represent the points at each node, \mathbf{M}_k and \mathbf{M}_{k+1} the tangent vectors, and u_k and u_{k+1} the parameters at the nodes, and the basis functions are:

$$h_{00}(u) = 2u^3 - 3u^2 + 1 \quad h_{10}(u) = u^3 - 2u^2 + u$$

$$h_{01}(u) = -2u^3 + 3u^2$$

3.10.2.2 *GeometryInstances*

A GeometryInstance represents a set of geometry stored by Revit in a default configuration, and then transformed into the proper location as a result of the properties of the element. The most common situation where GeometryInstances are encountered is in Family instances. Revit uses GeometryInstances to allow it to store a single copy of the geometry for a given family and reuse it in multiple instances.

Note that not all Family instances will include GeometryInstances. When Revit needs to make a unique copy of the family geometry for a given instance (because of the effect of local joins, intersections, and other factors related to the instance placement) no GeometryInstance will be encountered; instead the Solid geometry will be found at the top level of the hierarchy.

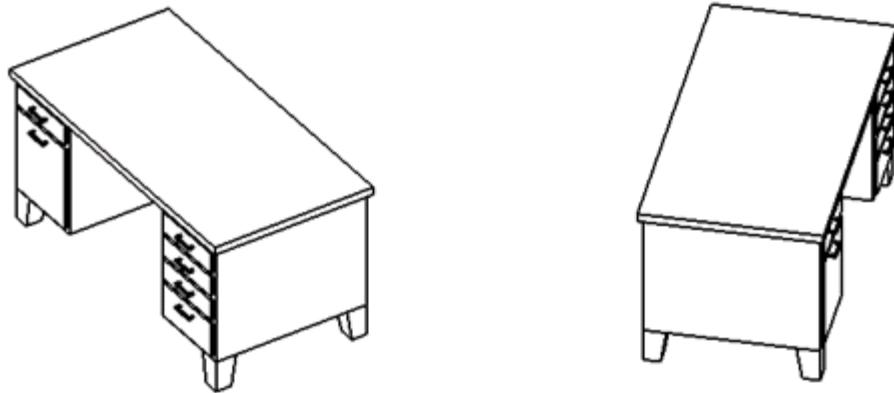
A GeometryInstance offers the ability to read its geometry through the GetSymbolGeometry() and GetInstanceGeometry() methods. These methods return another Autodesk.Revit.DB.GeometryElement which can be parsed just like the first level return.

GetSymbolGeometry() returns the geometry represented in the coordinate system of the family. Use this, for example, when you want a picture of the “generic” table without regards to the orientation and placement location within the project. This is also the only overload which returns the actual Revit geometry objects to you, and not copies. This is important because operations which use this geometry as input to creating other elements (for example, dimensioning or placement of face-based families) require the reference to the original geometry.

GetInstanceGeometry() returns the geometry represented in the coordinate system of the project where the instance is placed. Use this, for example, when you want a picture of the specific geometry of the instance in the project (for example, ensuring that tables are placed parallel to the walls of the room). This always returns a copy of the element geometry, so while it would be suitable for implementation of an exporter or a geometric analysis tool, it would not be appropriate to use this for the creation of other Revit elements referencing this geometry.

There are also overloads for both GetInstanceGeometry() and GetSymbolGeometry() that transform the geometry by any arbitrary coordinate system. These methods always return copies similar to GetInstanceGeometry().

The GeometryInstance also stored a transformation from the symbol coordinate space to the instance coordinates. This transform is accessible as the Transform property. It is also the transformation used when extracting a the copy of the geometry via GetInstanceGeometry(). For more details, refer to [Geometry Helper Classes](#).



2 family instances placed with different transforms - the same geometry will be acquired from both

Instances may be nested several levels deep for some families. If you encounter nested instances they may be parsed in a similar manner as the first level instance.

Two samples are presented to explain how geometry of instances can be parsed.

In this sample, curves are extracted from the GeometryInstance method GetInstanceGeometry().

Code Region: 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)
```

```
{  
    // Get the geometry instance which contains the geometry information  
    Autodesk.Revit.DB.GeometryInstance instance =  
        geoObject as Autodesk.Revit.DB.GeometryInstance;  
  
    if (null != instance)  
    {  
        GeometryElement instanceGeometryElement = instance.GetInstanceGeomet  
ry();  
  
        foreach (GeometryObject o in instanceGeometryElement)  
        {  
            // Try to find curves  
            Curve curve = o as Curve;  
            if (curve != null)  
            {  
                // The curve is already transformed into the project coordinat  
e system  
            }  
        }  
    }  
}
```

In this sample, the solids are obtained from an instance using `GetSymbolGeometry()`. The constituent points are then transformed into the project coordinate system using the `GeometryInstance.Transform`.

Code Region: 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)
    {
        // Get the geometry instance which contains the geometry info
        // rmation
        Autodesk.Revit.DB.GeometryInstance instance =
        geoObject as Autodesk.Revit.DB.GeometryInstance;
        if (null != instance)
        {
            GeometryElement instanceGeometryElement = instance.GetSymbolGeometry();
            foreach (GeometryObject instObj in instanceGeometryElement)
            {
                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);
    }
}

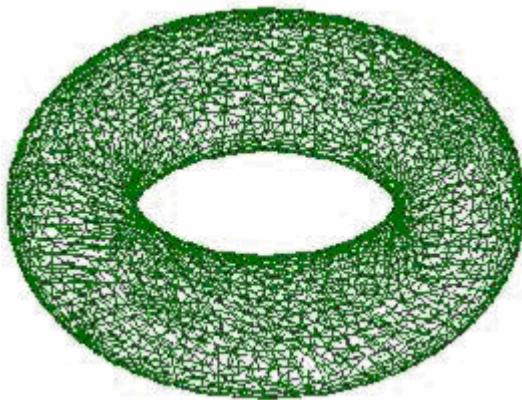
foreach (Edge edge in solid.Edges)
{
    foreach (XYZ ii in edge.Tessellate())
    {
        XYZ point = ii;

        XYZ transformedPoint = instTransform.OfPoint(point);
    }
}
}
```

Note: For more details about the retrieved geometry of family instances, refer to [Example: Retrieve Geometry Data from a Beam](#).

3.10.2.3 Meshes

A mesh is a collection of triangular boundaries which collectively forms a 3D shape. Meshes are typically encountered inside Revit element geometry if those elements were created from certain import operations and also are used in some native Revit elements such as TopographySolid. You can also obtain Meshes as the result of calls to Face.Triangulate() for any given Revit face. The property `Mesh.IsClosed` checks if each edge in the mesh belongs to at least two faces.



A mesh representing a torus

The following code sample illustrates how to get the geometry of a Revit face as a Mesh:

Code region: Extracting the geometry of a mesh

```
private void GetTrianglesFromFace(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);
```

```
    }  
}
```

Note: The approximation tolerance used for Revit display purposes is used by the parameterless overload of the Triangulate() method (used above) when constructing the Mesh. The overload of Triangulate() that takes a double allows a level of detail to be set between 0 (coarser) and 1 (finer).

3.10.2.4 Points

A point represents a visible coordinate in 3D space.

Points are typically encountered in mass family elements like ReferencePoint. The Point class provides read access to its coordinates, and an ability to obtain a reference to the point for use as input to other functions.

Point creation

There are two ways to create Points:

- Create(XYZ) - creates a Point at the given coordinates.
- Create(XYZ, ElementId) - creates a Point at given coordinates and assigns it a color based on the GraphicsStyle element (specified by ElementId).

3.10.2.5 PolyLines

A polyline is a collection of line segments defined by a set of coordinate points. These are typically encountered in imported geometry. The PolyLine class offers the ability to read the coordinates:

- PolyLine.NumberOfCoordinates – the number of points in the polyline
- PolyLine.GetCoordinate() – gets a coordinate by index
- PolyLine.GetCoordinates() – gets a collection of all coordinates in the polyline
- PolyLine.Evaluate() – given a normalized parameter (from 0 to 1) evaluates an XYZ point along the extents of the entire polyline

3.10.2.6 Solids, Faces and Edges

A Solid is a Revit API object which represents a collection of faces and edges. Typically in Revit these collections are fully enclosed volumes, but a shell or partially bounded volume can also be encountered. Note that sometimes the Revit geometry will contain unused solids containing zero edges and faces. Check the Edges and Faces members to filter out these solids.

The Revit API offers the ability to read the collections of faces and edges, and also to compute the surface area, volume, and centroid of the solid.

3.10.2.6.1 Edges

The `Edge` class represents the edge of a 3d solid. Edges are defined by intersections of surfaces that form faces of the solid. They have arbitrary parameterization that is normalized from 0 to 1. The `EdgeEndPoint` class represents the start or the end point of an Edge and `SolidUtils.FindAllEdgeEndPointsAtVertex()` finds all EdgeEndPoints at a vertex identified by the input EdgeEndPoint.

3.10.2.6.2 Edge and face parameterization

Edges are boundary curves for a given face.

Iterate the edges of a Face using the `EdgeLoops` property. Each loop represents one closed boundary on the face. Edges are always parameterized from 0 to 1. It is possible to extract the `Curve` representation of an Edge with the `Edge.AsCurve()` and `Edge.AsCurveFollowingFace()` functions.

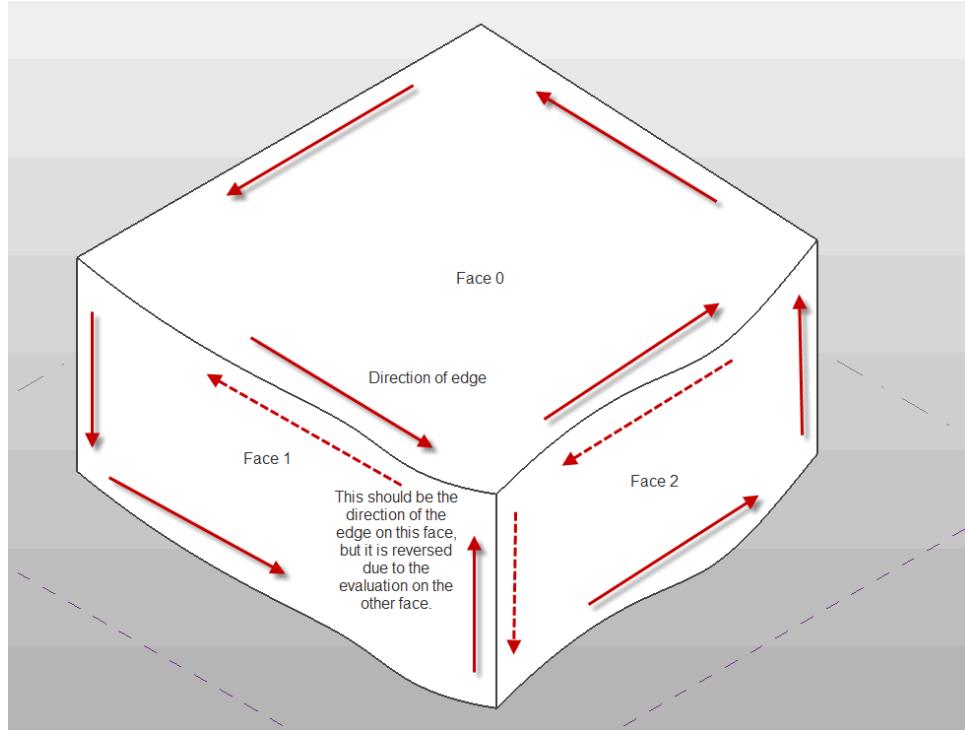
An edge is usually defined by computing intersection of two faces. But Revit doesn't recompute this intersection when it draws graphics. So the edge stores a list of points - end points for a straight edge and a tessellated list for a curved edge. The points are parametric coordinates on the two faces. These points are available through the `TessellateOnFace()` method.

Sections produce "cut edges". These are artificial edges - not representing a part of the model-level geometry, and thus do not provide a Reference.

Edge direction

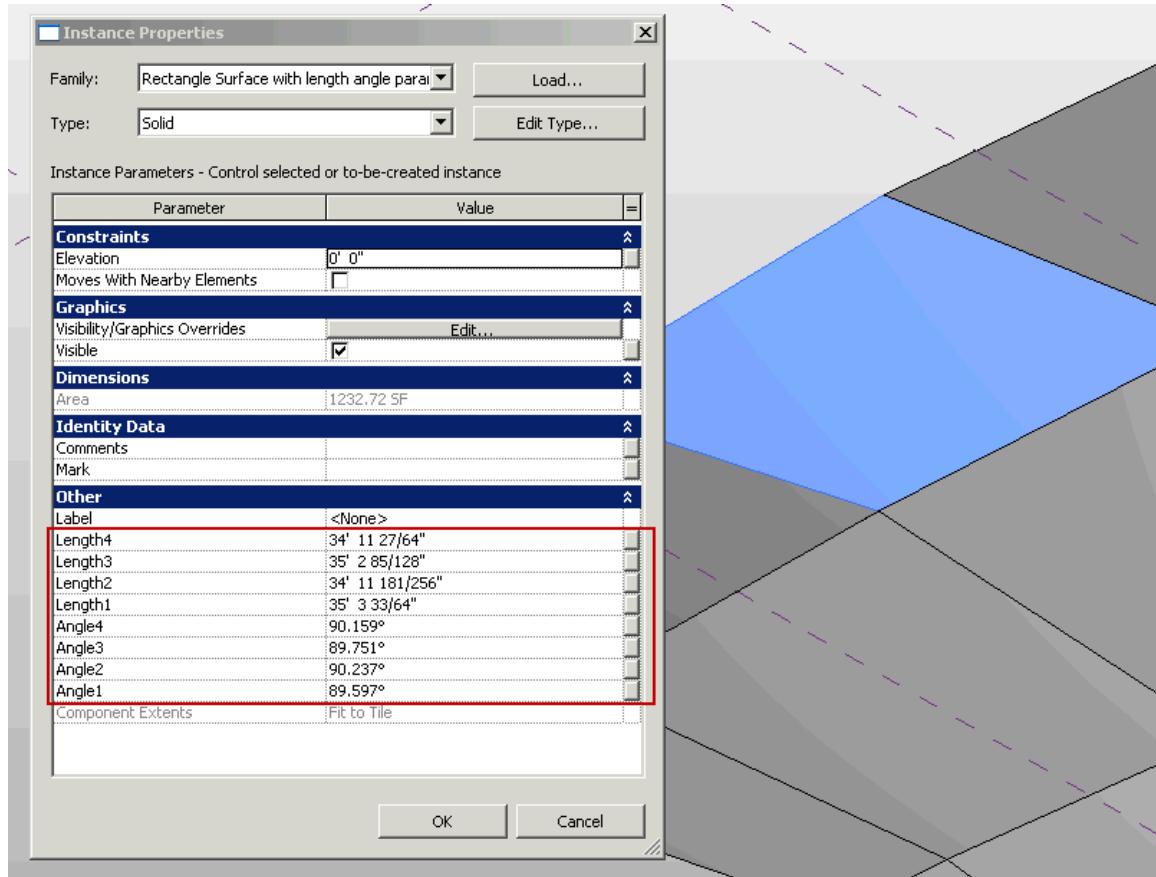
Direction is normally clockwise on the first face (first representing an arbitrary face which Revit has identified for a particular edge). But because two different faces meet at one particular edge, and the edge has the same parametric direction regardless of which face you are concerned with, sometimes you need to figure out the direction of the edge on a particular face.

The figure below illustrated how this works. For Face 0, the edges are all parameterized clockwise. For Face 1, the edge shared with Face 0 is not re-parameterized; thus with respect to Face 1 the edge has a reversed direction, and some edges intersect where both edges' parameters are 0 (or 1).



Edge parameterization

The API sample “PanelEdgeLengthAngle” shows how to recognize edges that are reversed for a given face. It uses the tangent vector at the edge endpoints to calculate the angle between adjacent edges, and detect whether or not to flip the tangent vector at each intersection to calculate the proper angle.

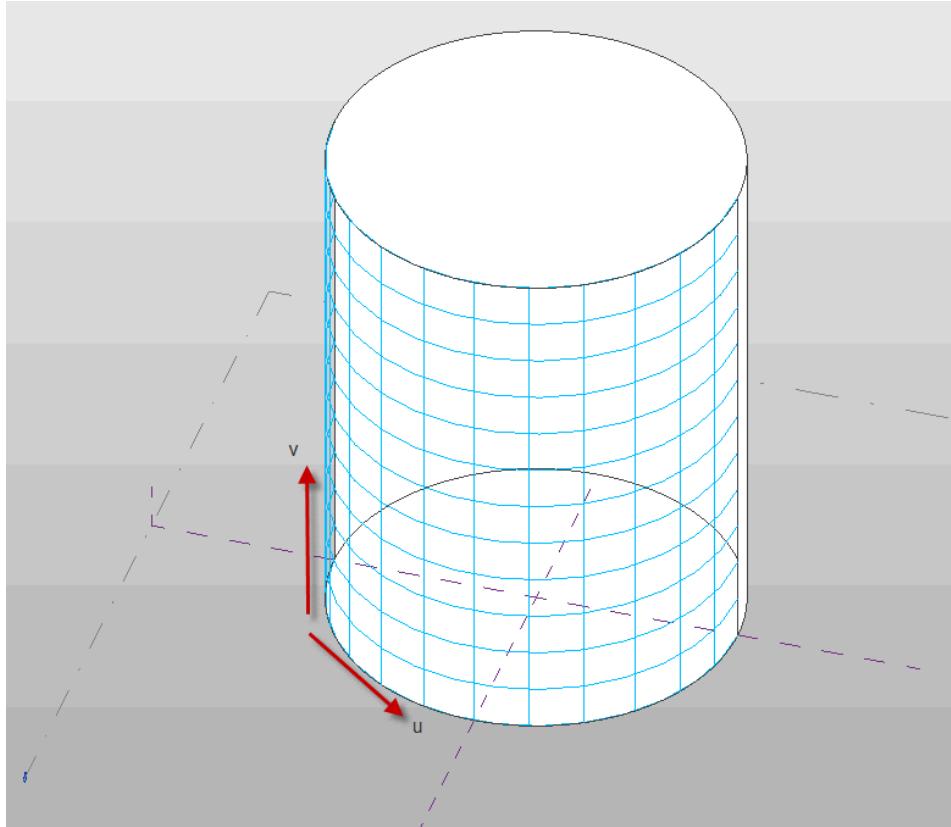


PanelEdgeLengthAngle results

3.10.2.6.3 Faces

Faces in the Revit API can be described as mathematical functions of two input parameters "u" and "v", where the location of the face at any given point in XYZ space is a function of the parameters.

The U and V directions are automatically determined based on the shape of the given face. Lines of constant U or V can be represented as gridlines on the face, as shown in the example below:



U and V gridlines on a cylindrical face

You can use the UV parameters to evaluate a variety of properties of the face at any given location:

- Whether the parameter is within the boundaries of the face, using Face.IsInside()
- The XYZ location of the given face at the specified UV parameter value. This is returned from Face.Evaluate(). If you are also calling ComputeDerivatives(), this is also the .Origin property of the Transform returned by that method.
- The tangent vector of the given face in the U direction. This is the .BasisX property of the Transform returned by Face.ComputeDerivatives()
- The tangent vector of the given face in the V direction. This is the .BasisY property of the Transform returned by Face.ComputeDerivatives().
- The normal vector of the given face. This is the .BasisZ property of the Transform returned by Face.ComputeDerivatives().
- The second derivative with respect to U. This is the .UUDerivative property of the FaceSecondDerivatives returned by Face.ComputeSecondDerivatives().
- The second derivative with respect to V. This is the .VVDerivative of the FaceSecondDerivatives returned by Face.ComputeSecondDerivatives().
- The mixed derivative of the given face. This is the .MixedDerivative of the FaceSecondDerivatives returned by Face.ComputeSecondDerivatives().

All of the vectors returned are non-normalized.

3.10.2.6.4 Face analysis

There are several Face methods which are tools suitable for use in geometric analysis.

Intersect()

The Intersect method computes the intersection between the face and a curve. It can be used in to identify:

- The intersection point(s) between the two objects
- The edge nearest the intersection point, if there is an edge close to this location
- Curves totally coincident with a face
- Curves and faces which do not intersect

Project()

The Project method projects a point on the input face, and returns information on the projection point, the distance to the face, and the nearest edge to the projection point.

Triangulate()

The Triangulate method obtains a triangular mesh approximating the face. There are two overloads to this method. The parameterless method is similar to Curve.Tessellate() in that the mesh's points are accurate within the input tolerance used by Revit (slightly larger than 1/16"). The second Triangulate method accepts a level of detail as an argument ranging from 0 (coarse) to 1 (fine).

3.10.2.6.5 Face Splitting

A face may be split into regions by the Split Face command. The Face.HasRegions property will report if the face contains regions created with the Split Face command, while the Face.GetRegions() method will return a list of faces, one for the main face of the object hosting the Split Face (such as wall or floor) and one face for each Split Face region.

The FaceSplitter class represents an element that splits a face. The FaceSplitter.SplitElementId property provides the id of the element whose face is split by this element. The FaceSplitter class can be used to filter and find these faces by type as shown below.

Code Region: Find face splitting elements

```
public void FindSplitting(Application app, Document doc)
```

```
{  
    Autodesk.Revit.DB.Options opt = app.Create.NewGeometryOptions();  
  
    opt.ComputeReferences = true;  
  
    opt.IncludeNonVisibleObjects = true;  
  
    FilteredElementCollector collector = new FilteredElementCollector(doc);  
  
    ICollection<FaceSplitter> splitElements = collector.OfClass(typeof(FaceSplitter)).Cast<FaceSplitter>().ToList();  
  
    foreach(FaceSplitter faceSplitter in splitElements)  
    {  
  
        Element splitElement = doc.GetElement(faceSplitter.SplitElementId);  
  
        Autodesk.Revit.DB.GeometryElement geomElem = faceSplitter.get_Geometry(opt);  
  
        foreach (GeometryObject geomObj in geomElem)  
        {  
  
            Line line = geomObj as Line;  
  
            if (line != null)  
            {  
  
                XYZ end1 = line.GetEndPoint(0);  
  
                XYZ end2 = line.GetEndPoint(1);  
  
                double length = line.ApproximateLength;  
  
            }  
        }  
    }  
}
```

3.10.2.6.6 Face types

Revit uses a variety of curve types to represent face geometry in a document. These include:

Face type	Revit API class	Definition	Notes
Plane	PlanarFace	A plane defined by the origin and unit vectors in U and V.	
Cylinder	CylindricalFace	A face defined by extruding a circle along an axis.	.Radius provides the “radius vectors” – the unit vectors of the circle multiplied by the radius value.
Cone	ConicalFace	A face defined by rotation of a line about an axis.	.Radius provides the “radius vectors” – the unit vectors of the circle multiplied by the radius value.
Revolved face	RevolvedFace	A face defined by rotation of an arbitrary curve about an axis.	.Radius provides the unit vectors of the plane of rotation, there is no “radius” involved.
Ruled surface	RuledFace	A face defined by sweeping a line between two profile curves, or one profile curve and one point.	Both curve(s) and point(s) can be obtained as properties.
Hermite face	HermiteFace	A face defined by Hermite interpolation between points.	

Mathematical representations of all of the Revit face types can be found in: [Mathematical representation of face types](#).

3.10.2.6.7 Mathematical representation of face types

This section describes the face types encountered in Revit geometry, their properties, and their mathematical representations.

PlanarFace

A plane defined by origin and unit vectors in U and V. Its parametric equation is

$$\mathbf{P}(u, v) = \mathbf{P}_0 + u\mathbf{n}_u + v\mathbf{n}_v$$

CylindricalFace

A face defined by extruding a circle along an axis. The Revit API provides the following properties:

- The origin of the face.
- The axis of extrusion.
- The “radius vectors” in X and Y. These vectors are the circle’s unit vectors multiplied by the radius of the circle. Note that the unit vectors may represent either a right handed or left handed control frame.

The parametric equation for this face is:

$$\mathbf{P}(u, v) = \mathbf{P}_0 + \mathbf{r}_x \cos(u) + \mathbf{r}_y \sin(u) + v\mathbf{n}_{axis}$$

ConicalFace

A face defined by rotation of a line about an axis. The Revit API provides the following properties:

- The origin of the face.
- The axis of the cone.
- The “radius vectors” in X and Y. These vectors are the unit vectors multiplied by the radius of the circle formed by the revolution. Note that the unit vectors may represent either a right handed or left handed control frame.
- The half angle of the face.

The parametric equation for this face is:

$$\mathbf{P}(u, v) = \mathbf{P}_0 + v[\sin(\alpha)(\mathbf{r}_x \cos(u) + \mathbf{r}_y \sin(u)) + \cos(\alpha)\mathbf{n}_{axis}]$$

RevolvedFace

A face defined by rotation of an arbitrary curve about an axis. The Revit API provides the following properties:

- The origin of the face
- The axis of the face
- The profile curve
- The unit vectors for the rotated curve (incorrectly called “Radius”)

The parametric equation for this face is:

$$\mathbf{P}(u, v) = \mathbf{P}_0 + \mathbf{C}(v)[\mathbf{r}_x \cos(u) + \mathbf{r}_y \sin(u) + \mathbf{n}_{axis}]$$

RuledFace

A ruled surface is created by sweeping a line between two profile curves or between a curve and a point. The Revit API provides the curve(s) and point(s) as properties.

The parametric equation for this surface is:

$\mathbf{P}(u, v) = \mathbf{C}_1(u) + v(\mathbf{C}_2(u) - \mathbf{C}_1(u))$ if both curves are valid. If one of the curves is replaced with a point, the equations simplify to one of:

$\mathbf{P}(u, v) = \mathbf{P}_1 + v(\mathbf{C}_2(u) - \mathbf{P}_1)$ $\mathbf{P}(u, v) = \mathbf{C}_1(u) + v(\mathbf{P}_2 - \mathbf{C}_1(u))$ A ruled face with no curves and two points is degenerate and will not be returned.

HermiteFace

A cubic Hermite spline face. The Revit API provides:

- Arrays of the u and v parameters for the spline interpolation points
- An array of the 3D points at each node (the array is organized in increasing u, then increasing v)
- An array of the tangent vectors at each node
- An array of the twist vectors at each node

The parametric representation of this surface, between nodes (u_1, v_1) and (u_2, v_2) is:

$\mathbf{P}(u, v) = \mathbf{U}^T [\mathbf{M}_H] [\mathbf{B}] [\mathbf{M}_H]^T \mathbf{V}$ Where $\mathbf{U} = [u^3 \ u^2 \ u \ 1]^T$, $\mathbf{V} = [v^3 \ v^2 \ v \ 1]^T$, \mathbf{M}_H is the Hermite matrix:

$$[\mathbf{M}_H] = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

And B is coefficient matrix obtained from the face properties at the interpolation points:

$$[\mathbf{B}] = \begin{bmatrix} \mathbf{P}_{u_1v_1} & \mathbf{P}_{u_1v_2} & \mathbf{P}'_{v(u_1v_1)} & \mathbf{P}'_{v(u_1v_2)} \\ \mathbf{P}_{u_2v_1} & \mathbf{P}_{u_2v_2} & \mathbf{P}'_{v(u_2v_1)} & \mathbf{P}'_{v(u_2v_2)} \\ \mathbf{P}'_{u(u_1v_1)} & \mathbf{P}'_{u(u_1v_2)} & \mathbf{P}'_{uv(u_1v_1)} & \mathbf{P}'_{uv(u_1v_2)} \\ \mathbf{P}'_{u(u_2v_1)} & \mathbf{P}'_{u(u_2v_2)} & \mathbf{P}'_{uv(u_2v_1)} & \mathbf{P}'_{uv(u_2v_2)} \end{bmatrix}$$

3.10.2.6.8 Solid analysis

Intersection between solid and curve

The method `Solid.IntersectWithCurve()` calculates the intersection between a closed volume solid and a curve. The `SolidCurveIntersectionOptions` class can specify whether the results from the `IntersectWithCurve()` method will include curve segments inside the solid volume or outside.

The curve segments inside the solid will include curve segments coincident with the face(s) of the solid. Both the curve segments and the parameters of the segments are available in the results.

The following example uses the `IntersectWithCurve()` method to calculate the length of rebar that lies within a column.

Code Region: Finding intersection between solid and curve

```
private void FindColumnRebarIntersections(Document document, FamilyInstance column)

{
    // We will be computing the total length of the rebar inside the column
    double totalRebarLengthInColumn = 0;

    // Find rebar hosted by this column
    RebarHostData rebarHostData = RebarHostData.GetRebarHostData(column);
    if (rebarHostData == null)
    {
        return;
    }

    IList<Rebar> rebars = rebarHostData.GetRebarsInHost();
    if (rebars.Count == 0)
    {
        return;
    }

    // Retrieve geometry of the column
    Options geomOptions = new Options();
```

```
geomOptions.ComputeReferences = true;
geomOptions.DetailLevel = ViewDetailLevel.Fine;
GeometryElement elemGeometry = column.get_Geometry(geomOptions);

// Examine all geometry primitives of the column
foreach (GeometryObject elemPrimitive in elemGeometry)
{
    // Skip objects that are not geometry instances
    GeometryInstance gInstance = elemPrimitive as GeometryInstance;
    if (gInstance == null)
    {
        continue;
    }

    // Retrieve geometry of each found geometry instance
    GeometryElement instGeometry = gInstance.GetInstanceGeometry();
    foreach (GeometryObject instPrimitive in instGeometry)
    {
        // Skip non-solid sobject
        Solid solid = instPrimitive as Solid;
        if (solid == null)
        {
            continue;
        }
    }
}
```

```
SolidCurveIntersectionOptions intersectOptions = new SolidCurveIntersectionOptions();

foreach (Rebar rebar in rebars)

{

    // Get the centerlines for the rebar to find their intersection with the column

    bool selfIntersection = false;

    bool suppressHooks = false;

    bool suppressBends = false;

    IList<Curve> curves = rebar.GetCenterlineCurves(selfIntersection, suppressHooks, suppressBends, MultiplanarOption.IncludeOnlyPlanarCurves, 0);

    // Examine every segment of every curve of the centerline

    foreach (Curve curve in curves)

    {

        SolidCurveIntersection intersection = solid.IntersectWithCurve(curve, intersectOptions);

        for (int segment = 0; segment <= intersection.SegmentCount - 1; segment++)

        {

            // Calculate length of the rebar that is inside the column

            Curve curveInside = intersection.GetCurveSegment(segment);

            double rebarLengthInColumn = curveInside.Length;

            totalRebarLengthInColumn = totalRebarLengthInColumn + rebarLengthInColumn;

        }

    }

}
```

```
        }
    }
}

}
```

3.10.2.6.9 Solid and face creation

Solids and faces are sometimes used as inputs to other utilities. The Revit API provides several routines which can be used to create such geometry from scratch or to derive it from other inputs.

Transformed geometry

The method

- `GeometryElement.GetTransformed()`

returns a copy of the input geometry element with a transformation applied. Because this geometry is a copy, its members cannot be used as input references to other Revit elements, but it can be used geometric analysis and extraction.

Geometry creation utilities

The `GeometryCreationUtilities` class is a utility class that allows construction of basic solid shapes:

- Extrusion
- Loft
- Revolution
- Sweep
- Blend
- SweptBlend

The resulting geometry is not added to the document as a part of any element. However, the created Solid can be used as inputs to other API functions, including:

- As the input face(s) to the methods in the Analysis Visualization framework (`SpatialFieldManager.AddSpatialFieldPrimitive()`) – this allows the user to visualize the created shape relative to other elements in the document

- As the input solid to finding 3D elements by intersection
- As one or more of the inputs to a Boolean operation
- As a part of a geometric calculation (using, for example, Face.Project(), Face.Intersect(), or other Face, Solid, and Edge geometry methods)

The following example uses the GeometryCreationUtilities class to create cylindrical shapes based on a location and height. This might be used, for example, to create volumes around the ends of a wall in order to find other walls within close proximity to the wall end points:

Code Region: Create cylindrical solid

```
public void MakeCyl(Wall wall, XYZ endPoint, double height, double elevation)
{
    // Build cylinder centered at wall end point, extending 3' in diameter
    CurveLoop cylinderLoop = new CurveLoop();
    XYZ arcCenter = new XYZ(endPoint.X, endPoint.Y, elevation);
    Application application = wall.Document.Application;
    Arc firstArc = Arc.Create(arcCenter, 1.5, 0, Math.PI, XYZ.BasisX, XYZ.BasisY);
    Arc secondArc = Arc.Create(arcCenter, 1.5, Math.PI, 2 * Math.PI, XYZ.BasisX, XYZ.BasisY);

    cylinderLoop.Append(firstArc);
    cylinderLoop.Append(secondArc);

    List<CurveLoop> singleLoop = new List<CurveLoop>();
    singleLoop.Add(cylinderLoop);

    Solid proximityCylinder = GeometryCreationUtilities.CreateExtrusionGeometry(singleLoop, XYZ.BasisZ, height);
}
```

Boolean operations

The BooleanOperationsUtils class provides methods for combining a pair of solid geometry objects.

The ExecuteBooleanOperation() method takes a copy of the input solids and produces a new solid as a result. Its first argument can be any solid, either obtained directly from a Revit element or created via another operation like GeometryCreationUtils.

The method ExecuteBooleanOperationModifyingOriginalSolid() performs the boolean operation directly on the first input solid. The first input must be a solid which is not obtained directly from a Revit element. The property GeometryObject.IsElementGeometry can identify whether the solid is appropriate as input for this method.

Options to both methods include the operations type: Union, Difference, or Intersect. The following example demonstrates how to get the intersection of two solids and then find the volume.

Code Region: Volume of Solid Intersection

```
private void ComputeIntersectionVolume(Solid solidA, Solid solidB)
{
    Solid intersection = BooleanOperationsUtils.ExecuteBooleanOperation(solid
A, solidB, BooleanOperationsType.Intersect);

    double volumeOfIntersection = intersection.Volume;
}
```

The methods CutWithHalfSpace() and CutWithHalfSpaceModifyingOriginalSolid() produce a solid which is the intersection of the input Solid with the half-space on the positive side of the given Plane. The positive side of the plane is the side to which Plane.Normal points. The first method creates a new Solid with the results, while the second modifies the existing solid (which must be a solid created by the application instead of one obtained from a Revit element).

3.10.3 Geometry Helper Classes

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.

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:

$$\begin{pmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} & \text{Origin.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} & \text{Origin.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} & \text{Origin.Z} \end{pmatrix} \times \begin{pmatrix} \text{XYZ.X} \\ \text{XYZ.Y} \\ \text{XYZ.Z} \\ 1 \end{pmatrix}$$

The Transform OfPoint method implements the previous function.

The Geometry.Transform class properties and methods are identified in the following sections.

Identity

Transform the Identity.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

CreateReflection()

Reflect a specified plane.

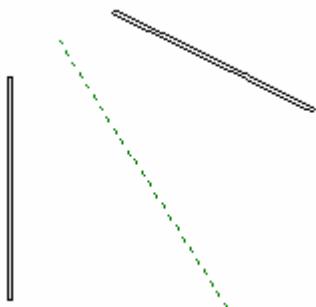


Figure 112: Wall Reflection relationship

As the previous picture shows, one wall is mirrored by a reference plane. The CreateReflection() method 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.CreateReflection(refPlane.GetPlane());

    return mirTrans;
}
```

CreateRotation() and CreateRotationAtPoint()

Rotate by a specified angle around a specified axis at (0,0,0) or at a specified point.

CreateTranslation()

Translate by a specified vector. Given a vector XYZ data, a transformation is created as follow:

$$(x \quad y \quad z) \rightarrow \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \end{pmatrix}$$

Determinant

Transformation determinant.

$$\begin{vmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} \end{vmatrix}$$

HasReflection

This is a Boolean value that indicates whether the transformation produces a reflection.

Scale

A value that represents the transformation scale.

Inverse

An inverse transformation. Transformation matrix A is invertible if a transformation matrix B exists such that $AB = BA = I$ (identity).

IsIdentity

Boolean value that indicates whether this transformation is an identity.

IsTranslation

Boolean value that indicates whether this transformation is a translation.

Geometry.Transform provides methods to perform basal matrix operations.

Multiply

Multiplies a transformation by a specified transformation and returns the result.

Operator* - Multiplies two specified transforms.

ScaleBasis

Scales the basis vectors and returns the result.

ScaleBasisAndOrigin

Scales the basis vectors and the transformation origin returns the result.

OfPoint

Applies the transformation to the point. The Origin property is used.

OfVector

Applies the transform to the vector. The Origin property is not used.

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.

Table 48: Transformed Methods

Class Name	Function Description
Curve.get_Transformed(Transform transform)	Applies the specified transformation to a curve
GeometryElement.GetTransformed(Transform transform)	Transforms a copy of the geometry in the original element.

`Profile.get_Transformed(Transform transform)` Transforms the profile and returns the result.

`Mesh.get_Transformed(Transform transform)` Transforms the mesh and returns the result.

Note: The transformed method clones itself then returns the transformed cloned result.

In addition to these methods, the `Instance` class (which is the parent class for elements like family instances, link instances, and imported CAD content) has two methods which provide the transform for a given `Instance`. The `GetTransform()` method obtains the basic transform for the instance based on how the instance is placed, while `GetTotalTransform()` provides the transform modified with the true north transform, for instances like import instances.

Reference

The Reference 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.GetEndPointReference` method
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.

The `Reference.ConvertToStableRepresentation()` method can be used to save a reference to a geometry object, for example a face, edge, or curve, as a string, and later in the same Revit session (or even in a different session where the same document is present) use `ParseFromStableRepresentation()` method to obtain an identical `Reference` using the string as input.

Options

Geometry is typically extracted from the indexed property `Element.Geometry`. The original geometry of a beam, column or brace, before the instance is modified by joins, cuts, coping, extensions, or other post-processing, can be extracted using the `FamilyInstance.GetOriginalGeometry()` method. Both `Element.Geometry` and

`FamilyInstance.GetOriginalGeometry()` accept 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.

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. This option cannot be set to true when used with `FamilyInstance.GetOriginalGeometry()`.

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, and it should be extracted.

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).

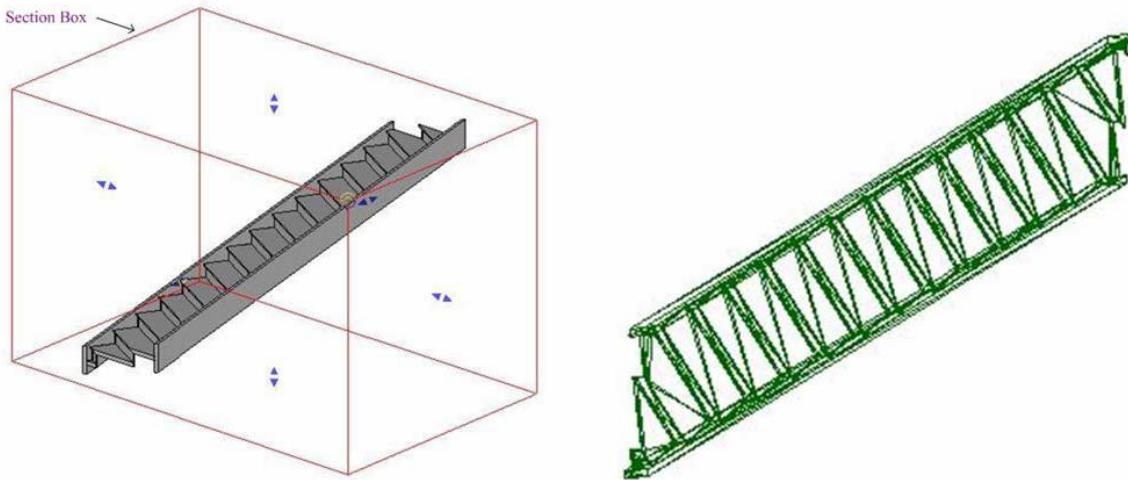


Figure 113: Different section boxes display different geometry

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.

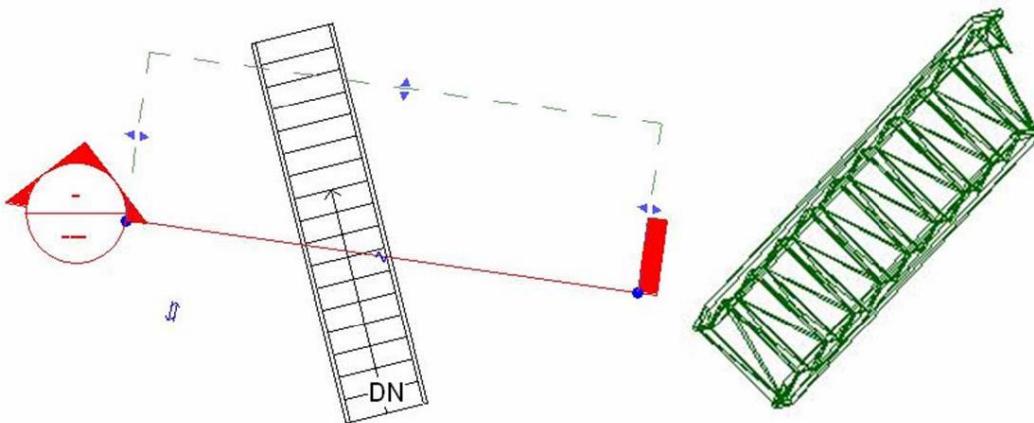


Figure 114: Retrieve Geometry section view

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 115: 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:

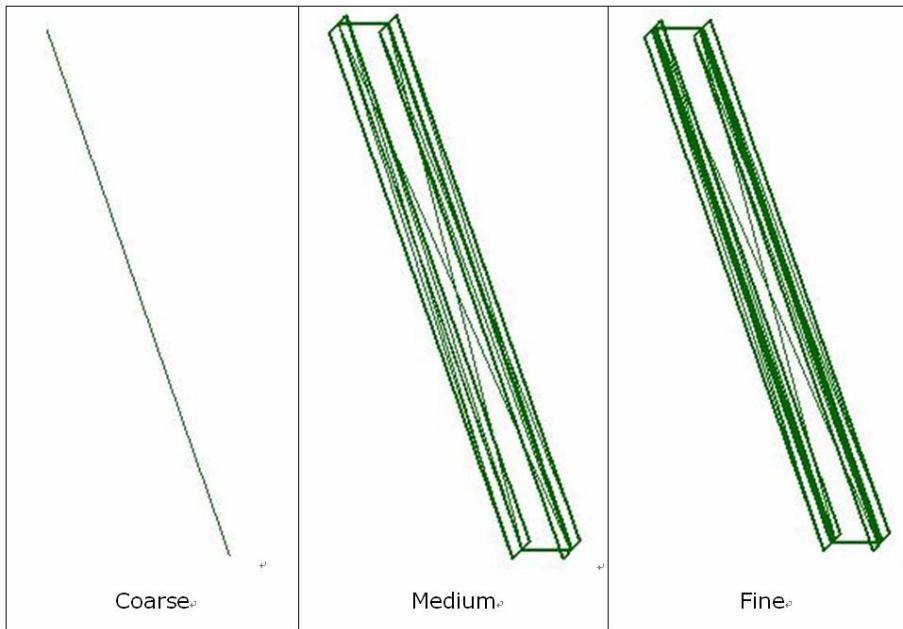


Figure 116: Detail geometry for a beam

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`.

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.

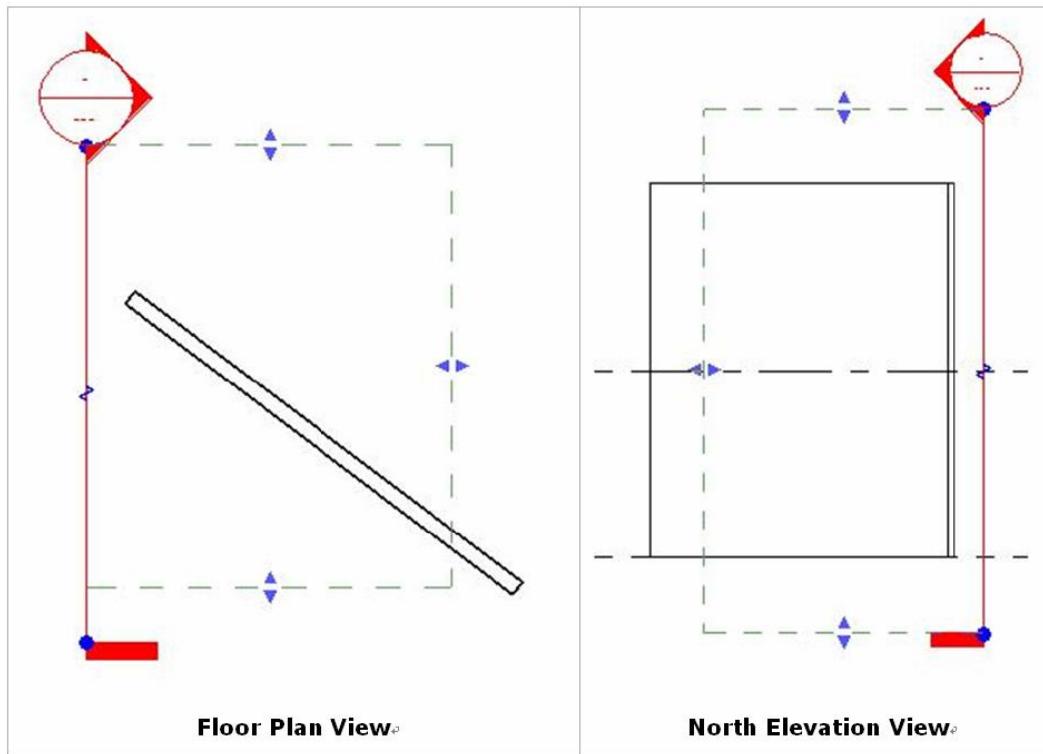


Figure 117: BoundingBoxXYZ in section view

The dash lines in the previous pictures show the section view boundary exposed as the CropBox property (a BoundingBoxXYZ instance).

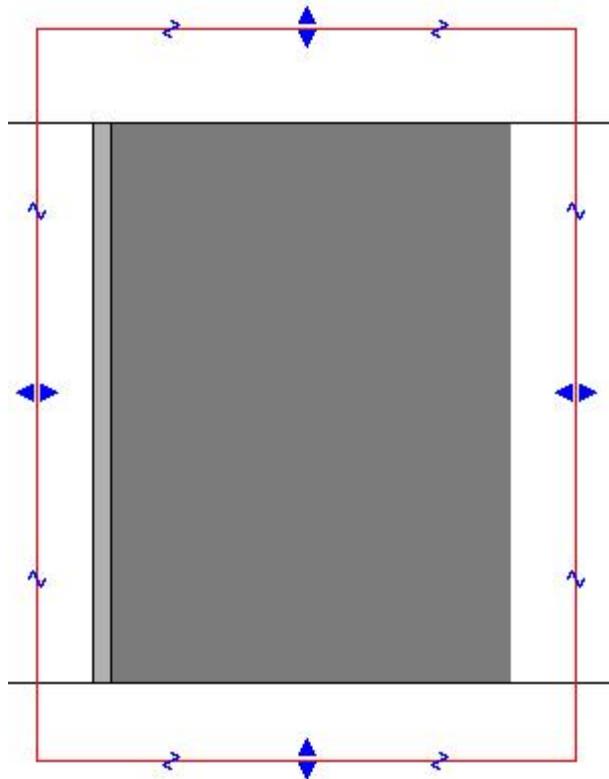


Figure 118: Created section view

The previous picture displays the corresponding section view. The wall outside the view boundary is not displayed.

Define a Section Box

BoundingBoxXYZ is also used to define a section box for a 3D view retrieved from the View3D.GetSectionBox() method. Select the Section Box property in the Properties Dialog box. The section box is shown as follows:

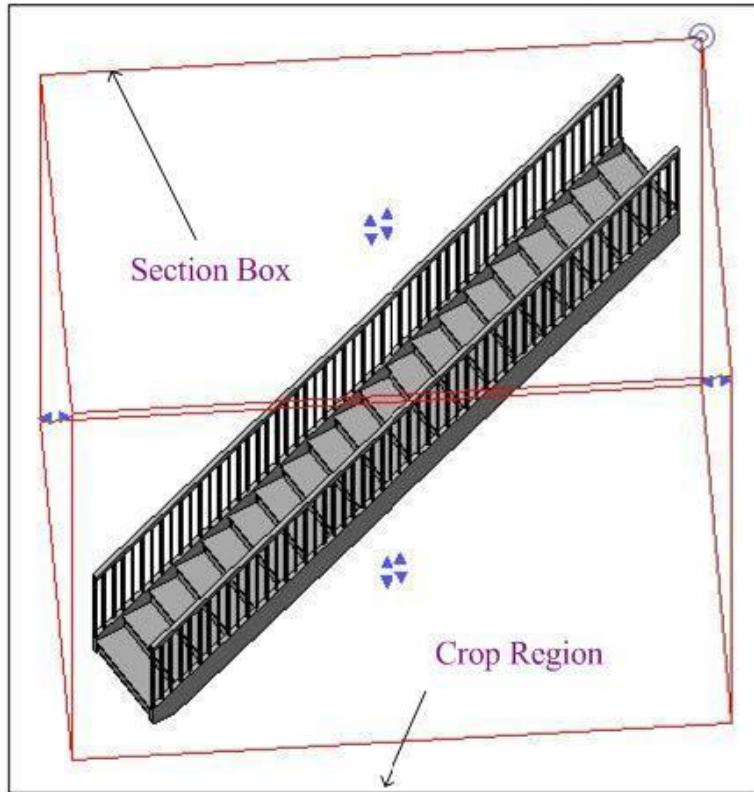


Figure 119: 3D view section box

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 `ViewSection.CreateDetail ()` method .

The following table identifies the main uses for this class.

Table 49: BoundingBoxXYZ properties

Property Name	Usage
Max/Min	Maximum/Minimum coordinates. These two properties define a 3D box parallel to any coordinate axis. The <code>Transform</code> 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.
MaxEnabled/ MinEnabled	<p>Defines whether the maximum/minimum bound is active for a given dimension.</p> <ul style="list-style-type: none"> If the Enable property is false, these two properties should also return false. <pre><table cellpadding="4" cellspacing="0" summary="" id="GUID-7845211-E211-4E27-B080-BE9FA3C50631__TABLE_4B666584884A4DFCA168F8AACD62FAA6" class="table" frame="border" border="1" rules="all"> <tbody class="tbody"> <tr class="row"> <td class="entry" valign="top"></td> <td class="entry" valign="top">If the crop view is turned on, both <i>MaxEnabled</i> property and <i>MinEnabled</i> property return true.</td> </td> </tr> <tr class="row"> <td class="entry" valign="top"></td> <td class="entry" valign="top">If the crop view is turned off, both <i>MaxEnabled</i> property and <i>MinEnabled</i> property return false.</td> </td> </tr> </tbody> </table></pre>
	<ul style="list-style-type: none"> 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.GetSectionBox() method, 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.

Bounds Wrapper for the Max/Min properties.

BoundEnabled Wrapper for the MaxEnabled/MinEnabled 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)
{
    if (!view3d.IsSectionBoxActive)
    {
        TaskDialog.Show("Revit", "The section box for View3D isn't active.");
        return;
    }

    BoundingBoxXYZ box = view3d.GetSectionBox();

    // Create a rotation transform to apply to the section box
    XYZ origin = new XYZ(0, 0, 0);
    XYZ axis = new XYZ(0, 0, 1);

    // Rotate 30 degrees
    Transform rotate = Transform.CreateRotationAtPoint(axis, Math.PI/6.0, origin);

    // Transform the View3D's section box with the rotation transform
```

```
    box.Transform = box.Transform.Multiply(rotate);

    // Set the section box back to the view (requires an open transaction)
    view3d.SetSectionBox(box);

}
```

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.

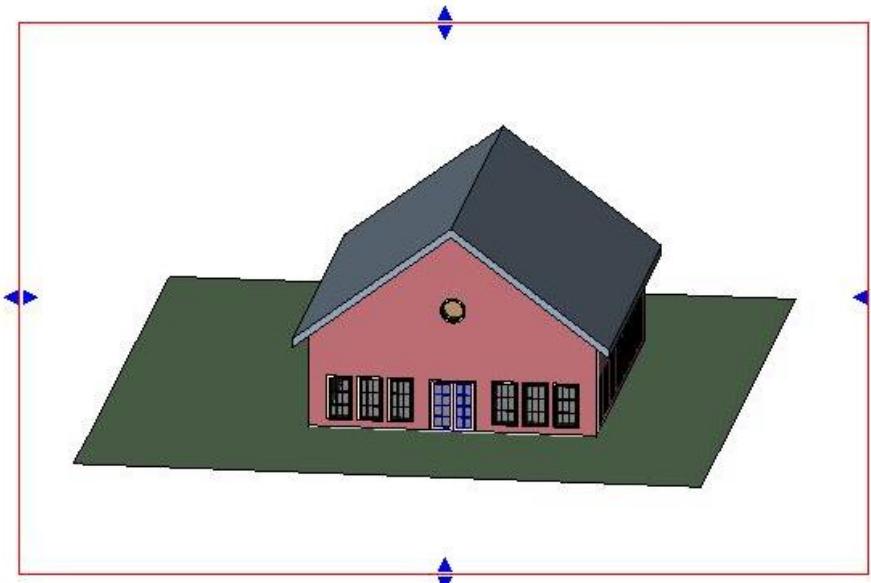


Figure 120: View outline

Two points define a BoundingBoxUV.

- Min point - The bottom-left endpoint.
- Max point - The upper-right endpoint.

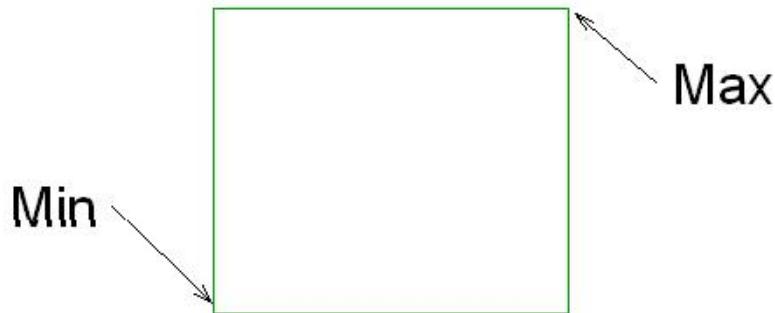


Figure 121: BoundingBoxUV Max and Min

Note: BoundingBoxUV cannot present a gradient rectangle as the following picture shows.

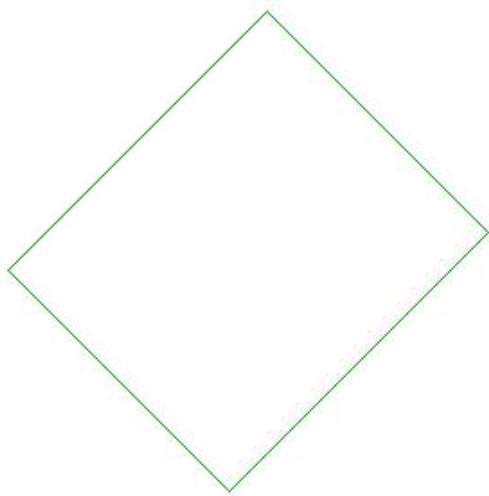


Figure 122: Gradient rectangle

3.10.4 Collection Classes

Specialized geometry collection classes in the Revit API.

The API provides the following collection classes based on the items they contain:

Table 50: Geometry Collection Classes

Class/Type	Corresponding Collection Classes	Corresponding Iterators
Edge	EdgeArray, EdgeArrayArray	EdgeArrayIterator, EdgeArrayArrayIterator

Face	FaceArray	FaceArrayIterator
Reference	ReferenceArray	ReferenceArrayIterator
Double value	DoubleArray	DoubleArrayIterator

All of these classes use very similar methods and properties to do similar work. For more details, refer to [Collections](#).

3.10.5 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. The `GeometryElement` may contain the desired geometry as a `Solid` or `GeometryInstance` depending on whether a beam is joined or standalone, and this code covers both cases.

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();
    System.Collections.Generic.List<Autodesk.Revit.DB.Solid> solids = new System.Collections.Generic.List<Autodesk.Revit.DB.Solid>();

    //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 System.Collections.Generic.List<Autodesk.Revit.DB.Solid> solids)
{
    foreach (Autodesk.Revit.DB.GeometryObject geomObj in geomElem)
    {
        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.Add(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);

    }

}

```

The above example uses the FamilyInstance.Geometry property to access the true geometry of the beam. To obtain the original geometry of a family instance before it is modified by joins, cuts, coping, extensions, or other post-processing, use the FamilyInstance.GetOriginalGeometry() method.

Note: For more information about how to retrieve the Geometry.Options type object, refer to [Geometry Helper Classes](#).

3.10.6 Extrusion Analysis of a Solid

The utility class ExtrusionAnalyzer allows you to attempt to “fit” a given piece of geometry into the shape of an extruded profile. An instance of this class is a single-time use class which should be supplied a solid geometry, a plane, and a direction. After the ExtrusionAnalyzer has been initialized, you can access the results through the following members:

- The GetExtrusionBase() method returns the calculated base profile of the extruded solid aligned to the input plane.
- The CalculateFaceAlignment() method can be used to identify all faces from the original geometry which do and do not align with the faces of the calculated extrusion. This could be useful to figure out if a wall, for example, has a slanted join at the top as would be the case if there is a join with a roof. If a face is unaligned, something is joined to the geometry that is affecting it.
- To determine the element that produced the non-aligned face, pass the face to Element.GetGeneratingElementIds(). For more details on this utility, see the following section.

The ExtrusionAnalyzer utility works best for geometry which are at least somewhat “extrusion-like”, for example, the geometry of a wall which may or may not be affected by end joins, floor joins, roof joins, openings cut by windows and doors, or other modifications. Rarely for specific shape and directional combinations the analyzer may be unable to determine a coherent face to act as the base of the extrusion – an InvalidOperationException will be thrown in these situations.

In this example, the extrusion analyzer is used to calculate and outline a shadow formed from the input solid and the sun direction.

Code Region: Use Extrusion Analyzer to calculate and draw a shadow outline.

```
/// <summary>
/// Draw the shadow of the indicated solid with the sun direction specified.
/// </summary>
/// <remarks>The shadow will be outlined with model curves added to the document.
/// A transaction must be open in the document.</remarks>
/// <param name="document">The document.</param>
/// <param name="solid">The target solid.</param>
/// <param name="targetLevel">The target level where to measure and draw the shadow.</param>
/// <param name="sunDirection">The direction from the sun (or light source).</param>
/// <returns>The curves created for the shadow.</returns>
/// <throws cref="Autodesk.Revit.Exceptions.InvalidOperationException">Thrown by ExtrusionAnalyzer when the geometry and direction combined do not permit a successful analysis.</throws>
private static ICollection<ElementId> DrawShadow(Document document, Solid solid, Level targetLevel, XYZ sunDirection)
{
    // Create target plane from level.
    Plane plane = Plane.CreateByNormalAndOrigin(XYZ.BasisZ, new XYZ(0, 0, targetLevel.ProjectElevation));
    // Create extrusion analyzer.
```

```
ExtrusionAnalyzer analyzer = ExtrusionAnalyzer.Create(solid,
plane, sunDirection);

// Get the resulting face at the base of the calculated extrusion.

Face result = analyzer.GetExtrusionBase();

// Convert edges of the face to curves.

CurveArray curves = document.Application.Create.NewCurveArray();

foreach (EdgeArray edgeLoop in result.EdgeLoops)

{

    foreach (Edge edge in edgeLoop)

    {

        curves.Append(edge.AsCurve());

    }

}

// Get the model curve factory object.

Autodesk.Revit.Creation.ItemFactoryBase itemFactory;

if (document.IsFamilyDocument)

    itemFactory = document.FamilyCreate;

else

    itemFactory = document.Create;
```

```
// Add a sketch plane for the curves.

CurveLoop loop = new CurveLoop();

foreach (Curve currentCurve in curves)

{

    loop.Append(currentCurve);

}

SketchPlane sketchPlane = SketchPlane.Create(document, loop.
GetPlane());

document.Regenerate();

// Add the shadow curves

ModelCurveArray curveElements = itemFactory.NewModelCurveArr
ay(curves, sketchPlane);

// Return the ids of the curves created

List<ElementId> curveElementIds = new List<ElementId>();

foreach (ModelCurve curveElement in curveElements)

{

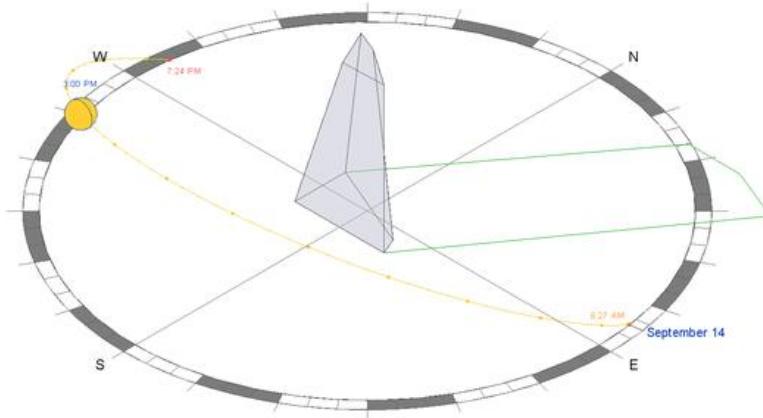
    curveElementIds.Add(curveElement.Id);

}

return curveElementIds;

}
```

The utility above can be used to compute the shadow of a given mass with respect to the current sun and shadows settings for the view:



3.10.7 Finding geometry by ray projection

The `ReferenceIntersector` class can be used to find elements that are intersected by a given ray.

ReferenceIntersector

This class allows an application to use Revit's picking tools to find elements and geometry. This class uses a ray from a point in a specified direction to find the geometry that is hit by the ray.

The class intersects 3D geometry only and requires a 3D view on creation. It is possible to use a 3D view which has been cut by a section box, or which has view-specific geometry and graphics options set. The visibility settings on the input view will determine if a particular element is returned (for example, hidden elements will never be returned by this tool, nor will elements whose geometry is outside the section box of the view).

The `ReferenceIntersector` class supports filtering the output based on element or reference type. The output can be customized based on which constructor is used or by using methods and properties of the class prior to calling a method to perform the ray projection.

There are 4 constructors.

Name	Description
<code>ReferenceIntersector(View3D)</code>	Constructs a <code>ReferenceIntersector</code> which is set to return intersections from all elements and representing all reference target types.

```
ReferenceIntersector(ElementFilter,
FindReferenceTarget, View3D)
```

Constructs a ReferenceIntersector which is set to return intersections from any element which passes the filter.

```
ReferenceIntersector(ElementId,
FindReferenceTarget, View3D)
```

Constructs a ReferenceIntersector which is set to return intersections from a single target element only.

```
ReferenceIntersector(ICollection<ElementId>,
FindReferenceTarget, View3D)
```

Constructs a ReferenceIntersector which is set to return intersections from any of a set of target elements.

The FindReferenceTarget enumeration includes the options: Element, Mesh, Edge, Curve, Face or All.

Finding elements

There are two methods to project a ray, both of which take as input the origin of the ray and its direction. Only references for elements that are in front of the ray will be returned. The Find() method returns a collection of ReferenceWithContext objects which match the ReferenceIntersector's criteria. This object contains the intersected reference, which can be both the elements and geometric references which intersect the ray. Some element references returned will have a corresponding geometric object which is also intersected (for example, rays passing through openings in walls will intersect the wall and the opening element). If interested only in true physical intersections an application should discard all references whose Reference is of type Element.

The FindNearest() method behaves similarly to the Find() method, but only returns the intersected reference nearest to the ray origin.

The ReferenceWithContext returned includes a proximity parameter. This is the distance between the origin of the ray and the intersection point. An application can use this distance to exclude items too far from the origin for a particular geometric analysis. An application can also use this distance to take on some interesting problems involving analyzing the in place geometry of the model.

Note: These methods will not return intersections with elements which are not in the active design option.

Elements in linked files

The FindReferencesInRevitLinks property offers an option to return element results encountered in Revit Links. If set to false, no Reference to any Element from a Revit Link will be found by the ReferenceIntersector, and all References returned will be to an element in the host document only. If set to true, the results may include both References to Elements in hosts and References to Elements from a link instance.

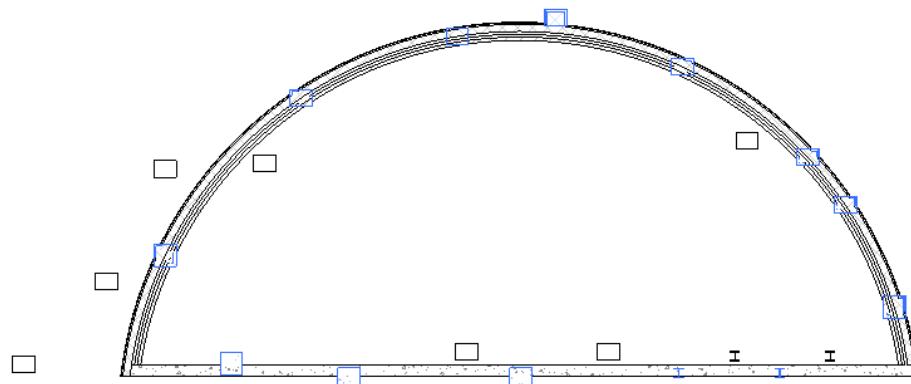
If a list of target ElementIds is set in the ReferenceIntersector, references will be returned only if the ElementId matches the id of the intersected RevitLinkInstance. If there is a match, any intersecting elements in the link will be returned (their ids will not be compared with the target ids list).

If there is an ElementFilter applied, the elements in the link will be evaluated against the stored ElementFilter. Note that results may not be as expected if the filter applied is geometric (such as a BoundingBox filter or ElementIntersects filter). This is because the filter will be evaluated for linked elements in the coordinates of the linked model, which may not match the coordinates of the elements as they appear in the host model. Also, ElementFilters that accept a Document and/or ElementId as input during their instantiation will not correctly pass elements that appear in the link, because the filter will not be able to match link elements to the filter's criteria.

Find elements near elements

One major use for this tool is to find elements in close proximity to other elements. This allows an application to use the tool as its "eyes" and determine relationships between elements which don't have a built-in relationship already.

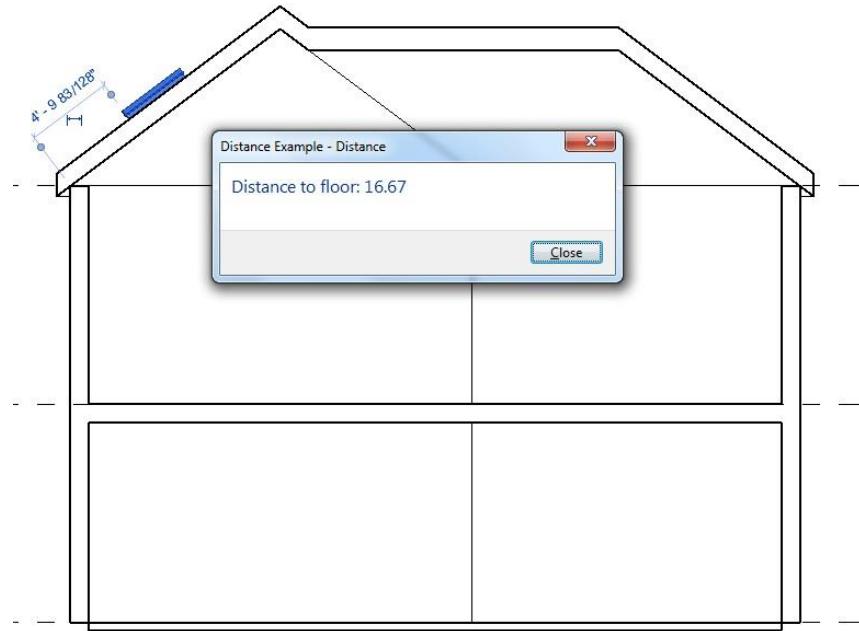
For example, the ray-tracing capability can be used to find columns embedded in walls. As columns and walls don't maintain a relationship directly, this class allows us to find potential candidates by tracing rays just outside the extents of the wall, and looking for intersections with columns.



Example: Find columns embedded in walls

Measure distances

This class could also be used to measure the vertical distance from a skylight to the nearest floor.



Example: measure with ReferenceIntersector.FindNearest()

Code Region: Measuring Distance using Ray Projection

```
public class RayProjection : IExternalCommand
{
    public Result Execute(ExternalCommandData revit, ref string message, ElementSet elements)
    {
        Document doc = revit.Application.ActiveUIDocument.Document;

        ICollection<ElementId> selectedIds = revit.Application.ActiveUIDocument.Selection.GetElementIds();

        // If skylight is selected, process it.
        FamilyInstance skylight = null;
        if (selectedIds.Count == 1)
```

```
{  
    foreach (ElementId id in selectedIds)  
    {  
        Element e = doc.GetElement(id);  
        if (e is FamilyInstance)  
        {  
            FamilyInstance instance = e as FamilyInstance;  
            bool isWindow = (instance.Category.Id.Value == (int)Built  
InCategory.OST_Windows);  
            bool isHostedByRoof = (instance.Host.Category.Id.Value ==  
(int)BuiltInCategory.OST_Roofs);  
  
            if (isWindow && isHostedByRoof)  
            {  
                skylight = instance;  
            }  
        }  
    }  
  
    if (skylight == null)  
    {  
        message = "Please select one skylight.";  
        return Result.Cancelled;  
    }  
  
    // Calculate the height
```

```
Line line = CalculateLineAboveFloor(doc, skylight);

// Create a model curve to show the distance

Plane plane = Plane.CreateByNormalAndOrigin(new XYZ(1, 0, 0), line.GetEndPoint(0));

SketchPlane sketchPlane = SketchPlane.Create(doc, plane);

ModelCurve curve = doc.Create.NewModelCurve(line, sketchPlane);

// Show a message with the length value

TaskDialog.Show("Distance", "Distance to floor: " + String.Format(
{0:f2}", line.Length));

return Result.Succeeded;

}

/// <summary>

/// Determines the line segment that connects the skylight to the nearest
floor.

/// </summary>

/// <returns>The line segment.</returns>

private Line CalculateLineAboveFloor(Document doc, FamilyInstance skyligh
t)

{

    // Find a 3D view to use for the ReferenceIntersector constructor

    FilteredElementCollector collector = new FilteredElementCollector(do
c);

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

```
    View3D view3D = collector.OfClass(typeof(View3D)).Cast<View3D>().First<View3D>(isNotTemplate);

    // Use the center of the skylight bounding box as the start point.
    BoundingBoxXYZ box = skylight.get_BoundingBox(view3D);
    XYZ center = box.Min.Add(box.Max).Multiply(0.5);

    // Project in the negative Z direction down to the floor.
    XYZ rayDirection = new XYZ(0, 0, -1);

    ElementClassFilter filter = new ElementClassFilter(typeof(Floor));

    ReferenceIntersector refIntersector = new ReferenceIntersector(filter,
        FindReferenceTarget.Face, view3D);

    ReferenceWithContext referenceWithContext = refIntersector.FindNearest(
        center, rayDirection);

    Reference reference = referenceWithContext.GetReference();

    XYZ intersection = reference.GlobalPoint;

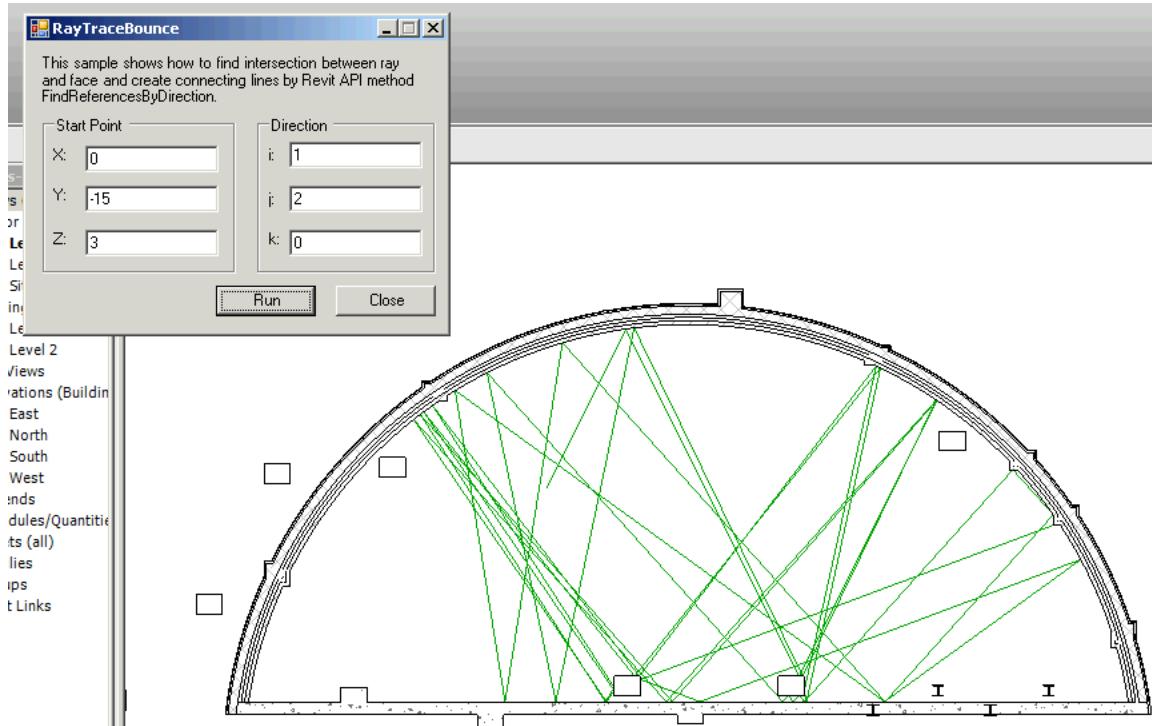
    // Create line segment from the start point and intersection point.
    Line result = Line.CreateBound(center, intersection);

    return result;
}
```

Ray bouncing/analysis

The references returned by `ReferenceIntersector.Find()` include the intersection point on the geometry. Knowing the intersection point on the face, the face's material, and the ray direction allows an application to analyze reflection and refraction within the building. The following image

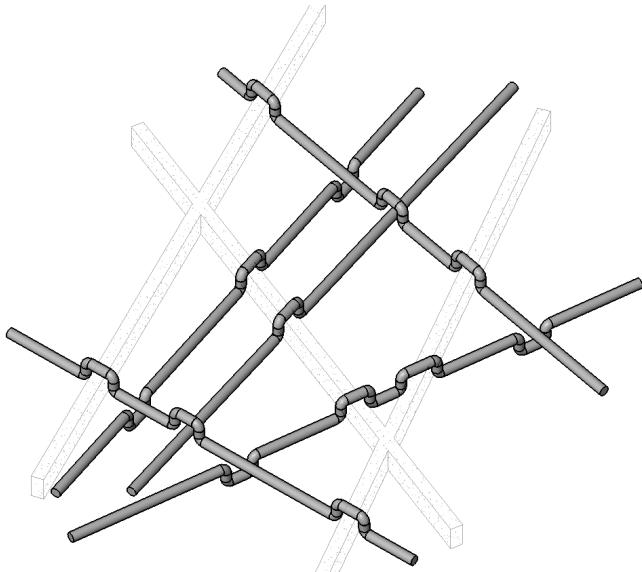
demonstrates the use of the intersection point to reflect rays intersected by model elements; model curves were added to represent the path of each ray.



Example: Rays bouncing off intersected faces

Find intersections/collisions

Another use of the ReferenceIntersector class would be to detect intersections (such as beams or pipes) which intersect/interference with the centerline of a given beam or pipe.



Example: Reroute elements around interferences

3.10.8 Geometry Utility Classes

A number of utility classes are available for working with geometry objects.

HostObjectUtils

The HostObjectUtils class offers methods as a shortcut to locate certain faces of compound HostObjects. These utilities retrieve the faces which act as the boundaries of the object's CompoundStructure:

- HostObjectUtils.GetSideFaces() – applicable to Walls and FaceWalls; you can obtain either the exterior or interior finish faces.
- HostObjectUtils.GetTopFaces() and HostObjectUtils.GetBottomFaces() – applicable to roofs, floors, and ceilings.

SolidUtils

The SolidUtils class contains methods to perform operations on solids.

- SolidUtils.Clone() - creates a new Solid which is a copy of the input Solid
- SolidUtils.SplitVolumes() - takes a solid which includes disjoint enclosed volumes and returns newly created Solid objects representing each volume. If no splitting is necessary, the input solid is returned.
- SolidUtils.TessellateSolidOrShell() - triangulates a given input Solid (which can be one or more fully closed volumes, or an open shell). Returns a TriangulatedSolidOrShell object which allows access to the stored triangulated boundary component of a solid or a triangulated connected component of a shell.
- SolidUtils.CreateTransformed() - creates a new solid which is the transformation of the input solid.

JoinGeometryUtils

The JoinGeometryUtils class contains methods for joining and unjoining elements, and for managing the order in which elements are joined. These utilities are not available for family documents.

- JoinGeometryUtils.AreElementsJoined() - determines whether two elements are joined
- JoinGeometryUtils.GetJoinedElements() - returns all elements joined to given element
- JoinGeometryUtils.JoinGeometry() - creates a join between two elements that share a common face. The visible edge between the joined elements is removed, and the joined elements then share the same line weight and fill pattern.
- JoinGeometryUtils.UnjoinGeometry() - removes a join between two joined elements
- JoinGeometryUtils.SwitchJoinOrder() - reverses the order in which two elements are joined. The cutting element becomes the cut element and vice versa.

- `JoinGeometryUtils . IsCuttingElementInJoin()` - determines whether the first of two joined elements is cutting the second element or vice versa.

FacetingUtils

This class is used to convert a triangulated structure into a structure in which some of the triangles have been consolidated into quadrilaterals.

- `FacetingUtils.ConvertTrianglesToQuads()` - this method takes a `TriangulationInterface` (constructed from a `TriangulatedSolidOrShell`) as input and returns a collection of triangles and quadrilaterals representing the original triangulated object.

3.10.9 Room and Space Geometry

The Revit API provides access to the 3D geometry of spatial elements (rooms and spaces).

The `SpatialElementGeometryCalculator` class can be used to calculate the geometry of a spatial element and obtain the relationships between the geometry and the element's boundary elements. There are 2 options which can be provided to this utility:

- `SpatialElementBoundaryLocation` – whether to use finish faces or boundary element centerlines for calculation
- `StoredFreeBoundaryFaces` – whether to include faces which don't map directly to a boundary element in the results.

The results of calculating the geometry are contained in the class `SpatialElementGeometryResults`. From the `SpatialElementGeometryResults` class, you can obtain:

- The Solid volume representing the geometry (`GetGeometry()` method)
- The boundary face information (a collection `SpatialElementBoundarySubfaces`)

Each subface offers:

- The face of the spatial element
- The matching face of the boundary element
- The subface (the portion of the spatial element face bounded by this particular boundary element)
- The subface type (bottom, top, or side)

Some notes about the use of this utility:

- The calculator maintains an internal cache for geometry it has already processed. If you intend to calculate geometry for several elements in the same project you should use a single instance of this class. Note that the cache will be cleared when any change is made to the document.
- Floors are almost never included in as boundary elements. Revit uses the 2D outline of the room to form the bottom faces and does not match them to floor geometry.

- Openings created by wall-cutting features such as doors and windows are not included in the returned faces.
- The geometry calculations match the capabilities offered by Revit. In some cases where Revit makes assumptions about how to calculate the volumes of boundaries of rooms and spaces, these assumptions will be present in the output of the utility.

The following example calculates a room's geometry and finds its boundary faces

Code Region: Face Area using SpatialElementGeometryCalculator

```
public void SpaceArea(Document doc, Room room)

{
    SpatialElementGeometryCalculator calculator = new SpatialElementGeometryC
alculator(doc);

    // compute the room geometry

    SpatialElementGeometryResults results = calculator.CalculateSpatialElemen
tGeometry(room);

    // get the solid representing the room's geometry

    Solid roomSolid = results.GetGeometry();

    foreach (Face face in roomSolid.Faces)
    {
        double faceArea = face.Area;

        // get the sub-faces for the face of the room

        IList<SpatialElementBoundarySubface> subfaceList = results.GetBoundar
yFaceInfo(face);

        foreach (SpatialElementBoundarySubface subface in subfaceList)
        {
    }
```

```

        if (subfaceList.Count > 1) // there are multiple sub-faces that define the face

    {

        // get the area of each sub-face

        double subfaceArea = subface.GetSubface().Area;

        // sub-faces exist in situations such as when a room-bounding wall has been

        // horizontally split and the faces of each split wall combine to create the

        // entire face of the room

    }

}

}

}

```

The following example calculates a room's geometry and finds its the material of faces that belong to the elements that define the room.

Code Region: Face Material using SpatialElementGeometryCalculator

```

public void MaterialFromFace(Document doc)

{

    string s = "";

    UIDocument uidoc = new UIDocument(doc);

    Room room = doc.GetElement(uidoc.Selection.PickObject(ObjectType.Element).ElementId) as Room;

    SpatialElementBoundaryOptions spatialElementBoundaryOptions = new SpatialElementBoundaryOptions();

```

```

    spatialElementBoundaryOptions.SpatialElementBoundaryLocation = SpatialElementBoundaryLocation.Finish;

    SpatialElementGeometryCalculator calculator = new SpatialElementGeometryCalculator(doc, spatialElementBoundaryOptions);

    SpatialElementGeometryResults results = calculator.CalculateSpatialElementGeometry(room);

    Solid roomSolid = results.GetGeometry();

    foreach (Face roomSolidFace in roomSolid.Faces)

    {

        foreach (SpatialElementBoundarySubface subface in results.GetBoundaryFaceInfo(roomSolidFace))

        {

            Face boundingElementface = subface.GetBoundingElementFace();

            ElementId id = boundingElementface.MaterialElementId;

            s += doc.GetElement(id).Name + ", id = " + id.Value.ToString()
            () + "\n";

        }

    }

    TaskDialog.Show("revit",s);
}

```

3.11 Sketching

Sketches define the shape of many elements in Revit such as:

- Ceiling
- Extrusion
- Filled Region
- Floor
- Opening
- Stair

- Railing
- Roof

In addition to Sketch Elements, ModelCurve is also described in this chapter. For more details about Element Classification, see [Element Classification](#) in the [Elements Essentials](#) section.

Sketches of Ceilings, Floors, Walls, or Openings

Getting the Sketch

You cannot retrieve a Sketch object by iterating the Document with a FilteredElementCollector. Instead, use these properties to get the element id of the element's sketch.

- Ceiling.SketchId
- Floor.SketchId
- Opening.SketchId
- Wall.SketchId

For a given sketch, you can get element (Floor, Wall...) that owns the sketch with the `Sketch.OwnerId` property. `Sketch.GetAllElements()` returns all elements (ModelCurve, ReferencePlane, Dimension) that belong to a sketch.

Editing and Validating the Sketch

Use the `SketchEditScope` class to edit the sketch. While a Sketch editing session is active, you can add, delete or modify Sketch elements (curves, dimensions, reference planes). A transaction will be needed to make the changes. When you finish the session, the Sketch-based element will be updated.

Key methods include:

- SketchEditScope constructor - Creates a new SketchEditScope
- Start() - Starts editing a particular sketch. After this is started only elements owned by the Sketch and new elements to be added to the Sketch may be modified
- StartWithNewSketch() - Because a valid sketch may not initially exist for some Revit elements (such as Walls or Analytical Elements), a valid sketch will need to be created before it can be edited.
- Commit() - Commits the changes
- IsSketchEditingSupported() - Checks if a sketch can be edited with a SketchEditScope

`ElementTransformUtils.CopyElements(View, ICollection<ElementId>, View, Transform, CopyPasteOptions)` can be used to copy sketch members from a sketch to the main document.

Additional supported copy/paste cases:

- Copying within one Sketch - If there is an active sketch edit mode, you can now copy sketch members of the active sketch. The copied elements will be added to the active sketch.

- Copying between Sketches - Allows you to copy sketch members from one sketch to another. To do this, sketches must be parallel and the destination sketch must be in edit mode.
- Copying ModelCurves from the Document to a Sketch - Allows you to copy ModelCurves from the document to a sketch, if the sketch is in edit mode. To do this, the sketch plane must be parallel to the WorkPlane that ModelCurves are based on.

Boundary Validation

The `BoundaryValidation` class provides methods to validate curve loops for sketching:

- `IsValidHorizontalBoundary` identifies whether the provided curve loops create a valid horizontal boundary
- `IsValidBoundaryOnView` checks that a curve loop boundary is valid on the view's sketch plane.
- `IsValidBoundaryOnSketchPlane` checks that a curve loop boundary is valid on a sketch plane

Code Region: Edit a Floor Sketch

```
private void EditSketch(Floor floor)
{
    // Delete all lines in the sketch & create two new arcs that form a circle
    Document doc = floor.Document;
    Sketch sketch = doc.GetElement(floor.SketchId) as Sketch;
    using (SketchEditScope edit = new SketchEditScope(doc, "Edit Floor"))
    {
        edit.Start(floor.SketchId);
        using (Transaction t = new Transaction(doc, "Edit Sketch"))
        {
            t.Start();

            doc.Delete(sketch.GetAllElements());

            doc.Create.NewModelCurve(
                Arc.Create(sketch.SketchPlane.GetPlane(), 5, 0, Math.PI * 2),
                sketch.SketchPlane);
        }
    }
}
```

```

        t.Commit();

    }

    edit.Commit(new failuresPreprocessor());
}

}

private class failuresPreprocessor : IFailuresPreprocessor
{
    public FailureProcessingResult PreprocessFailures(FailuresAccessor failuresAccess
sor)
    {
        return FailureProcessingResult.Continue;
    }
}
}

```

3.11.1 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 accessing the Family's 3D modeling information, Sketch objects are important to forming the geometry. For more details, refer to [3D Sketch](#).

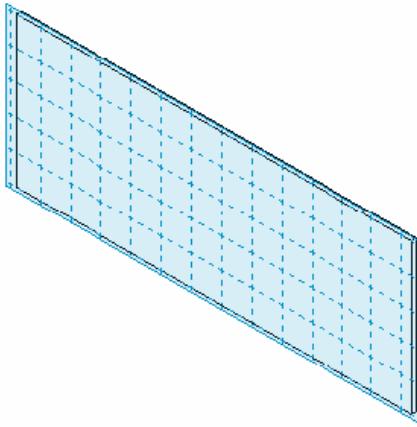
SketchPlane is the basis for all 2D sketch classes such as ModelCurve and Sketch. SketchPlane is also the basis for 2D Annotation Elements such as DetailCurve. Both ModelCurve and DetailCurve have the SketchPlane property and need a SketchPlane in the corresponding creation method. SketchPlane is always 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 77: 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 78: 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:

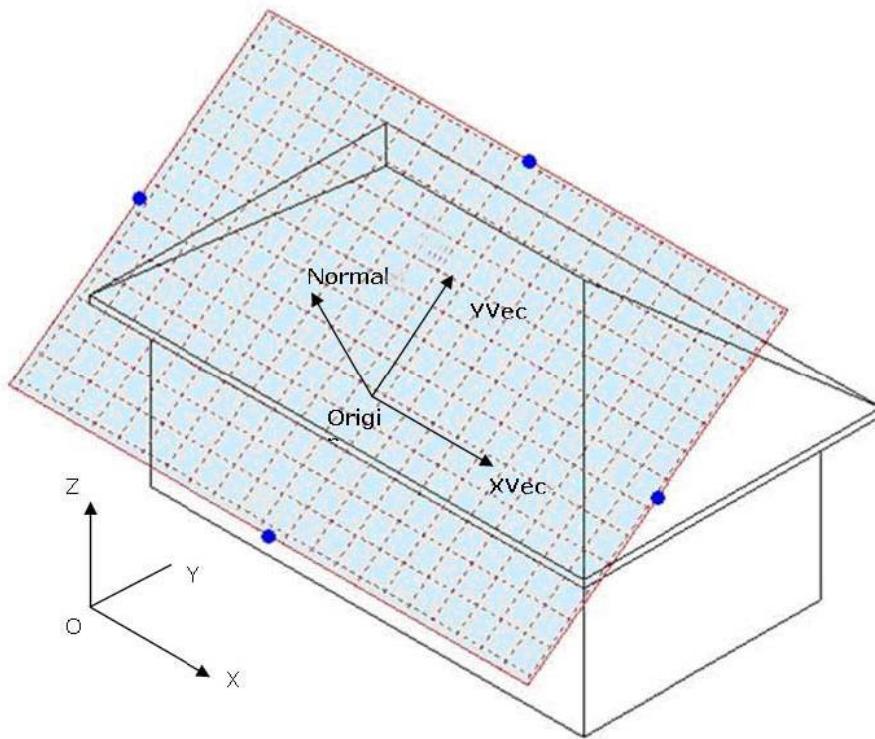


Figure 79: SketchPlane and Plane coordinate system

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 = Plane.CreateByNormalAndOrigin(newNormal, newOrigin);

    // create sketch plane

    SketchPlane sketchPlane = SketchPlane.Create(application.ActiveUIDocument.Document, geometryPlane);

    return sketchPlane;
}

```

3.11.2 3D 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 Class](#).

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:

Table 40: Extrusion 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.

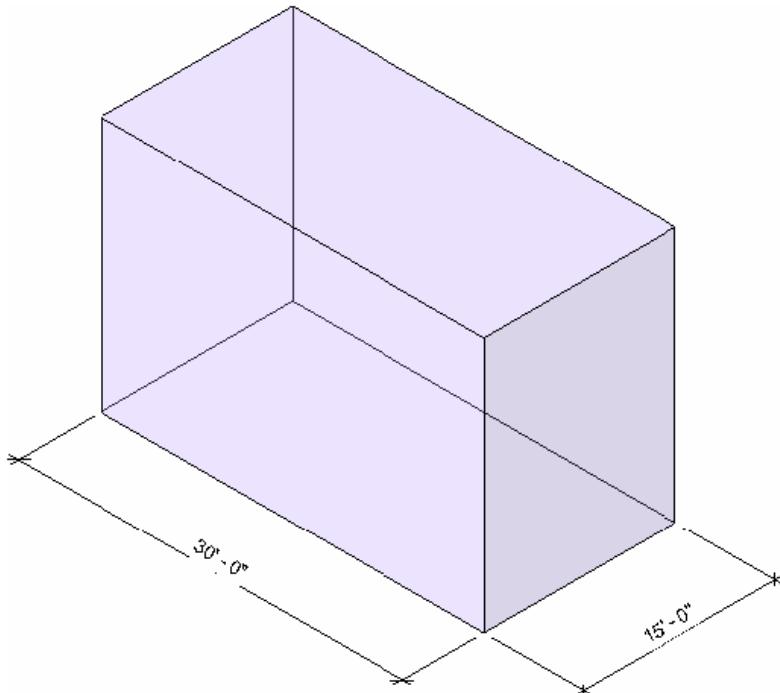


Figure 81: Extrusion result

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:

Table 41: Revolution 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.

EndAngle is consistent with the same parameter in the Revit UI. The following pictures illustrate the Revolution corresponding parameter, the sketch, and the result.

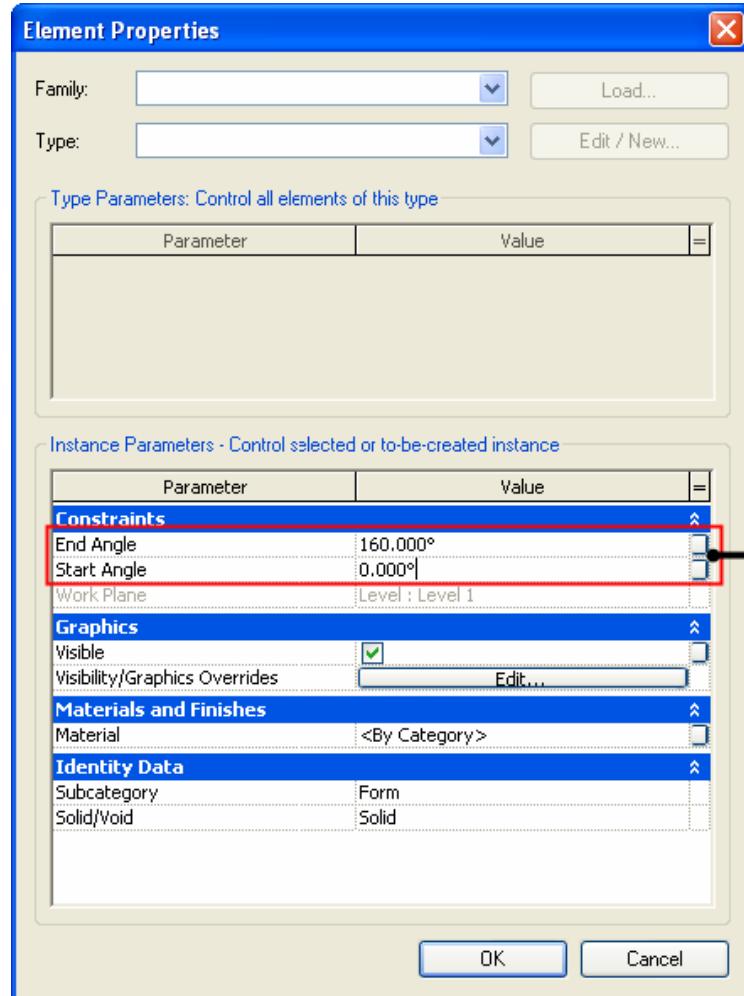


Figure 82: Corresponding parameter

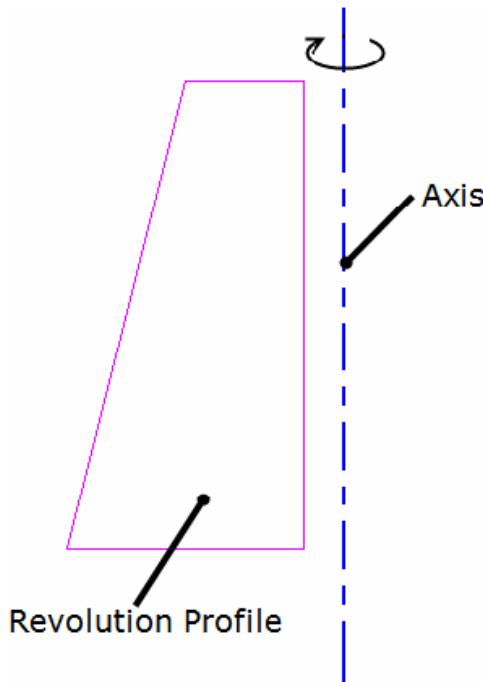


Figure 83: Revolution sketch

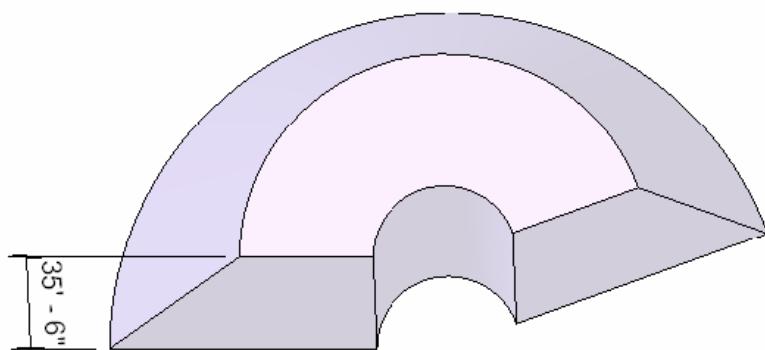


Figure 84: Revolution result

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

Blend

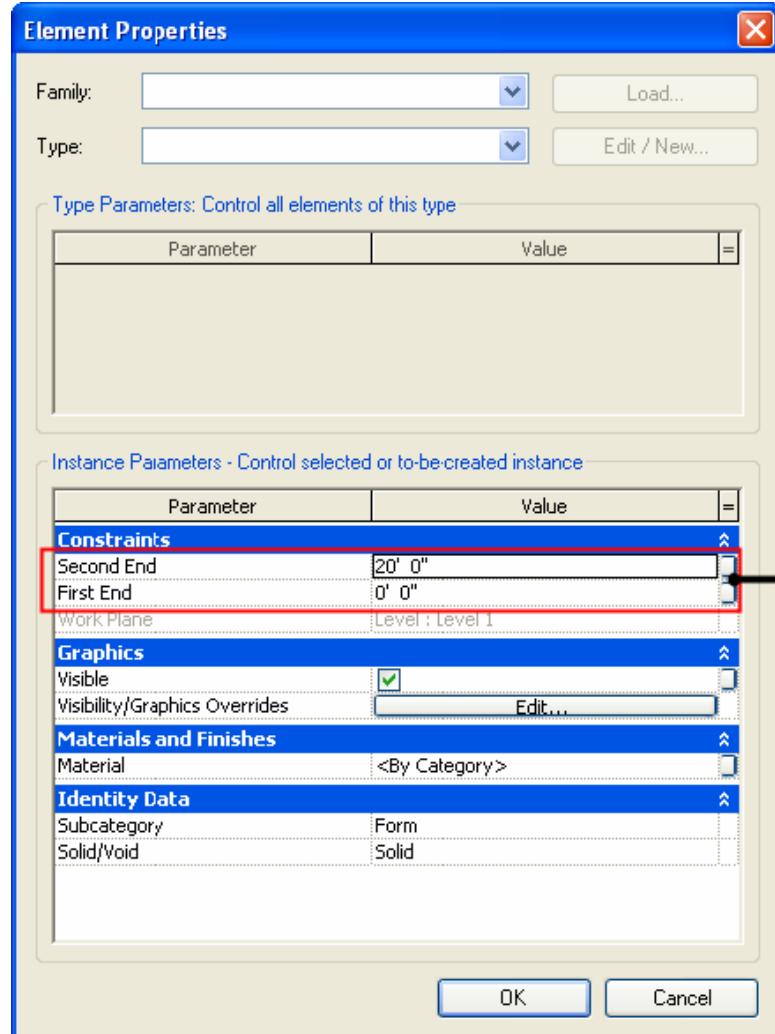
The Blend command blends two profiles together. For example, if you sketch a large rectangle and a smaller rectangle on top of it, Revit blends the two shapes together.

Query the Blend Form object for a generic form to use in family modeling and massing. The Blend class has the following properties:

Table 42: Blend 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.

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.



The value of FirstEnd
property is 0'0".
The value of
SecondEnd is 20'0".

Figure 85: Blend parameters in the UI

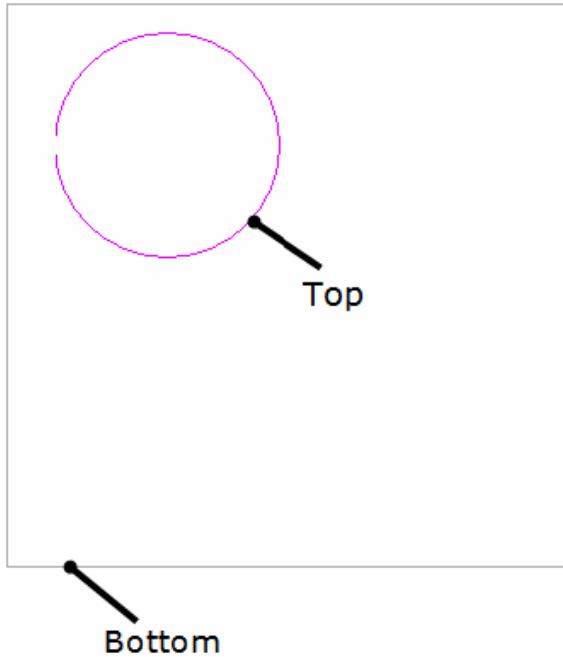


Figure 86: Blend top sketch and bottom sketch

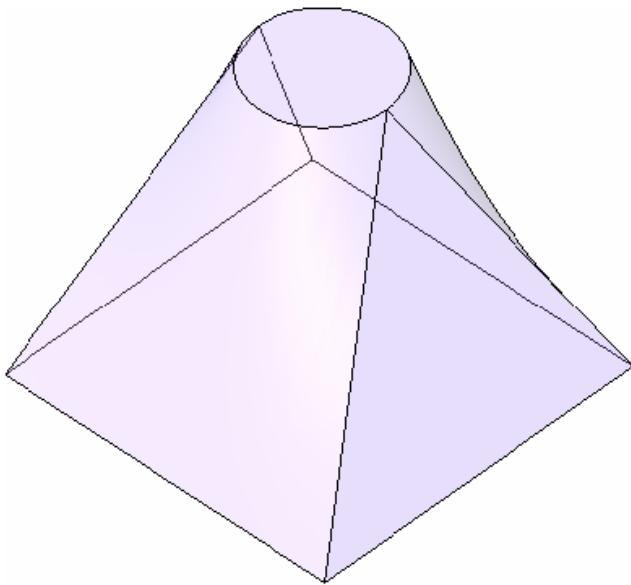


Figure 87: Blend result

Sweep

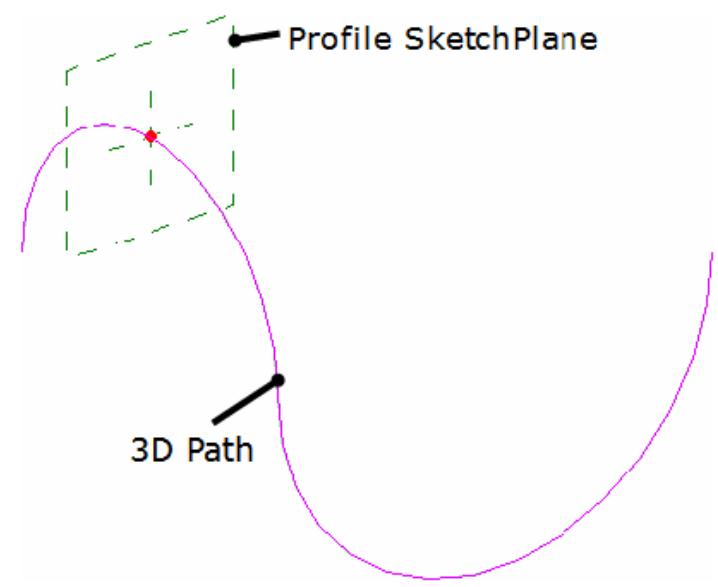
The Sweep command sweeps one profile along a created 2D path or selected 3D path. The path may be an open or closed loop, but must pierce the profile plane.

Query the Sweep Form object for a generic form for use in family modeling and massing. The Sweep class has the following properties:

Table 43: Sweep 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.

Creating a 2D Path is similar to other forms. The 3D Path is fetched by picking the created 3D curves.

**Figure 88: 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.

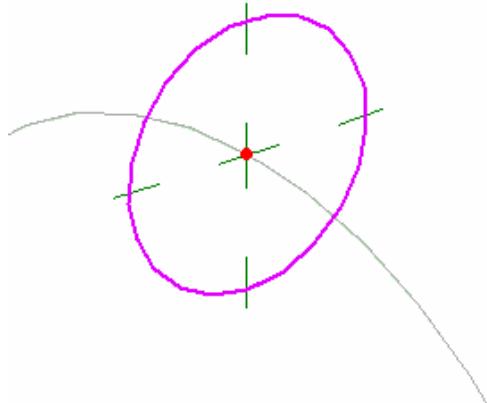
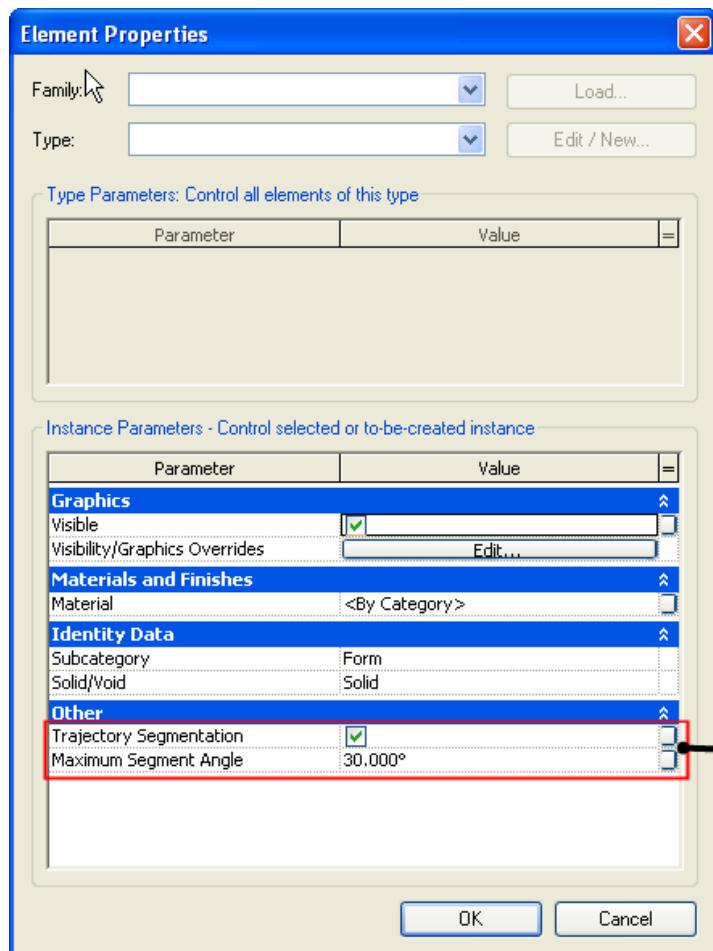


Figure 89: Sweep profile sketch

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.



The value of
EnableTrajSegmentation
property is True.
The value of
MaxSegmentAngle
property is 30.

Figure 90: 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°.

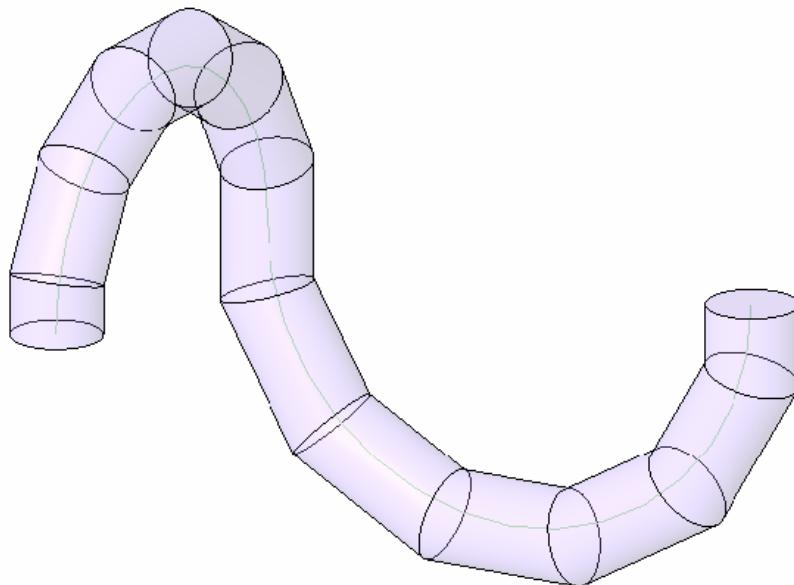


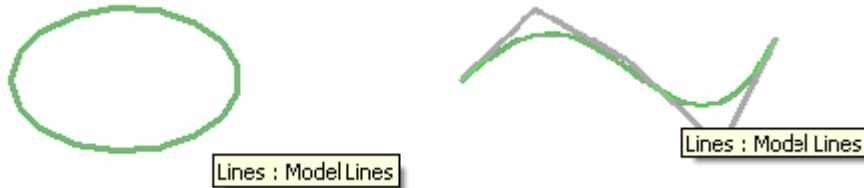
Figure 91: Sweep result

3.11.3 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 92:ModelLine and ModelArc**Figure 93: ModelEllipse and ModelNurbSpline**

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

```
public void CreateModelCurves(Document document)
{
    // 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 = Line.CreateBound(startPoint, endPoint);

    // 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 = Arc.Create(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 = Plane.CreateByNormalAndOrigin(normal, origin);

// Create a sketch plane in current document

SketchPlane sketch = SketchPlane.Create(document, geomPlane);

// Create a ModelLine element using the created geometry line and sketch
plane

ModelLine line = document.Create.NewModelCurve(geomLine, sketch) as Model
Line;

// Create a ModelArc element using the created geometry arc and sketch pl
ane

ModelArc arc = document.Create.NewModelCurve(geomArc, sketch) as ModelAr
c;
}

```

Members

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

```
public void GetCurve(ModelCurve 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.

Line Styles

Line styles are represented by the GraphicsStyle class. All line styles for a ModelCurve are available from the GetLineStyleIds() method which returns a set of ElementIds of GraphicsStyle elements.

3.12 Material

In the Revit Platform API, material data is stored and managed as an Element. Just as in the Revit UI, a material can have several assets associated with it, but only thermal and structural (referred to as Physical in the Revit UI) assets can be assigned using the API.

Some material features are represented by properties on the Material class itself, such as FillPattern, Color, or Render while others are available as properties of either a structural or thermal asset associated with the material.

In this chapter, you learn how to access material elements and how to manage the Material objects in the document. [Element Material](#) provides a walkthrough showing how to get a window material.

3.12.1 General Material Information

Before you begin the walkthrough, read through the following section for a better understanding of the Material class.

All Material objects can be retrieved using a Material class filter. 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 Material class.

Properties

A material will have one or more aspects pertaining to rendering appearance, structure, or other major material category. Each aspect is represented by properties on the Material class itself or via one of its assets, structural or thermal. The StructuralAsset class represents the properties of a material pertinent to structural analysis. The ThermalAsset class represents the properties of a material pertinent to energy analysis.

Code Region 19-3: Getting material properties

```
private void ReadMaterialProps(Document document, Material material)
{
    ElementId strucAssetId = material.StructuralAssetId;
    if (strucAssetId != ElementId.InvalidElementId)
    {
        PropertySetElement pse = document.GetElement(strucAssetId) as PropertySetElement;
        if (pse != null)
```

```
{  
  
    StructuralAsset asset = pse.GetStructuralAsset();  
  
    // Check the material behavior and only read if Isotropic  
  
    if (asset.Behavior == StructuralBehavior.Isotropic)  
  
    {  
  
        // Get the class of material  
  
        StructuralAssetClass assetClass = asset.StructuralAssetClass;  
        // Get other material properties  
  
        // Get other material properties  
  
        double poisson = asset.PoissonRatio.X;  
        double youngMod = asset.YoungModulus.X;  
        double thermCoeff = asset.ThermalExpansionCoefficient.X;  
        double unitweight = asset.Density;  
        double shearMod = asset.ShearModulus.X;  
        if (assetClass == StructuralAssetClass.Metal)  
  
        {  
  
            double dMinStress = asset.MinimumYieldStress;  
        }  
        else if (assetClass == StructuralAssetClass.Concrete)  
  
        {  
  
            double dConcComp = asset.ConcreteCompression;  
        }  
    }  
}
```

{}

Classification

The material classification relevant to structural analysis (i.e. steel, concrete, wood) can be obtained from the StructuralAssetClass property of the StructuralAsset associated with the material.

Note: The API does not provide access to the values of Concrete Type for Concrete material.

The material classification relevant to energy analysis (i.e. solid, liquid, gas) can be obtained from the ThermalMaterialType property of the ThermalAsset associated with the material.

Other Properties

The material object properties identify a specific type of material including color, fill pattern, and more.

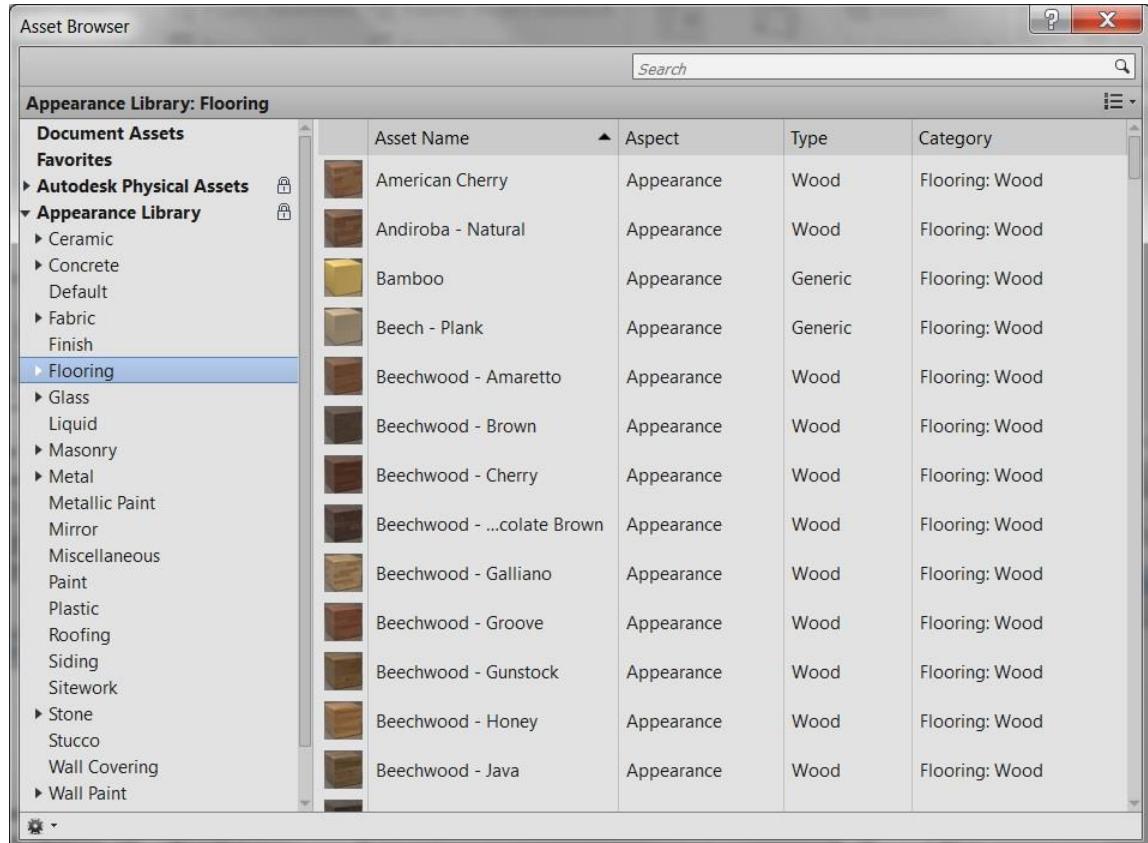
Properties and Parameter

Some Material properties are only available as a Parameter. A few, such as Color, are available as a property or as a Parameter using the BuiltInParameter MATERIAL_PARAM_COLOR.

Rendering Information

Collections of rendering data are organized into objects called Assets. 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.AppearanceAssetId property.

The following figure shows the Appearance Library section of the Asset Browser dialog box, which shows how some rendering assets are displayed in the UI.

**Figure 106: Appearance Library**

The Materials sample application included with the SDK shows how to set the RenderAppearance property to a material selected in a dialog. The dialog is populated with all the Asset objects in Application.Assets.

Editing properties of an Appearance Asset

Editing properties in an appearance Asset requires establishment of an edit scope. The new class

```
Autodesk.Revit.DB.Visual.AppearanceAssetEditScope
```

allows an application to create and maintain an editing session for an appearance asset. The scope provides access to an editable Asset object whose properties may be changed. Once all of the desired changes have been made to the asset's properties, the edit scope should be committed, which causes the changes to be sent back into the document. (This is the only part of the process when a transaction must be opened).

The methods of this class are

- `AppearanceAssetEditScope.Start()` – Starts the edit scope for the asset contained in a particular `AppearanceAssetElement`. The editable Asset is returned from this method.

- AppearanceAssetEditScope.Commit() – Finishes the edit scope: all changes made during the edit scope will be committed. Provides an option to force the update of all open views.
- AppearanceAssetEditScope.Cancel() – Cancels the edit scope and discards any changes.

Connected Assets

Connected assets are associated to properties in appearance assets, and represent subordinate objects encapsulating a collection of related properties. One example of a connected asset is the "Unified Bitmap" representing an image and its mapping parameters and values. AssetProperty offers methods to provide the ability to modify, add or delete the asset connected to a property:

- AssetProperty.GetSingleConnectedAsset() – Gets the single connected asset of this property
- AssetProperty.RemoveConnectedAsset() – Removes the single connected asset of this property
- AssetProperty.AddConnectedAsset (String schemaId) – Create a new default asset of schema type and connects it to this property
- AssetProperty.AddCopyAsConnectedAsset() – Connects the property to a copy of the asset

Schemas & Property names

Appearance asset properties are aligned with specific schemas. Each schema contains necessary properties which define how the appearance of the material will be generated. There are 14 standard material schemas:

- Ceramic
- Concrete
- Generic
- Glazing
- Hardwood
- MasonryCMU
- Metal
- MetallicPaint
- Mirror
- PlasticVinyl
- SolidGlass
- Stone
- WallPaint
- Water

In addition, there are 5 schemas representing "advanced" materials - these may be encountered as a result of import from other Autodesk products:

- AdvancedLayered
- AdvancedMetal
- AdvancedOpaque
- AdvancedTransparent
- AdvancedWood

Finally, there are 10 schemas used for the aspects of the connected assets:

- BumpMap
- Checker
- Gradient
- Marble
- Noise
- Speckle
- Tile
- UnifiedBitmap
- Wave
- Wood

The method:

```
AssetProperty.IsValidSchemaIdentifier(String schemaName)
```

identifies if the input name is a valid name for a supported schema.

To assist in creating code accessing and manipulating the properties of a given schema, predefined properties have been introduced to allow a compile-time reference to a property name without requiring you to transcribe it as a string in your code. These predefined property names are available in static classes named similar to the schema names, above, e.g. Autodesk.Revit.DB.Visual.Ceramic.

Asset Utilities

The method:

- Application.GetAssets(AssetType)

returns a list of assets available to the session.

The method:

- AppearanceAssetElement.Duplicate()

creates a copy of an appearance asset element and the asset contained by it.

The operator:

- Asset.operator[]

accesses a particular AssetProperty associated to the given asset.

FillPattern

All FillPatterns in a document are available using a FilteredElementCollector filtering on class FillPatternElement. A FillPatternElement is an element that contains a FillPattern while the FillPattern class provides access to the pattern name and the set of FillGrids that make up the pattern.

There are two kinds of FillPatterns: Drafting and Model. In the UI, you can only set Drafting fill patterns to Material.CutForegroundPatternId or Material.CutBackgroundPatternId. The fill pattern type is exposed via the FillPattern.Target property. 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)
{
    FilteredElementCollector collector = new FilteredElementCollector(document);
    ICollection<ElementId> fillPatternElements = collector.OfClass(typeof(FillPatternElement)).ToElementIds();
    foreach (ElementId fillPatternId in fillPatternElements)
    {
        material.CutForegroundPatternId = fillPatternId;
        material.SurfaceForegroundPatternId = fillPatternId;
    }
}
```

3.12.2 Material Management

You can use filtering to retrieve all materials in the document. Every Material object in the Document is identified by a unique name.

The following example illustrates how to use the material name to get material.

Code Region 19-5: Getting a material by name

```
public void MaterialByName(Document document)
{
    FilteredElementCollector elementCollector = new FilteredElementCollector(document);
    elementCollector.WherePasses(new ElementClassFilter(typeof(Material)));
    IList<Element> materials = elementCollector.ToElements();

    Material floorMaterial = null;
    string floorMaterialName = "Default Floor";

    foreach (Element materialElement in materials)
    {
        Material material = materialElement as Material;
        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.)

Creating Materials

There are two ways to create a new Material object in the API.

- Duplicate an existing Material
- Add a new Material.

When using the Duplicate() method, the returned Material object is the same type as the original.

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 m
ethod

    //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 static method `Material.Create()` to add a new Material directly. No matter how it is applied, it is necessary to specify a unique name for the material and any assets belonging to the material. The unique name is the `Material` object key.

Code Region 19-7: Adding a new Material

```
public void CreateMaterial(Document document)
{
    //Create the material
    ElementId materialId = Material.Create(document, "My Material");
    Material material = document.GetElement(materialId) as Material;

    //Create a new property set that can be used by this material
    StructuralAsset strucAsset = new StructuralAsset("My Property Set", StructuralAssetClass.Concrete);
    strucAsset.Behavior = StructuralBehavior.Isotropic;
    strucAsset.Density = 232.0;

    //Assign the property set to the material.
    PropertySetElement pse = PropertySetElement.Create(document, strucAsset);
    material.SetMaterialAspectByPropertySet(MaterialAspect.Structural, pse.Id);
}
```

Deleting Materials

To delete a material use:

- Document.Delete()

Document.Delete() is a generic method. See [Editing Elements](#) for details.

3.12.3 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 section and refer to [Walls, Floors, Roofs and Openings](#) 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 [Walls, Floors, Roofs and Openings](#)).
- If the component's material returns null, get the material from the corresponding Element.Category sub Category.

Material in a Parameter

If the Element object has a Parameter with a Definition whose GetDataType() method returns SpecTypeid.Reference.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 Structural 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: Getting an element material from a parameter

```
public void GetMaterialFromParameter(Document document, FamilyInstance family
Instance)

{
    foreach (Parameter parameter in familyInstance.Parameters)
    {
        Definition definition = parameter.Definition;
        // material is stored as element id
    }
}
```

```
if (parameter.StorageType == StorageType.ElementId)
{
    if (definition.GetGroupId() == GroupTypeId.Materials &&
        definition.GetDataType() == SpecTypeId.Reference.Material)
    {
        Autodesk.Revit.DB.Material material = null;

        Autodesk.Revit.DB.ElementId materialId = parameter.AsElementId();

        if (materialId == ElementId.InvalidElementId)
        {
            //Invalid ElementId, assume the material is "By Category"
            if (null != familyInstance.Category)
            {
                material = familyInstance.Category.Material;
            }
        }
        else
        {
            material = document.GetElement(materialId) as Material;
        }
    }

    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 ElementId.InvalidElementId 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.

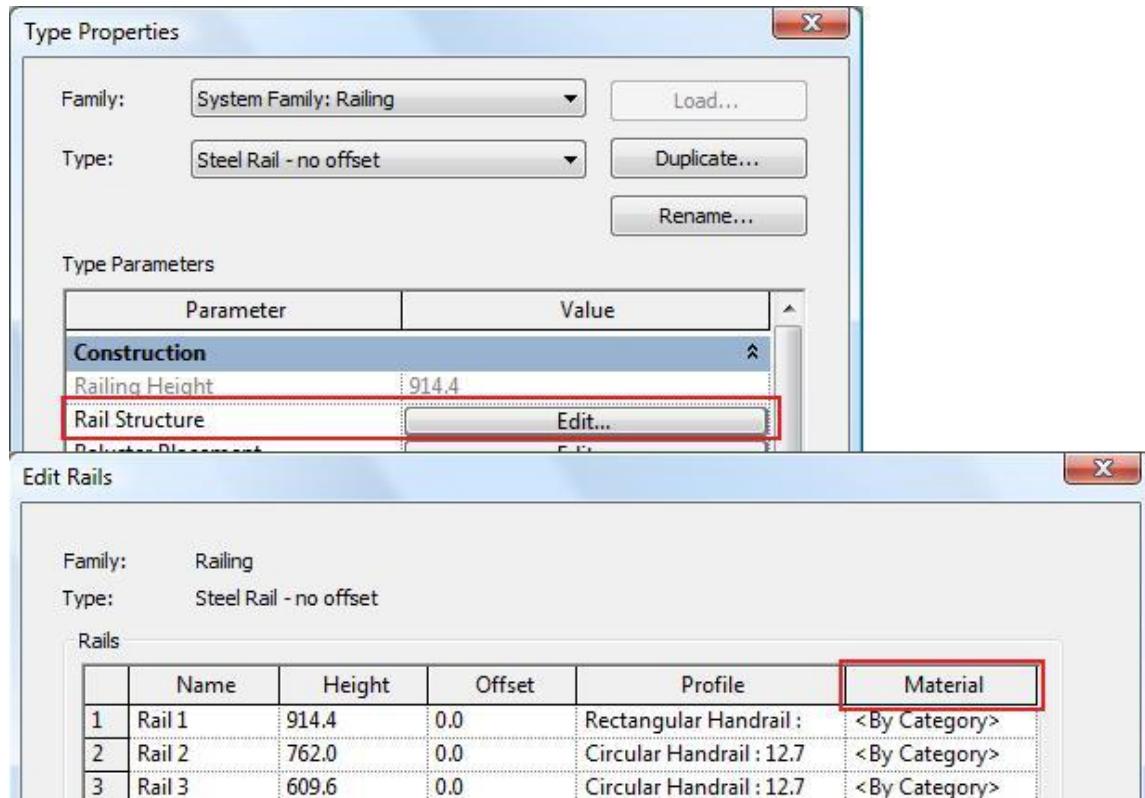


Figure 107: Rail structure property

Material and FamilyInstance

Beam, Column and Foundation FamilyInstances have another way to get their material using their StructuralMaterialId property. This property returns an ElementId which identifies the material that defines the instance's structural analysis properties.

Code Region: Getting an element material from a family instance

```
public Material GetFamilyInstanceMaterial(Document document, FamilyInstance beam)
```

```

    {
        Material material = document.GetElement(beam.StructuralMaterialId) as Material;

        return material;
    }
}

```

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.

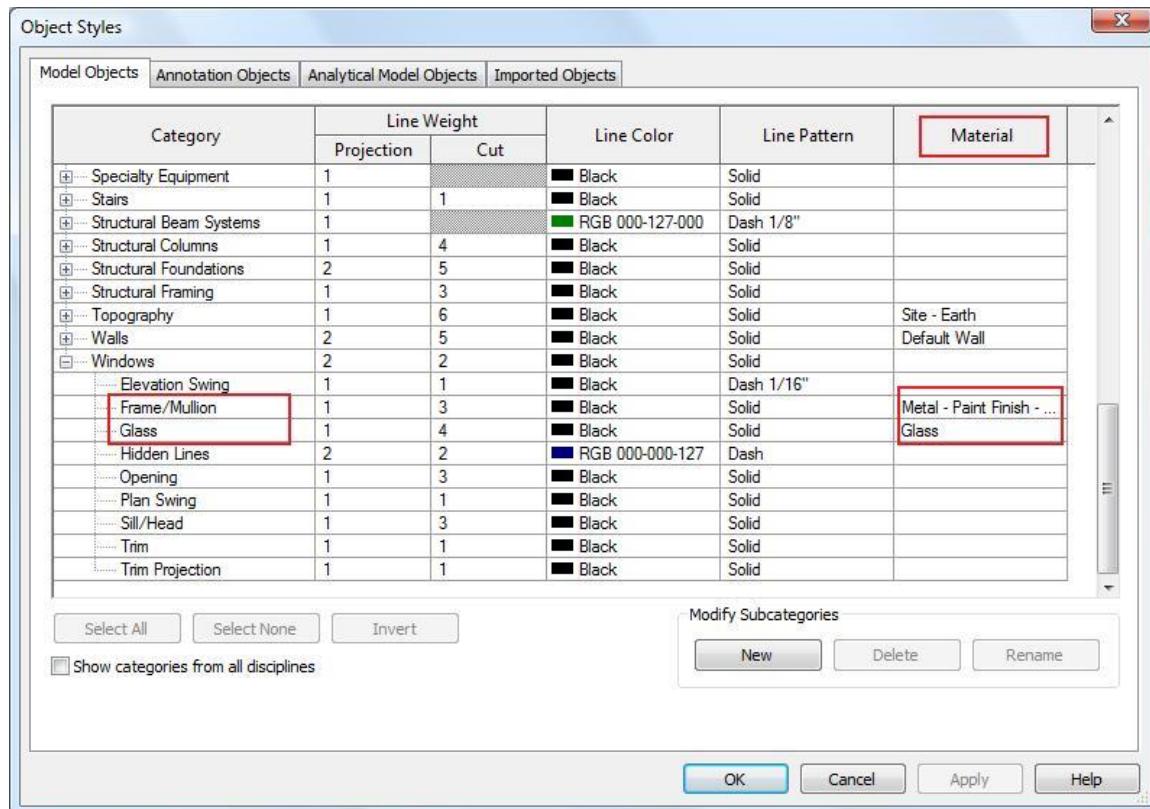


Figure 108: Category material

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 [Elements Essentials](#).

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 ElementId.InvalidElementId.

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 [Element Material](#).

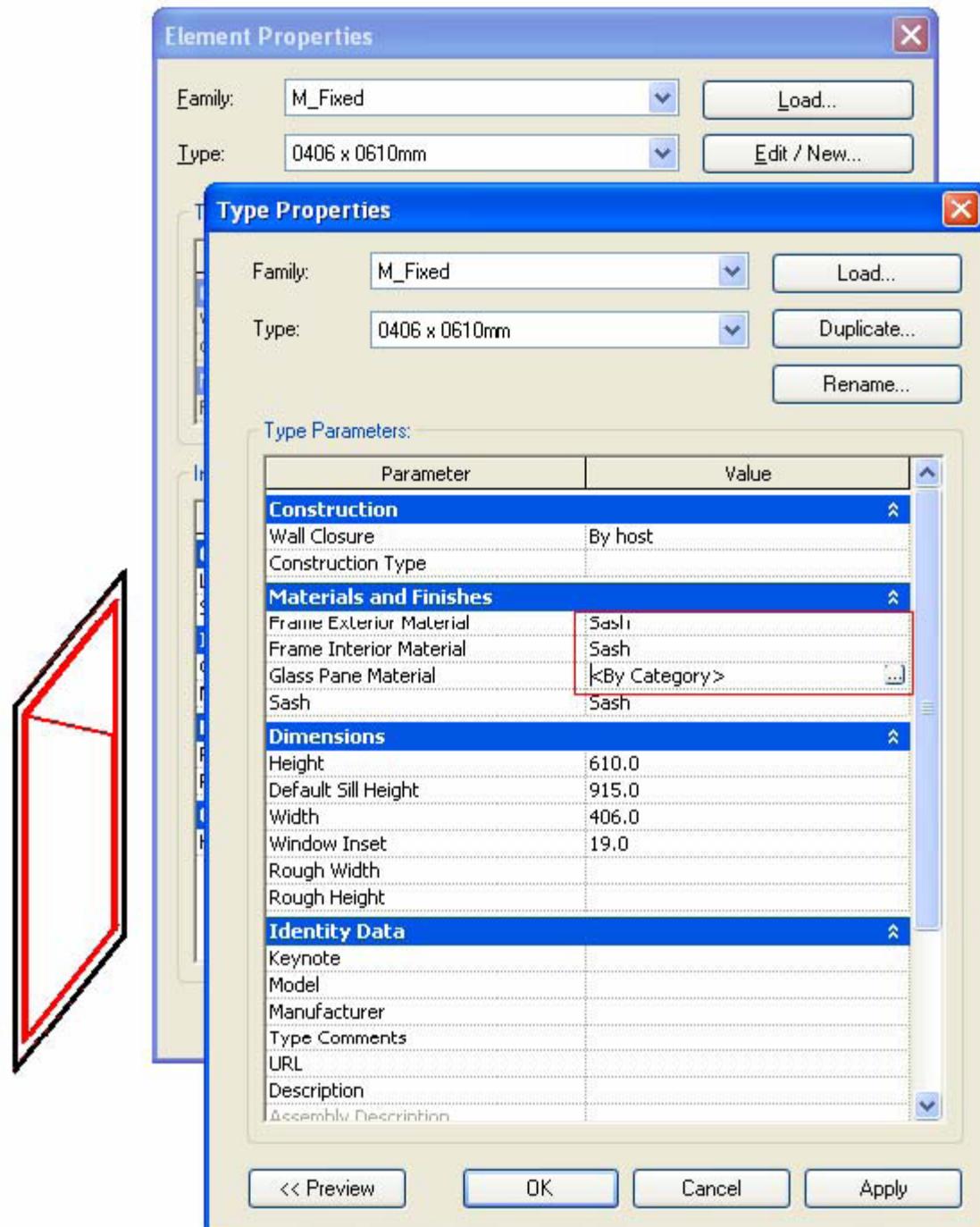


Figure 109: Window material

CompoundStructureLayer Material

You can get the CompoundStructureLayer object from HostObjAttributes. For more details, refer to [Walls, Floors, Ceilings, Roofs and Openings](#).

Retrieve Element Materials

The following diagram shows the workflow to retrieve Element Materials:

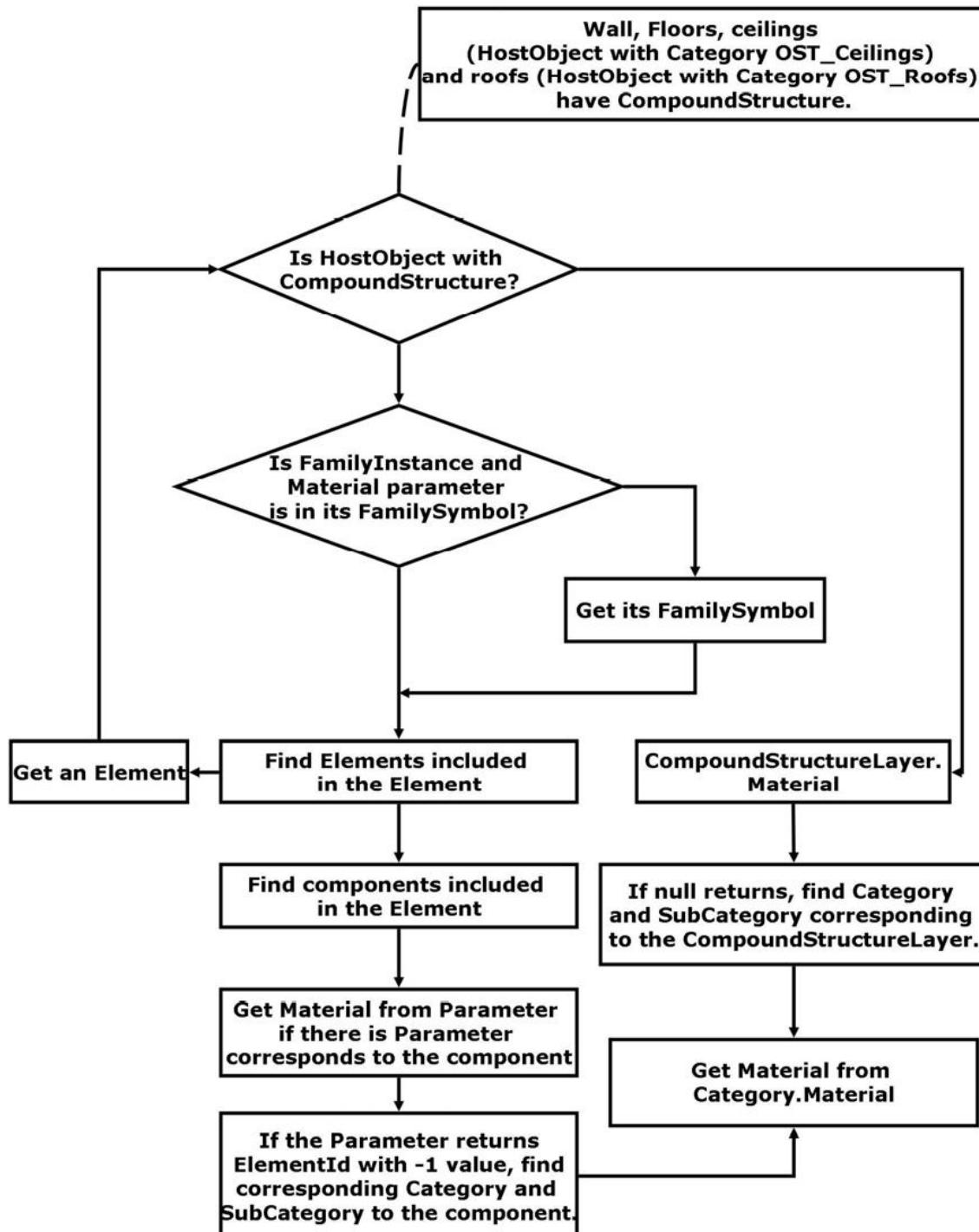


Figure 110: Getting Element Material workflow

This workflow illustrates the following process:

- The workflow shows how to get the 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 Material object from the CompoundStructureLayer class MaterialId property.
 - Others - Get the Material from the Parameters.
- When you get a null Material object or an invalid ElementId with a value of ElementId.InvalidElementId, 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 Structural 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 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.

Walkthrough: Get Window Materials

The following code illustrates how to get the Window Materials.

Code Region: Getting window materials walkthrough

```
public void GetWindowMaterial(Document document, FamilyInstance window)
{
    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 parameters first
    foreach (Parameter parameter in windowSymbol.Parameters)
    {
```

```
switch (parameter.Definition.Name)

{
    case "Frame Exterior Material":
        frameExteriorMaterial = document.GetElement(parameter.AsElementId()) as Material;
        break;

    case "Frame Interior Material":
        frameInteriorMaterial = document.GetElement(parameter.AsElementId()) as Material;
        break;

    case "Sash":
        sashMaterial = document.GetElement(parameter.AsElementId()) as Material;
        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);

    }

```

3.12.4 Material quantities

There are methods to directly obtain the material volumes and areas computed by Revit for material takeoff schedules:

- Element.GetMaterialIds() – obtains a list of materials within an element
- Element.GetMaterialVolume() – obtains the volume of a particular material in an element
- Element.GetMaterialArea() – obtains the area of a particular material in an element

The methods apply to categories of elements where Category.HasMaterialQuantities property is true. In practice, this is limited to elements that use compound structure, like walls, roofs, floors, ceilings, a few other basic 3D elements like stairs, plus 3D families where materials can be assigned to geometry of the family, like windows, doors, columns, MEP equipment and fixtures, and generic model families. Note that within these categories there are further restrictions about how material quantities can be extracted. For example, curtain walls and curtain roofs will not report any material quantities themselves; the materials used by these constructs can be extracted from the individual panel elements that make up the curtain system.

Note that the volumes and areas computed by Revit may be approximate in some cases. For example, for individual layers within a wall, minor discrepancies might appear between the volumes visible in the model and those shown in the material takeoff schedule. These discrepancies tend to occur when you use the wall sweep tool to add a sweep or a reveal to a wall, or under certain join conditions.

The SDK sample “MaterialQuantities” combines both the material quantity extraction tools and temporary suppression of cutting elements (opening, windows, and doors) to extract both gross and net material quantities.

	A	B	C	D	E
1					
2	Totals for Roof elements	Gross volume(cubic ft)	Net volume(cubic ft)	Gross area(sq ft)	Net area(sq ft)
3	Concrete - Precast Concrete	11835.92	11356.5	17753.89	17034.76
4	Insulation / Thermal Barriers - Rigid insulation	9295.36	8921.58	17757.11	17037.9
5	Roofing - EPDM Membrane	369.94	354.96	17757.11	17037.9
6	Default Roof	769.78	769.78	1343.72	1343.72
7					
8	Totals for Roof element Concrete Deck w/Tapered Insulation (id 180471)	Gross volume(cubic ft)	Net volume(cubic ft)	Gross area(sq ft)	Net area(sq ft)
9	Concrete - Precast Concrete	11835.92	11356.5	17753.89	17034.76
10	Insulation / Thermal Barriers - Rigid insulation	9295.36	8921.58	17757.11	17037.9
11	Roofing - EPDM Membrane	369.94	354.96	17757.11	17037.9
12					
13	Totals for Roof element Generic - 12" (id 188333)	Gross volume(cubic ft)	Net volume(cubic ft)	Gross area(sq ft)	Net area(sq ft)
14	Default Roof	578.47	578.47	578.47	578.47
15					
16	Totals for Roof element Generic - 3" (id 246676)	Gross volume(cubic ft)	Net volume(cubic ft)	Gross area(sq ft)	Net area(sq ft)
17	Default Roof	191.31	191.31	765.25	765.25
18					
19	Totals for Wall elements	Gross volume(cubic ft)	Net volume(cubic ft)	Gross area(sq ft)	Net area(sq ft)
20	Masonry - Concrete Masonry Units	9397.86	8860.45	14788.36	13942.65
21	EIFS - Exterior Insulation and Finish System	3606.45	3324.94	10805.74	9961.67
22	Air Barrier - Air Infiltration Barrier	0	0	10859.81	10013.99
23	Vapor / Moisture Barriers - Vapor Retarder	0	0	9022.4	8176.57
24	Gypsum Wall Board	3343.51	3148.03	80218.76	75528.08
25	Metal - Stud Layer	9236.01	8675.3	31666.31	29743.88
26	Concrete - Cast-in-Place Concrete	513.78	513.78	685.04	685.04
27					
28	Totals for Wall element Exterior - EIFS on CMU (id 180225)	Gross volume(cubic ft)	Net volume(cubic ft)	Gross area(sq ft)	Net area(sq ft)
29	Masonry - Concrete Masonry Units	621.26	607.92	977.43	956.43

3.12.5 Painting the Face of an Element

The Paint tool functionality is available through the Revit API. Faces of elements such as walls, floors, and roofs can be painted with a material to change their appearance. It does not change the structure of the element.

The methods related to painting elements are part of the Document class. Document.Paint() applies a material to a specified face of an element. Document.RemovePaint() will remove the applied material. Additionally, the IsPainted() and GetPaintedMaterial() methods return information about the face of an element.

Code Region: Paint faces of a wall

```
// Paint any unpainted faces of a given wall
public void PaintWallFaces(Wall wall, ElementId matId)
```

```
{  
    Document doc = wall.Document;  
  
    GeometryElement geometryElement = wall.get_Geometry(new Options());  
  
    foreach (GeometryObject geometryObject in geometryElement)  
    {  
  
        if (geometryObject is Solid)  
        {  
  
            Solid solid = geometryObject as Solid;  
  
            foreach (Face face in solid.Faces)  
            {  
  
                if (doc.IsPainted(wall.Id, face) == false)  
                {  
  
                    doc.Paint(wall.Id, face, matId);  
  
                }  
            }  
        }  
    }  
}
```

3.13 Stairs and Railings

Stairs

Classes in the Revit API in the Autodesk.Revit.DB.Architecture namespace allow access to stairs and related components such as landings and runs. Stairs can be created or modified using the Revit API. The Stairs class represents stairs created "by component". Stair elements that were created by sketch cannot be accessed as a Stair object in the API. The static method Stairs.IsByComponent() can be used to determine if an ElementId represents stairs that were created by component.

3.13.1 Creating and Editing Stairs

StairsEditScope

As with other types of elements in the Revit document, a Transaction is necessary to edit stairs and stairs components. However, to create new components such as runs and landings, or to create new stairs themselves, it is necessary to use an Autodesk.Revit.DB.StairsEditScope object which maintains a stairs-editing session.

StairsEditScope acts like a TransactionGroup. After a StairsEditScope is started, you can start transactions and edit the stairs. Individual transactions created inside StairsEditScope will not appear in the undo menu. All transactions committed during the edit mode will be merged into a single one which will bear the name passed into the StairsEditScope constructor.

StairsEditScope has two Start methods. One takes the ElementId for an existing Stairs object for which it starts a stairs-editing session. The second Start method takes the ElementIds for base and top levels and creates a new empty stairs element with a default stairs type in the specified levels and then starts a stairs edit mode for the new stairs.

After runs and landings have been added to the Stairs and editing is complete, call StairsEditScope.Commit() to end the stairs-editing session.

Adding Runs

The StairsRun class has three static methods for creating new runs for a Stairs object:

- **CreateSketchedRun** - Creates a sketched run by providing a group of boundary curves and riser curves.
- **CreateStraightRun** - Creates a straight run.
- **CreateSpiralRun** - Creates a spiral run by providing the center, start angle and included angle.

Adding Landings

Either automatic or sketched landings can be added between two runs. The static method StairsLanding.CanCreateAutomaticLanding() will check whether two stairs runs meet restriction to create automatic landing(s). The static StairsLanding.CreateAutomaticLanding() method will return the Ids of all landings created between the two stairs runs.

The static StairsLanding.CreateSketchedLanding method creates a customized landing between two runs by providing the closed boundary curves of the landing. One of the inputs to the CreateSketchedLanding method is a double value for the base elevation. The elevation has following restriction:

- The base elevation is relative to the base elevation of the stairs.
- The base elevation will be rounded automatically to a multiple of the riser height.
- The base elevation should be equal to or greater than half of the riser height.

Example

The following example creates a new Stairs object, two runs (one sketched, one straight) and a landing between them.

Code Region: Creating Stairs, Runs and a Landing

```
public class StairCreation

{
    // FailurePreprocessor class required for StairsEditScope

    class StairsFailurePreprocessor : IFailuresPreprocessor
    {

        public FailureProcessingResult PreprocessFailures(FailuresAccessor failuresAccessor)
        {
            // Use default failure processing

            return FailureProcessingResult.Continue;
        }
    }

    private ElementId CreateStairs(Document document, Level levelBottom, Level levelTop)
    {
        ElementId newStairsId = null;

        using (StairsEditScope newStairsScope = new StairsEditScope(document,
        "New Stairs"))

        {
            newStairsId = newStairsScope.Start(levelBottom.Id, levelTop.Id);

            using (Transaction stairsTrans = new Transaction(document, "Add R
uns and Landings to Stairs"))

            {

```

```
stairsTrans.Start();  
  
// Create a sketched run for the stairs  
IList<Curve> bdryCurves = new List<Curve>();  
IList<Curve> riserCurves = new List<Curve>();  
IList<Curve> pathCurves = new List<Curve>();  
  
XYZ pnt1 = new XYZ(0, 0, 0);  
XYZ pnt2 = new XYZ(15, 0, 0);  
XYZ pnt3 = new XYZ(0, 10, 0);  
XYZ pnt4 = new XYZ(15, 10, 0);  
  
// boundaries  
bdryCurves.Add(Line.CreateBound(pnt1, pnt2));  
bdryCurves.Add(Line.CreateBound(pnt3, pnt4));  
  
// riser curves  
const int riserNum = 20;  
for (int ii = 0; ii <= riserNum; ii++)  
{  
    XYZ end0 = (pnt1 + pnt2) * ii / (double)riserNum;  
    XYZ end1 = (pnt3 + pnt4) * ii / (double)riserNum;  
    XYZ end2 = new XYZ(end1.X, 10, 0);  
    riserCurves.Add(Line.CreateBound(end0, end2));  
}  
  
//stairs path curves
```

```
XYZ pathEnd0 = (pnt1 + pnt3) / 2.0;
XYZ pathEnd1 = (pnt2 + pnt4) / 2.0;
pathCurves.Add(Line.CreateBound(pathEnd0, pathEnd1));

StairsRun newRun1 = StairsRun.CreateSketchedRun(document, new
StairsId, levelBottom.Elevation, bdryCurves, riserCurves, pathCurves);

// Add a straight run
Line locationLine = Line.CreateBound(new XYZ(20, -5, newRun1.
TopElevation), new XYZ(35, -5, newRun1.TopElevation));
StairsRun newRun2 = StairsRun.CreateStraightRun(document, new
StairsId, locationLine, StairsRunJustification.Center);
newRun2.ActualRunWidth = 10;

// Add a landing between the runs
CurveLoop landingLoop = new CurveLoop();
XYZ p1 = new XYZ(15, 10, 0);
XYZ p2 = new XYZ(20, 10, 0);
XYZ p3 = new XYZ(20, -10, 0);
XYZ p4 = new XYZ(15, -10, 0);
Line curve_1 = Line.CreateBound(p1, p2);
Line curve_2 = Line.CreateBound(p2, p3);
Line curve_3 = Line.CreateBound(p3, p4);
Line curve_4 = Line.CreateBound(p4, p1);

landingLoop.Append(curve_1);
landingLoop.Append(curve_2);
landingLoop.Append(curve_3);
```

```
        landingLoop.Append(curve_4);

        StairsLanding newLanding = StairsLanding.CreateSketchedLandin
g(document, newStairsId, landingLoop, newRun1.TopElevation);

        stairsTrans.Commit();

    }

    // A failure preprocessor is to handle possible failures during t
he edit mode commitment process.

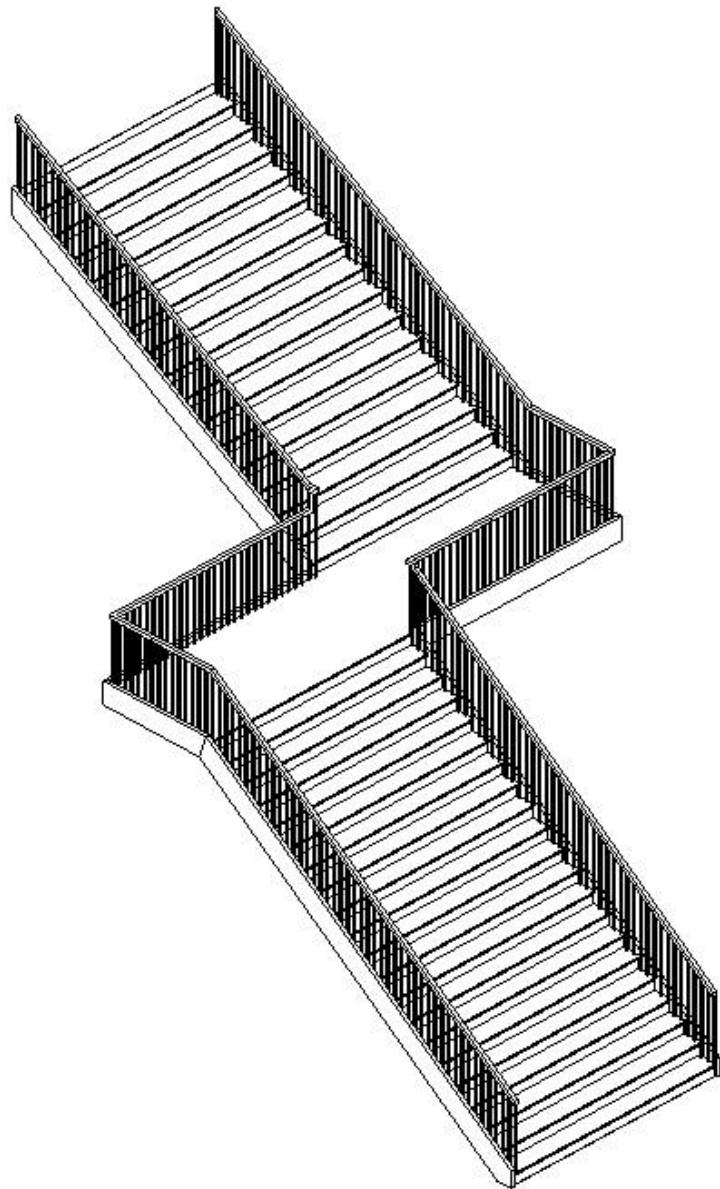
    newStairsScope.Commit(new StairsFailurePreprocessor());

}

return newStairsId;
}

}
```

The stairs resulting from the above example:



Multistory Stairs

The `MultistoryStairs` class allows stairs to span multiple levels. A multistory stairs element may contain multiple stairs whose extents are governed by base and top levels.

This element will contain one or more `Stairs` elements. `Stairs` elements are either:

- a reference instance which is copied to each level covered by groups of identical stairs instances which share the same level height,
- or individual `Stairs` instances which are not connected to a group with the same level height.

By default, when adding new levels to the multistory stair, new stairs will be added to the group. For groups of duplicate stairs at different levels, the instances can be found as Subelements of

the Stairs element. Stairs in a connected group can be edited together by modifying the associated Stairs instance. For specific floors that need special designs, stairs can be separated from the group with the Unpin method - changes made to the unpinned Stairs will not affect other any other instance in the element. The stairs can later be added back into the group with the Pin method, however any changes made to the stair will be lost since the stair's properties will be overridden by the group specifications.

Code Region: Create Multistory Stairs

```
public void CreateMultiStory(Stairs stairs)
{
    Document doc = stairs.Document;
    // create new MultistoryStairs
    MultistoryStairs multistoryStairs = MultistoryStairs.Create(stairs);

    // get all levels that can be connected to this multistoryStairs
    IEnumerable<ElementId> levelIds = new FilteredElementCollector(doc).OfClas
ss(typeof(Level)).Cast<Level>().Where(q => multistoryStairs.CanConnectLevel
(q.Id))
    .Select(q => q.Id);

    // Connect the levels to the multistoryStairs
    // The input to ConnectLevels is a HashSet or SortedSet, so a HashSet is
    // created from the IEnumerable returned by FilteredElementCollector
    multistoryStairs.ConnectLevels(new HashSet<ElementId>(levelIds));
}
```

When new stairs are created using the `StairsEditScope.Start(ElementId, ElementId)` method, they have default railings associated with them. However, the `Railing.Create()` method can be used to create new railings with the specified railing type on all sides of a stairs element for stairs without railings. Unlike run and landing creation which require the use of `StairsEditScope`, railing creation cannot be performed inside an open stairs editing session.

Since railings cannot be created for Stairs that already have railings associated with them, the following example deletes the existing railings associated with a Stairs object before creating new railings.

Code Region: Create Railings

```
private void CreateRailing(Document document, Stairs stairs)
{
    using (Transaction trans = new Transaction(document, "Create Railings"))
    {
        trans.Start();

        // Delete existing railings
        ICollection<ElementId> railingIds = stairs.GetAssociatedRailings();
        foreach (ElementId railingId in railingIds)
        {
            document.Delete(railingId);
        }

        // Find RailingType
        FilteredElementCollector collector = new FilteredElementCollector(document);
        ICollection<ElementId> RailingTypeIds = collector.OfClass(typeof(RailingType)).ToElementIds();

        Railing.Create(document, stairs.Id, RailingTypeIds.First(), RailingPlacementPosition.Treads);

        trans.Commit();
    }
}
```

3.13.2 Railings

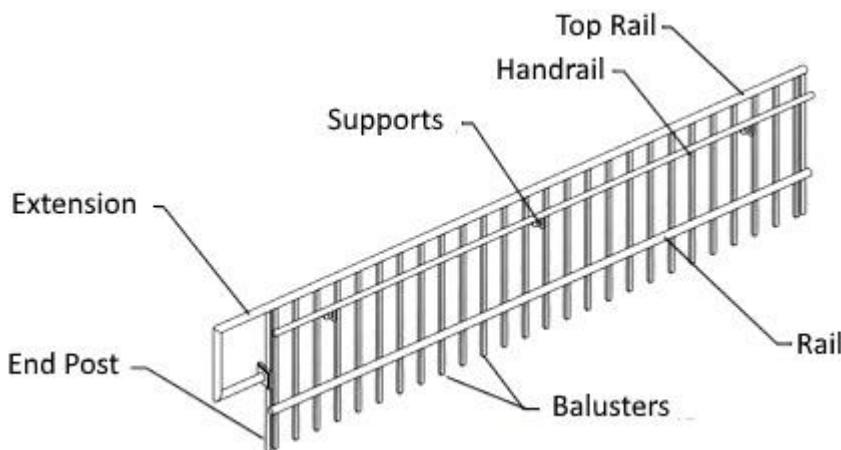
The Autodesk.Revit.DB.Architecture.Railing class represents a railing element in an Autodesk Revit project. Although railings are associated with stairs, they can be associated with another host (such as a floor) or placed in space.

A Railing may be associated with up to five continuous rails and a RailingType which contains information about balusters and any number of non-continuous rails. Continuous rails may either be a top rail or a hand rail.

Railings represent the entire structure which includes continuous rails, non-continuous rails, balusters, and corner posts. Continuous rails refer to the top rail (TopRail class is a subclass of ContinuousRail) and the two possible hand rails (HandRail class is a subclass of ContinuousRail), each of which may appear on the left, right, or both left and right which counts as 2 of the 5 possible continuous rails. The continuous rails are directly accessible from the Railing via the TopRail property and the GetHandRails() method.

A non-continuous rail is a rail which runs parallel to the Railing's path elevated by a height not greater than the railing's height. A default-generated railing's non-continuous rails are broken into sections by the balusters. A Railing may have zero or more rails and must have at least one baluster. The non-continuous rails and baluster placement are accessible via the RailingType.

Railings associated with stairs can be retrieved from the Stairs class with the GetAssociatedRailings() method. There are only a few properties and methods specific to railings such as the TopRail property which returns the ElementId of the top rail and Flipped which indicates if the Railing is flipped. The Railing.Flip() method flips the railing (for a stair-hosted railing, flip changes the railing position between placement on the treads or stringers), while the RemoveHost() method will remove the association between the railing and its host.



The following example retrieves all of the railings associated with a Stairs object and flips each railing that is the default railing that the system generates.

Code Region: Working with Railings

```
private void FlipDefaultRailings(Stairs stairs)
{
    ICollection<ElementId> railingIds = stairs.GetAssociatedRailings();
```

```

    Transaction trans = new Transaction(stairs.Document, "Flip Railings");

    trans.Start();

    foreach (ElementId railingId in railingIds)
    {
        Railing railing = stairs.Document.GetElement(railingId) as Railing;

        if (railing.IsDefault == true)
        {
            railing.Flip();
        }
    }

    trans.Commit();
}

```

The Railing class has a Create method which automatically creates new railings with the specified railing type on all sides of a stairs element. Railing creation is demonstrated in the [Creating and Editing Stairs](#) section.

The RailingType class represents the railing type used in the generation of a railing. It contains a number of properties about the railing, such as the height, lateral offset and type of the primary and secondary handrails as well as the top rail.

Code Region: RailingType

```

private void GetRailingType(Stairs stairs)
{
    ICollection<ElementId> railingIds = stairs.GetAssociatedRailings();

    foreach (ElementId railingId in railingIds)
    {
        Railing railing = stairs.Document.GetElement(railingId) as Railing;
    }
}

```

```

        RailingType railingType = stairs.Document.GetElement(railing.GetTypeId())
d()) as RailingType;

        // Format railing type info for display
        string info = "Railing Type: " + railingType.Name;
        info += "\nPrimary Handrail Height: " + railingType.PrimaryHandrailHeight;
        info += "\nTop Rail Height: " + railingType.TopRailHeight;

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

```

Non-continuous Railings

RailingType.RailStructure

provide access to a collection of information about non-continuous rails that are part of a RailingType. The Autodesk.Revit.DB.Architecture.NonContinuousRailStructure returns a collection of Autodesk.Revit.DB.Architecture.NonContinuousRailInfo objects, each of which represents the properties needed to define a single non-continuous rail.

Baluster Placement

RailingType.BalusterPlacement

provides access to the baluster and post placement information for a given railing type. The Autodesk.Revit.DB.Architecture.BalusterPlacement that it returns may contain instances of:

- Autodesk.Revit.DB.Architecture.BalusterPattern which represents the baluster pattern properties, containing 1 or more objects of the type BalusterInfo.
- Autodesk.Revit.DB.Architecture.PostPattern which represents the post pattern properties, containing up to 3 objects of the type BalusterInfo
- Autodesk.Revit.DB.Architecture.BalusterInfo which represents the properties governing an instance of a single railing baluster or post

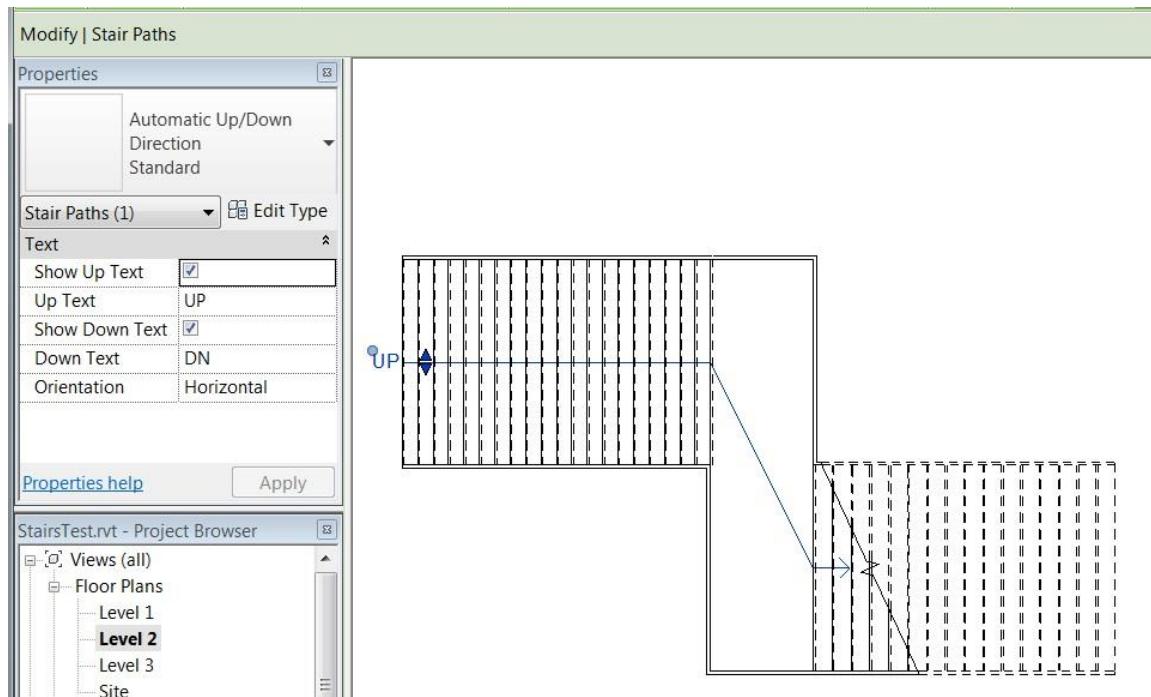
To get the name used to reference the Host or Top Rail, use:

- BalusterInfo.GetReferenceNameForHost()
- BalusterInfo.GetReferenceNameForTopRail()

3.13.3 Stairs Annotations

The `StairsPath` class can be used to annotate the slope direction and walk line of a stair. The static `StairsPath.Create()` method will create a new stairs path for the specified stairs with the specified stairs path type in a specific plan view in which the stairs must be visible.

The `StairsPath` class has the same properties that are available in the Properties window when editing a stairs path in the Revit UI, such as properties to set the up and down text along or whether the text should be shown at all. Additionally offsets for the up and down text can be specified as can an offset for the stairs path from the stairs centerline.



The following example finds a `StairsPathType` and a `FloorPlan` in the project and uses them to create a new `StairsPath` for a given `Stairs`.

Code Region: Create a StairsPath

```
private void CreateStairsPath(Document document, Stairs stairs)
{
    Transaction transNewPath = new Transaction(document, "New Stairs Path");
}
```

```
transNewPath.Start();

// Find StairsPathType

FilteredElementCollector collector = new FilteredElementCollector(document);

ICollection<ElementId> stairsPathIds = collector.OfClass(typeof(StairsPathType)).ToElementIds();

// Find a FloorPlan

ElementId planViewId = ElementId.InvalidElementId;

FilteredElementCollector viewCollector = new FilteredElementCollector(document);

ICollection<ElementId> viewIds = viewCollector.OfClass(typeof(View)).ToElementIds();

foreach (ElementId viewId in viewIds)

{

    View view = document.GetElement(viewId) as View;

    if (view.ViewType == ViewType.FloorPlan)

    {

        planViewId = view.Id;

        break;

    }

}

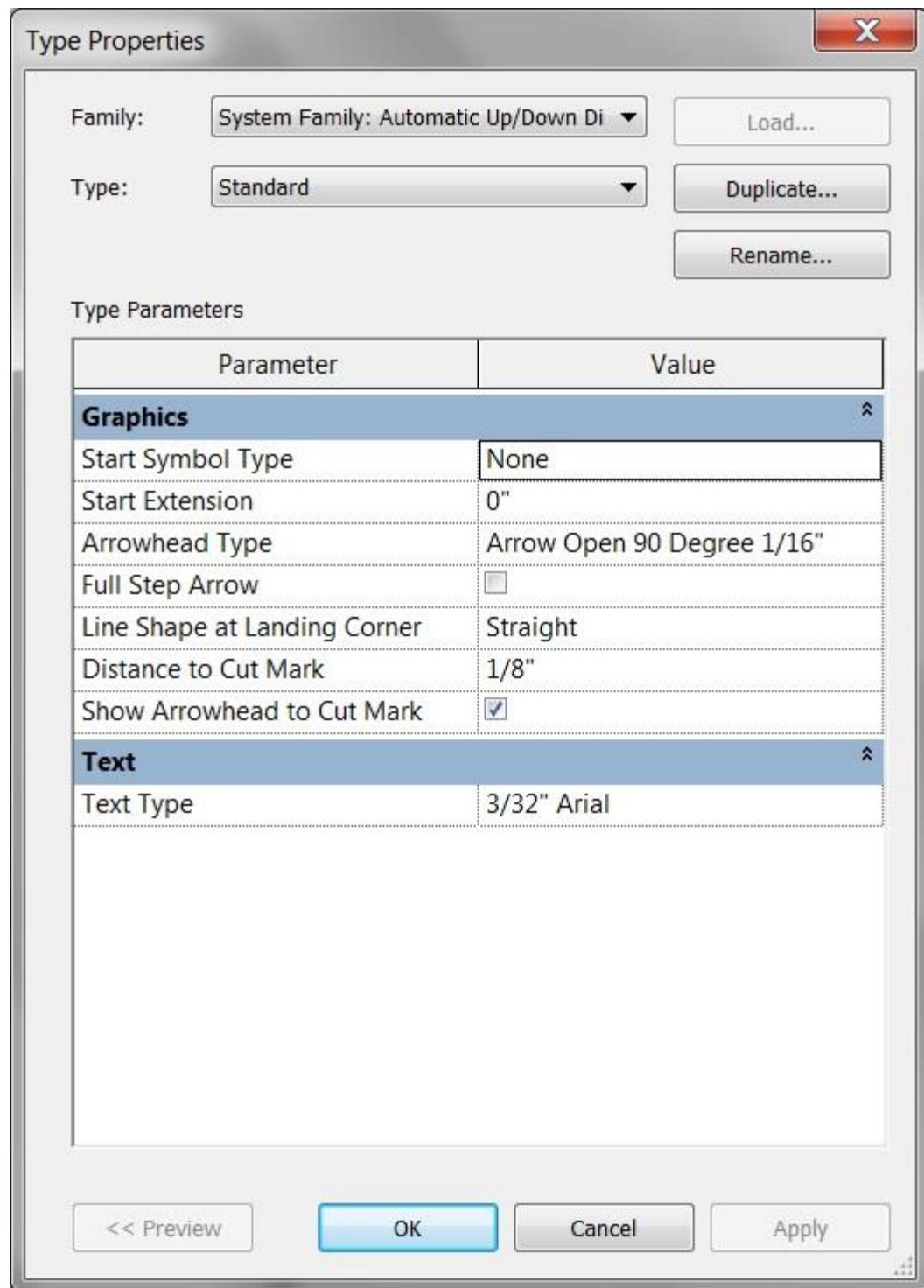
LinkElementId stairsLinkId = new LinkElementId(stairs.Id);

StairsPath.Create(stairs.Document, stairsLinkId, stairsPathIds.First(), planViewId);

transNewPath.Commit();
```

}

A StairsPath has a StairsPathType. Stair path types are available from 2 predefined system families: Automatic Up/Down Direction and Fixed Up Direction. The properties available for these two types are available as properties in the StairsPathType class, such as FullStepArrow and DistanceToCutMark.



The `CutMarkType` class represents a cut mark type in the Revit UI and it has properties to represent the same properties available when editing a cut mark type in the UI, such as `CutLineAngle` and `CutLineExtension`. It is associated with a `StairsType` object and can be retrieved using the `BuiltInParameter STAIRSTYPE_CUTMARK_TYPE` as shown below.

Code Region: Getting the CutMarkType for Stairs

```
private CutMarkType GetCutMark(Stairs stairs)
{
    CutMarkType cutMarkType = null;

    StairsType stairsType = stairs.Document.GetElement(stairs.GetTypeId()) as
    StairsType;

    Parameter paramCutMark = stairsType.GetParameter(ParameterTypeId.Stairsty
peCutmarkType);

    if (paramCutMark.StorageType == StorageType.ElementId) // should be an e
lement id

    {
        ElementId cutMarkId = paramCutMark.AsElementId();

        cutMarkType = stairs.Document.GetElement(cutMarkId) as CutMarkType;
    }

    return cutMarkType;
}
```

3.13.4 Stairs Components

The Stairs class represents a stairs element in Revit and contains properties that represent information about the treads, risers, number of stories, as well as the height of the stairs and base and top elevation. Methods of the Stairs class can be used to get the stairs landing components, stairs run components and stairs supports.

The following example finds all of the Stairs that are by component and outputs some information on each of the Stairs to a Task Dialog. Note that this example uses a category filter with the BuiltInCategory.OST_Stairs which will return ElementIds for all stairs, therefore requiring a test to see if each ElementId represents a Stairs By Component before being cast to a Stairs class when retrieved from the document.

Code Region: Getting stairs information

```
private Stairs GetStairInfo(Document document)

{
    Stairs stairs = null;

    FilteredElementCollector collector = new FilteredElementCollector(document)
        .WhereElementIsNotElementType()
        .OfCategory(BuiltInCategory.OST_Stairs).ToElementIds();

    foreach (ElementId stairId in stairsIds)
    {
        if (Stairs.IsByComponent(document, stairId) == true)
        {
            stairs = document.GetElement(stairId) as Stairs;

            // Format the information

            String info = "\nNumber of stories: " + stairs.NumberOfStories;
            info += "\nHeight of stairs: " + stairs.Height;
            info += "\nNumber of treads: " + stairs.ActualTreadsNumber;
            info += "\nTread depth: " + stairs.ActualTreadDepth;

            // Show the information to the user.

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

    return stairs;
}
```

{

The StairsType class represents the type for a Stairs element. It contains information about the Stairs, such as the type of all runs and landings in the stairs object, types and offsets for supports on the left, right and middle of the stairs, and numerous other properties related to stair generation such as the maximum height of each riser on the stair element. The following example gets the StairsType for a Stairs element and displays some information about it in a TaskDialog.

Code Region: Getting StairsType Info

```
private void GetStairsType(Stairs stairs)
{
    StairsType stairsType = stairs.Document.GetElement(stairs.GetTypeId()) as
    StairsType;

    // Format stairs type info for display
    string info = "Stairs Type: " + stairsType.Name;
    info += "\nLeft Lateral Offset: " + stairsType.LeftLateralOffset;
    info += "\nRight Lateral Offset: " + stairsType.RightLateralOffset;
    info += "\nMax Riser Height: " + stairsType.MaxRiserHeight;
    info += "\nMin Run Width: " + stairsType.MinRunWidth;

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

Runs

Stairs by component are composed of runs, landings and supports. Each of these items can be retrieved from the Stairs class. A run is represented in the Revit API by the StairsRun class. The following example gets each run for a Stairs object and makes sure that it begins and ends with a riser.

Code Region: Working with StairsRun

```
private void AddStartandEndRisers(Stairs stairs)
{
    ICollection<ElementId> runIds = stairs.GetStairsRuns();

    foreach (ElementId runId in runIds)
    {
        StairsRun run = stairs.Document.GetElement(runId) as StairsRun;
        if (null != run)
        {
            run.BeginsWithRiser = true;
            run.EndsWithRiser = true;
        }
    }
}
```

The StairsRun class provides access to run properties such as the StairsRunStyle (straight, winder, etc.), BaseElevation, TopElevation, and properties about the risers. There are also methods in the StairsRun class to access the supports hosted by the run, either all, or just those on the left or right side of the run boundaries. The GetStairsPath() method will return the curves representing the stairs path on the run, which are projected onto the base level of the stairs. The GetFootprintBoundary() method returns the run's boundary curves which are also projected onto the stairs' base level.

There are three static methods of the StairsRun class for creating new runs. These are covered in the [Creating and Editing Stairs](#) section.

The StairsRunType class represents the type of a StairsRun. It contains many properties about the treads and risers of the run as well as other information about the run. The following example gets the StairsRunType for the first run in a Stairs element and displays the riser and tread thicknesses along with the type's name.

Code Region: Getting StairsRunType Info

```
private void GetRunType(Stairs stairs)

{
    ICollection<ElementId> runIds = stairs.GetStairsRuns();

    ElementId firstRunId = runIds.First();

    StairsRun firstRun = stairs.Document.GetElement(firstRunId) as StairsRun;

    if (null != firstRun)

    {
        StairsRunType runType = stairs.Document.GetElement(firstRun.GetTypeId
()) as StairsRunType;

        // Format landing type info for display

        string info = "Stairs Run Type: " + runType.Name;
        info += "\nRiser Thickness: " + runType.RiserThickness;
        info += "\nTread Thickness: " + runType.TreadThickness;

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

Landings

Landings are represented by the `StairsLanding` class. The following example finds the thickness for each landing of a `Stairs` object.

Code Region: Working with `StairsLanding`

```
private void GetStairLandings(Stairs stairs)
```

```
{
    ICollection<ElementId> landingIds = stairs.GetStairsLandings();

    string info = "Number of landings: " + landingIds.Count;

    int landingIndex = 0;

    foreach (ElementId landingId in landingIds)
    {
        landingIndex++;

        StairsLanding landing = stairs.Document.GetElement(landingId) as StairsLanding;

        if (null != landing)
        {
            info += "\nThickness of Landing " + landingIndex + ": " + landing.Thickness;
        }
    }

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

Similar to StairsRun, StairsLanding has a GetStairsPath() method which returns the curves representing the stairs path on the landing projected onto the base level of the stairs and a GetFootprintBoundary() method which returns the landing's boundary curves, also projected onto the stairs' base level. Also similar to StairsRun, there is a method to get all of the supports hosted by the landing.

The StairsLanding class has a method to create a new landing between two runs. It is covered in the [Creating and Editing Stairs](#) section.

The StairsLandingType class represents a landing type in the Revit API. The StairsLandingType class has only a couple of properties specific to it, namely IsMonolithic which is true if the stairs landing is monolithic, and Thickness, representing the thickness of the stairs landing.

StairsComponentConnection

Both StairsRun and StairsLanding have a GetConnections() method which provides information about connections among stairs components (run to run, or run to landing). The method returns a collection of StairsComponentConnection objects which have properties about each connection, including the connection type (to a landing, the start of a stairs run, or the end of a stairs run) and the Id of the connected stairs component.

Supports

The Revit API does not expose a class for stairs supports. When getting the supports for Stairs, StairsRun, or a StairsLanding, the supports will be generic Revit Elements. The following example gets the names of all the supports for a Stairs object.

Code Region: Getting Stairs Supports

```
private void GetStairSupports(Stairs stairs)
{
    ICollection<ElementId> supportIds = stairs.GetStairsSupports();
    string info = "Number of supports: " + supportIds.Count;

    int supportIndex = 0;
    foreach (ElementId supportId in supportIds)
    {
        supportIndex++;
        Element support = stairs.Document.GetElement(supportId);
        if (null != support)
        {
            info += "\nName of support " + supportIndex + ": " + support.Name;
        }
    }
    TaskDialog.Show("Revit", info);
}
```

3.14 Surfaces

The surface class represents the mathematical representation of a surface.

The surface class is not derived from the GeometryObject class and not bounded by edges or edge loops. A bounded surface in Revit is represented by the [Face](#) class.

Surface is the base class for more specific surfaces:

- Plane
- CylindricalSurface
- ConicalSurface
- RuledSurface
- RevolvedSurface
- HermiteSurface
- OffsetSurface - Represents a surface offset by a fixed distance in the direction of the originating surface normal. The signed offset distance is in the direction of the originating surface's oriented normal vector at any given point, which can be the same as or opposite to the originating surface's parametric normal vector.

These subclasses contain Create() methods and read-only properties and are suitable for constructing import geometry. See the [DirectShape](#) topic for examples of using Surfaces in geometry creation.

3.15 DirectShape

This element type can store arbitrary geometry created by the Revit API or obtained from import operations or calculations in either a project or family document.

The DirectShape element and related classes support the ability to store externally created geometric shapes in a Revit document. The geometry can include closed solids or meshes. DirectShape is primarily intended for importing shapes from other data formats such as IFC or STEP where not enough information is available to create a "real" Revit element.

A DirectShape object may be assigned a top-level Model category, such as the Wall category. Sub-categories cannot be assigned to DirectShape elements. The IsValidCategoryId() method can test a category id to make sure it is a top-level built-in category approved for use with DirectShape and the Category.CategoryType enumerated value will indicate if the category type is Model. Assigning a category will affect how that object is displayed in Revit and will grant the object a collection of available parameters and some limited behaviors.

Creation

The static CreateElement() method will create a new instance-level DirectShape. It requires the document in which the DirectShape will be added and the id of an appropriate built-in category.

DirectShape provides the ApplicationId and ApplicationDataId string parameters that provide context for the source of the created shape.

Once the DirectShape is created, the shape can be set using one of the overloaded SetShape() method. The shape can be set either directly from a ShapeBuilder object or from a list of GeometryObjects. If you are using a ShapeBuilder object to construct geometry for the DirectShape anyway, there may be a slight performance advantage to using the ShapeBuilder input, as Revit will bypass repetitive validation of the input geometry. It is also possible to append additional geometry objects to the DirectShape using versions of the AppendShape() method. Note that AppendShape() will not merge or join the incoming geometry with any geometry already present, the geometry will be stored independently.

DirectShapes accept the following geometry types as input:

- Solid (this can be a closed or open shell)
- Mesh
- Curve
- Point

In addition, you can specify geometry to be used in a view-specific representation of a DirectShape. This geometry is input along with the input of a DirectShapeTargetViewType. When setting a view-specific shape representation, it will only be used in views of that type. Currently the only view-specific representation supported is for Plan views.

`DirectShapeType.SetFamilyName()` provides the ability to set a custom Family name for a DirectShapeType. The validator: `DirectShapeType.CanChangeFamilyName()` provides the ability to check if a DirectShapeType supports a custom Family name because certain categories do not support custom Family names.

The following example demonstrates how to create a simple DirectShape from a sphere created using the GeometryCreationUtilities class. Note the use of a reference Frame in creating the geometry. Prior to using it to create geometry, it is good practice to call the static method Frame.CanDefineRevitGeometry() which tests whether the supplied Frame object may be used to define a Revit curve or surface. In order to satisfy the requirements the Frame must be orthonormal and its origin is expected to lie within the Revit design limits. (When creating geometry, it also can be useful to use the static XYZ.IsWithinLengthLimits() to ensure the point is within Revit design limits.)

Code Region: Create a DirectShape

```
// Create a DirectShape Sphere
public void CreateSphereDirectShape(Document doc)
{
    List<Curve> profile = new List<Curve>();
```

```
// first create sphere with 2' radius

XYZ center = XYZ.Zero;
double radius = 2.0;

XYZ profile00 = center;

XYZ profilePlus = center + new XYZ(0, radius, 0);
XYZ profileMinus = center - new XYZ(0, radius, 0);

profile.Add(Line.CreateBound(profilePlus, profileMinus));
profile.Add(Arc.Create(profileMinus, profilePlus, center + new XYZ(radius, 0, 0)));

CurveLoop curveLoop = CurveLoop.Create(profile);

SolidOptions options = new SolidOptions(ElementId.InvalidElementId, ElementId.InvalidElementId);

Frame frame = new Frame(center, XYZ.BasisX, -XYZ.BasisZ, XYZ.BasisY);

if (Frame.CanDefineRevitGeometry(frame) == true)

{
    Solid sphere = GeometryCreationUtilities.CreateRevolvedGeometry(frame, new CurveLoop[] { curveLoop }, 0, 2 * Math.PI, options);

    using (Transaction t = new Transaction(doc, "Create sphere direct shape"))

    {
        t.Start();

        // create direct shape and assign the sphere shape

        DirectShape ds = DirectShape.CreateElement(doc, new ElementId(BuiltInCategory.OST_GenericModel));
    }
}
```

```

        ds.ApplicationId = "Application id";
        ds.ApplicationDataId = "Geometry object id";
        ds.SetShape(new GeometryObject[] { sphere });
        t.Commit();
    }
}
}

```

The geometry for a DirectShape can also be created using a subclass of the ShapeBuilder class or from a TessellatedShapeBuilder.

ShapeBuilder

ViewShapeBuilder and WireframeBuilder can be used to create geometry to store in a DirectShape class. The ViewShapeBuilder class builds and verifies a view-specific shape representation. It is limited to curve-based representations for plan views. WireframeBuilder constructs a 3D shape representation consisting of points and curves. Both types of ShapeBuilders can be applied to a DirectShape element using the DirectShape.SetShape() or DirectShape.AppendShape() overload that takes a ShapeBuilder parameter.

TessellatedShapeBuilder

TessellatedShapeBuilder can be used create solid, shell, or polymeshes bounded by a set of connected planar facets, created by adding TessellatedFace objects one by one. Faces can only be added to the build while a face set is "open". Use the OpenConnectedFaceSet() method to open a face set. After adding all the TessellatedFaces, call CloseConnectedFaceSet() to close the face set. The builder allows for the possibility of multiple face sets - in such cases the first set should represent the outer 'surface' of a body and all following sets represent interior voids. The builder tries to create a geometry valid in Revit despite inconsistencies or omissions in the input data.

After defining all faces and closing the face set, call the Build() method to build the designated geometrical objects from the stored face sets. The Target, Fallback and GraphicsStyleId properties of the TessellatedShapeBuilder can be set prior to calling Build() or default options will be used. The results of Build() are stored in the TessellatedShapeBuilder and can be retrieved by calling GetBuildResult(). The TessellatedShapeBuilderResult.GetGeometricalObjects() method will return a list of GeometryObjects which can be used with the corresponding DirectShape.SetShape() or DirectShape.AppendShape() overload, as shown in the example below.

Code Region: Create a DirectShape using TessellatedShapeBuilder

```
// Create a pyramid-shaped DirectShape using given material for the faces

public void CreateTessellatedShape(Document doc, ElementId materialId)
{
    List<XYZ> loopVertices = new List<XYZ>(4);

    TessellatedShapeBuilder builder = new TessellatedShapeBuilder();

    builder.OpenConnectedFaceSet(true);

    // create a pyramid with a square base 4' x 4' and 5' high

    double length = 4.0;
    double height = 5.0;

    XYZ basePt1 = XYZ.Zero;
    XYZ basePt2 = new XYZ(length, 0, 0);
    XYZ basePt3 = new XYZ(length, length, 0);
    XYZ basePt4 = new XYZ(0, length, 0);
    XYZ apex = new XYZ(length / 2, length / 2, height);

    loopVertices.Add(basePt1);
    loopVertices.Add(basePt2);
    loopVertices.Add(basePt3);
    loopVertices.Add(basePt4);

    builder.AddFace(new TessellatedFace(loopVertices, materialId));

    loopVertices.Clear();
    loopVertices.Add(basePt1);
```

```
loopVertices.Add(apex);

loopVertices.Add(basePt2);

builder.AddFace(new TessellatedFace(loopVertices, materialId));

loopVertices.Clear();

loopVertices.Add(basePt2);

loopVertices.Add(apex);

loopVertices.Add(basePt3);

builder.AddFace(new TessellatedFace(loopVertices, materialId));

loopVertices.Clear();

loopVertices.Add(basePt3);

loopVertices.Add(apex);

loopVertices.Add(basePt4);

builder.AddFace(new TessellatedFace(loopVertices, materialId));

loopVertices.Clear();

loopVertices.Add(basePt4);

loopVertices.Add(apex);

loopVertices.Add(basePt1);

builder.AddFace(new TessellatedFace(loopVertices, materialId));

builder.CloseConnectedFaceSet();

builder.Target = TessellatedShapeBuilderTarget.Solid;

builder.Fallback = TessellatedShapeBuilderFallback.Abort;

builder.Build();
```

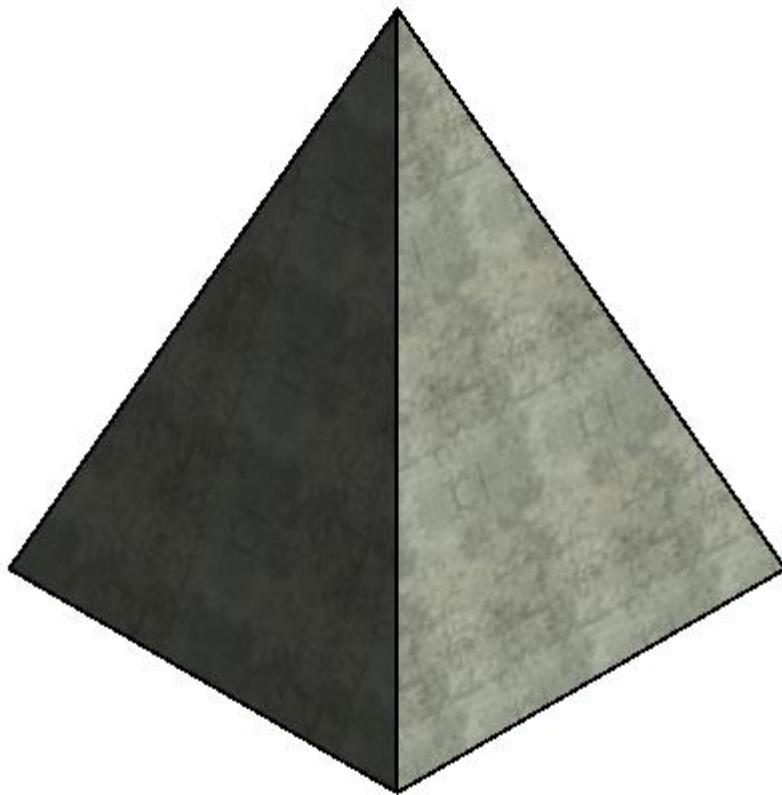
```
TessellatedShapeBuilderResult result = builder.GetBuildResult();

using (Transaction t = new Transaction(doc, "Create tessellated direct shape"))
{
    t.Start();

    DirectShape ds = DirectShape.CreateElement(doc, new ElementId(BuiltInCategory.OST_GenericModel));
    ds.ApplicationId = "Application id";
    ds.ApplicationDataId = "Geometry object id";

    ds.SetShape(result.GetGeometricalObjects());
    t.Commit();
}
```

The image below is the result of running the example above with a concrete material id specified.



BRepBuilder

The BRepBuilder class offers the ability to construct Revit boundary representation geometry (solids, open shells, etc.) as a result of inputs of surfaces, edges, and boundary loops of edges. If the construction of the boundary representation is successful, the resulting geometry objects can be used directly in any other Revit tool that accepts geometry, or the BRepBuilder can be passed directly to populate a DirectShape via the SetShape() and AppendShape() methods of the DirectShape class. Below is an example of using the SetShape() method to assign a cylinder shape to a new DirectShape object.

Code Region: Create a DirectShape using BRepBuilder

```
private void CreateDirectShapeFromCylinder(Document doc)
{
    // Naming convention for faces and edges: we assume that x is to the left
    // and pointing down, y is horizontal and pointing to the right, z is up

    BRepBuilder brepBuilder = new BRepBuilder(BRepType.Solid);
```

```

// The surfaces of the four faces.

Frame basis = new Frame(new XYZ(50, -100, 0), new XYZ(0, 1, 0), new XYZ(-1, 0, 0), new XYZ(0, 0, 1));

CylindricalSurface cylSurf = CylindricalSurface.Create(basis, 50);

Plane top = Plane.CreateByNormalAndOrigin(new XYZ(0, 0, 1), new XYZ(0, 0, 100)); // normal points outside the cylinder

Plane bottom = Plane.CreateByNormalAndOrigin(new XYZ(0, 0, 1), new XYZ(0, 0, 0)); // normal points inside the cylinder

// Add the four faces

BRepBuilderGeometryId frontCylFaceId = brepBuilder.AddFace(BRepBuilderSurfaceGeometry.Create(cylSurf, null), false);

BRepBuilderGeometryId backCylFaceId = brepBuilder.AddFace(BRepBuilderSurfaceGeometry.Create(cylSurf, null), false);

BRepBuilderGeometryId topFaceId = brepBuilder.AddFace(BRepBuilderSurfaceGeometry.Create(top, null), false);

BRepBuilderGeometryId bottomFaceId = brepBuilder.AddFace(BRepBuilderSurfaceGeometry.Create(bottom, null), true);

// Geometry for the four semi-circular edges and two vertical linear edges

BRepBuilderEdgeGeometry frontEdgeBottom = BRepBuilderEdgeGeometry.Create(Arc.Create(new XYZ(0, -100, 0), new XYZ(100, -100, 0), new XYZ(50, -50, 0)));

BRepBuilderEdgeGeometry backEdgeBottom = BRepBuilderEdgeGeometry.Create(Arc.Create(new XYZ(100, -100, 0), new XYZ(0, -100, 0), new XYZ(50, -150, 0)));

BRepBuilderEdgeGeometry frontEdgeTop = BRepBuilderEdgeGeometry.Create(Arc.Create(new XYZ(0, -100, 100), new XYZ(100, -100, 100), new XYZ(50, -50, 100)));

BRepBuilderEdgeGeometry backEdgeTop = BRepBuilderEdgeGeometry.Create(Arc.Create(new XYZ(0, -100, 100), new XYZ(100, -100, 100), new XYZ(50, -150, 100)));

```

```
BRepBuilderEdgeGeometry linearEdgeFront = BRepBuilderEdgeGeometry.Create(new XYZ(100, -100, 0), new XYZ(100, -100, 100));  
BRepBuilderEdgeGeometry linearEdgeBack = BRepBuilderEdgeGeometry.Create(new XYZ(0, -100, 0), new XYZ(0, -100, 100));  
  
// Add the six edges  
BRepBuilderGeometryId frontEdgeBottomId = brepBuilder.AddEdge(frontEdgeBottom);  
BRepBuilderGeometryId frontEdgeTopId = brepBuilder.AddEdge(frontEdgeTop);  
BRepBuilderGeometryId linearEdgeFrontId = brepBuilder.AddEdge(linearEdgeFront);  
BRepBuilderGeometryId linearEdgeBackId = brepBuilder.AddEdge(linearEdgeBack);  
BRepBuilderGeometryId backEdgeBottomId = brepBuilder.AddEdge(backEdgeBottom);  
BRepBuilderGeometryId backEdgeTopId = brepBuilder.AddEdge(backEdgeTop);  
  
// Loops of the four faces  
BRepBuilderGeometryId loopId_Top = brepBuilder.AddLoop(topFaceId);  
BRepBuilderGeometryId loopId_Bottom = brepBuilder.AddLoop(bottomFaceId);  
BRepBuilderGeometryId loopId_Front = brepBuilder.AddLoop(frontCylFaceId);  
BRepBuilderGeometryId loopId_Back = brepBuilder.AddLoop(backCylFaceId);  
  
// Add coedges for the loop of the front face  
brepBuilder.AddCoEdge(loopId_Front, linearEdgeBackId, false);  
brepBuilder.AddCoEdge(loopId_Front, frontEdgeTopId, false);  
brepBuilder.AddCoEdge(loopId_Front, linearEdgeFrontId, true);  
brepBuilder.AddCoEdge(loopId_Front, frontEdgeBottomId, true);
```

```
brepBuilder.FinishLoop(loopId_Front);

brepBuilder.FinishFace(frontCylFaceId);

// Add coedges for the loop of the back face

brepBuilder.AddCoEdge(loopId_Back, linearEdgeBackId, true);
brepBuilder.AddCoEdge(loopId_Back, backEdgeBottomId, true);
brepBuilder.AddCoEdge(loopId_Back, linearEdgeFrontId, false);
brepBuilder.AddCoEdge(loopId_Back, backEdgeTopId, true);
brepBuilder.FinishLoop(loopId_Back);

brepBuilder.FinishFace(backCylFaceId);

// Add coedges for the loop of the top face

brepBuilder.AddCoEdge(loopId_Top, backEdgeTopId, false);
brepBuilder.AddCoEdge(loopId_Top, frontEdgeTopId, true);
brepBuilder.FinishLoop(loopId_Top);
brepBuilder.FinishFace(topFaceId);

// Add coedges for the loop of the bottom face

brepBuilder.AddCoEdge(loopId_Bottom, frontEdgeBottomId, false);
brepBuilder.AddCoEdge(loopId_Bottom, backEdgeBottomId, false);
brepBuilder.FinishLoop(loopId_Bottom);
brepBuilder.FinishFace(bottomFaceId);

brepBuilder.Finish();

using (Transaction tr = new Transaction(doc, "Create a DirectShape"))
```

```

    {

        tr.Start();

        DirectShape ds = DirectShape.CreateElement(doc, new ElementId(BuiltIn
Category.OST_GenericModel));

        ds.SetShape(brepBuilder);

        tr.Commit();

    }

}

```

ShapeImporter

The ShapeImporter utility class supports conversion of geometry stored in external formats (such as SAT and Rhino) into a collection of GeometryObjects which can be used to set the shape for a DirectShape. Use ShapeImporter.Convert() to generate the geometry objects (and where possible, corresponding materials and graphics styles in the associated document).

Code Region: Create a DirectShape from SAT file

```

public void ReadSATFile(Document revitDoc)
{
    // Allow the user to select a SAT file.

    System.Windows.Forms.OpenFileDialog ofd = new System.Windows.Forms.OpenFi
leDialog();

    ofd.Filter = "SAT Files (*.sat)|*.sat";

    if (System.Windows.Forms.DialogResult.OK == ofd.ShowDialog())

    {

        ShapeImporter shapeImporter = new ShapeImporter();

        shapeImporter.InputFormat = ShapeImporterSourceFormat.SAT;

        IList<GeometryObject> shapes = shapeImporter.Convert(revitDoc, ofd.Fi
leName);
    }
}

```

```
    if (shapes.Count != 0)

    {
        using (Transaction tr = new Transaction(revitDoc, "Create a DirectShape"))

        {
            tr.Start();

            DirectShape dsImportedSat = DirectShape.CreateElement(revitDoc, new ElementId(BuiltInCategory.OST_Walls));
            dsImportedSat.SetShape(shapes);

            tr.Commit();
        }
    }
}
```

Reference Curves, Planes, and Points

The methods:

- DirectShape.AddReferenceCurve()
- DirectShape.AddReferencePlane()
- DirectShape.AddReferencePoint()
- DirectShapeType.AddReferenceCurve()
- DirectShapeType.AddReferencePlane()
- DirectShapeType.AddReferencePoint()

enable the creation of reference curves, planes and points inside DirectShape elements. Explicit bounds can be provided for direct shape reference curves and planes. Revit tools that can use named references within families will also be able to select the references inside the DirectShape elements.

The overloads for these methods include an optional `DirectShapeReferenceOptions` input.

Use `DirectShapeReferenceOptions.Name` to set the assigned name for the reference. If the name is specified, it is visible when picking the reference's geometry. Otherwise, the default DirectShape element name is displayed. The validator `DirectShapeReferenceOptions.IsValidReferenceName()` validates the name assigned to `DirectShapeReferenceOptions.Name`.

The validators:

- `DirectShape.IsValidReferenceCurve()`
- `DirectShape.IsValidReferencePlaneBoundingBoxUV()`
- `DirectShapeType.IsValidReferenceCurve()`
- `DirectShapeType.IsValidReferencePlaneBoundingBoxUV()`

validates the inputs needed for specifying a plane or curve explicit reference in the DirectShape.

Options

The `DirectShapeOptions` class is used to control behavior of a `DirectShape` object. Use `DirectShape.SetOptions()` to set the options used by a `DirectShape` object. The `GetOptions()` method will return the `DirectShapeOptions` currently used by the `DirectShape` object.

`DirectShape` elements, by default, support element references, including dimensions, alignments, and face hosting, as well as snapping. This default behavior can be changed using the `DirectShapeOptions.ReferencingOption` property. If it is set to `NotReferenceable`, the geometry may not be used for dimensioning, snapping, alignment, or face-hosting. The element may still be selected by the user for operations which do not reference individual geometry objects.

`DirectShape` elements also support the ability to participate in room boundary calculations, if they are of an appropriate category for room boundary calculations, and if the associated "Room Bounding" parameter is set to true. The property `DirectShapeOptions.RoomBoundingOption` identifies whether the `DirectShape` supports an option for the "Room Bounding" parameter to permit participation in room boundary calculations. The default value is `NotApplicable`, but this will be changed automatically to `SetByParameter` for applicable `DirectShapes`.

Tagging Direct Shape Geometry

References (such as dimensions) to `DirectShape` geometry can be maintained when the geometry changes. To do this, add geometry to the `DirectShape` using the `AddExternallyTaggedGeometry` method. The sample below shows how to create an instance of the `BRepBuilderPersistentIds` class and use `AddSubTag` to build a map of named `ExternalGeometryId`'s and their corresponding `BRepBuilderGeometryId`'s. When the geometry is modified, use `UpdateExternallyTaggedGeometry` and use the same names for the `ExternalGeometryId`'s that you will associate with the new geometry.

The `CreateDirectShape` sample creates a direct shape with a face named "face1". After running this command, manually create a dimension that references this face. Then run `ModifyDirectShape` to create a new face in the direct shape. Because the old and new faces are both named "face1", the dimension will survive the operation and reference the new face. If

you change the *ModifyDirectShape* code to give the face a different name, then the dimension will fail regeneration with the error that "One or more dimension references are or have become invalid.".

You can also provide an external tag to reference geometry. There is a change in behavior for the following methods. These methods will associate an external ID with the added reference object, if the `DirectShapeReferenceOptions` specify one. An exception is thrown if the `DirectShape` or `DirectShapeType` already has reference geometry with the specified external ID.

- `DirectShape.AddReferencePlane()`
- `DirectShape.AddReferencePoint()`
- `DirectShape.AddReferenceCurve()`
- `DirectShapeType.AddReferencePlane()`
- `DirectShapeType.AddReferencePoint()`
- `DirectShapeType.AddReferenceCurve()`

```
public DirectShape CreateDirectShape(Document doc)
{
    DirectShape ds;
    using (Transaction tr = new Transaction(doc, "Create DirectShape"))
    {
        tr.Start();
        ds = DirectShape.CreateElement(doc, new ElementId(BuiltInCategory.OST_GenericModel));
        ds.AddExternallyTaggedGeometry(CreateTriangularFace("face1", 1));
        tr.Commit();
    }
    return ds;
}

public void ModifyDirectShape(DirectShape ds)
{
    using (Transaction tr = new Transaction(ds.Document, "Update DirectShape"))
```

```
{  
    tr.Start();  
  
    ds.UpdateExternallyTaggedGeometry(CreateTriangularFace("face1", 10));  
  
    tr.Commit();  
}  
  
}  
  
  
private ExternallyTaggedBRep CreateTriangularFace(string name, double x)  
{  
  
    BRepBuilder brepBuilder = new BRepBuilder(BRepType.OpenShell);  
  
  
    XYZ pt0 = new XYZ(x, 0, 0);  
    XYZ pt1 = new XYZ(x, 10, 0);  
    XYZ pt2 = new XYZ(x, 10, 10);  
  
  
    BRepBuilderGeometryId faceId = brepBuilder.AddFace(BRepBuilderSurfaceGeometr  
y.Create(  
  
    Plane.CreateByOriginAndBasis(pt0, XYZ.BasisZ, XYZ.BasisY), null), true);  
  
  
    BRepBuilderGeometryId loopId = brepBuilder.AddLoop(faceId);  
  
    brepBuilder.AddCoEdge(loopId, brepBuilder.addEdge(BRepBuilderEdgeGeometry.Cre  
ate(pt0, pt1)), false);  
  
    brepBuilder.AddCoEdge(loopId, brepBuilder.addEdge(BRepBuilderEdgeGeometry.Cre  
ate(pt1, pt2)), false);  
  
    brepBuilder.AddCoEdge(loopId, brepBuilder.addEdge(BRepBuilderEdgeGeometry.Cre  
ate(pt2, pt0)), false);  
  
    brepBuilder.FinishLoop(loopId);  
  
  
    brepBuilder.FinishFace(faceId);  
}
```

```

        brepBuilder.Finish();

        BRepBuilderPersistentIds persistentIds = new BRepBuilderPersistentIds(brepBuilder);

        persistentIds.AddSubTag(new ExternalGeometryId(name), faceId);

        ExternallyTaggedBRep result = brepBuilder.GetResult(new ExternalGeometryId("MyShape"), persistentIds);

        return result;
    }
}

```

3.16 SubElements

Several Revit elements can now contain a subdivision known as a Subelement. Subelements provide a way for parts of an element to behave as though they were real elements without incurring the overhead of adding more full elements to the model.

Many Revit features - for example parameters, schedules, and tags - were designed to operate on Elements. As a result, the Revit code needs to represent objects as Elements for them to participate in those features. This can lead to scalability problems, because every Element adds overhead and adding many Elements may decrease the performance of the model.

An alternative is to use Subelements. An element can expose a set of "Subelements" that it contains, specifying characteristics like their category and parameters, and certain Revit capabilities will treat those Subelements the same as ordinary Elements. For example, a Subelement may contribute geometry to the main element and may be able to be selected independently of its parent Element. It will possibly have its own (settable) type as well as an assigned category which can be different from its parent Element.

In the API, the new Subelement class is used to refer to either an Element or a specific subelement of a given Element. It is typically directly related to a Reference to either the Element or the specific subelement.

4 Discipline-Specific Functionality

4.1 Architecture

This chapter covers API functionality that is specific to the architectural features of Revit:

- Functionality related to rooms (Element.Room, RoomTag, etc.)

4.1.1 Rooms

The following sections cover information about the room class, its parameters, and how to use the room class in the API.

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:

Table 55: Room, Area, and Tags relationship

Element	Class	Category	Boundary	Location	Can Create
Room in Plan View	Room	OST_Rooms	Has if in an enclosed region	LocationPoint	NewRoom() 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

Note: Room.Name is the combination of the room name and room number. For example, for a room whose number is 2 and whose name is "Upstairs Bedroom", Room.Name returns "Upstairs Bedroom 2". Use the ROOM_NAME BuiltInParameter to get the room name.

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.

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 Plan Topology below.

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;

if (null != planCircuit)
{
    using (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);
        }
        else
        {
            // 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;
}

```

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.

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

Retrieving Room Boundaries

The boundary around a room is obtained from the base class method SpatialElement.GetBoundarySegments(). The method returns null when the room is not in an enclosed region or only exists in the schedule. Each room may have several regions, each of which have several segments hence the data is returned in the form of a list of BoundarySegment lists.

The following figure shows a room boundary selected in the Revit UI:

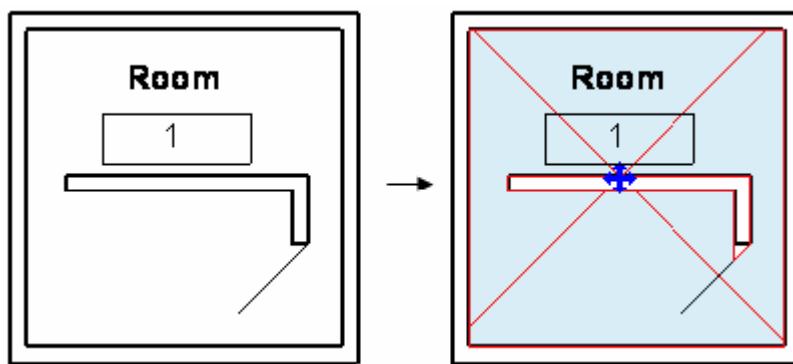


Figure 138: Room boundary

The size of the segment list depends on the enclosed region topology. Each BoundarySegment list 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.

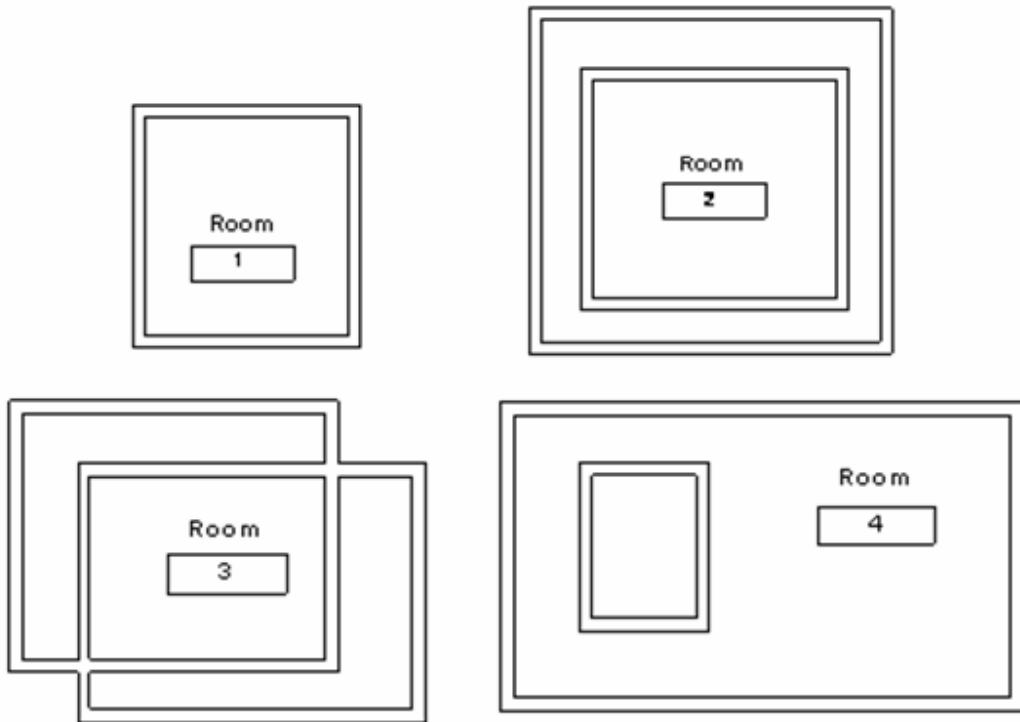


Figure 139: Rooms 1, 2, 3, 4

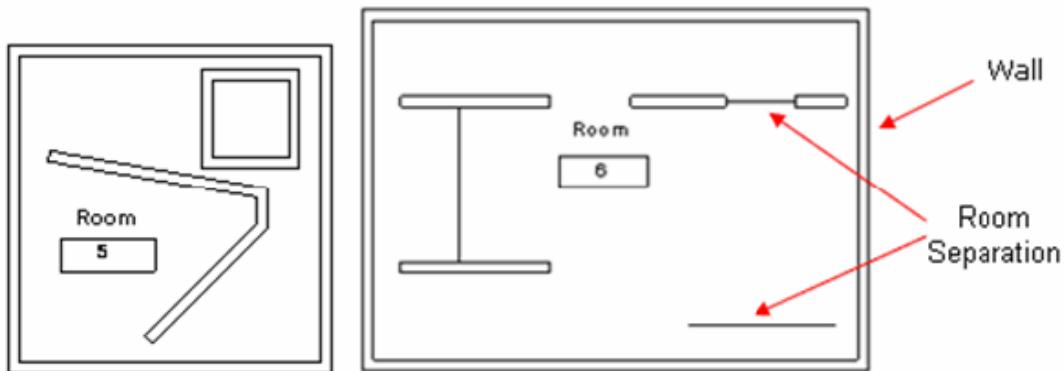


Figure 140: Room 5, 6

The following table provides the `Room.GetBoundarySegments().Size` results for the previous rooms:

Table 56: Room.GetBoundarySegments().Size

Room	<code>Room.GetBoundarySegments().Size</code>
Room 1	1
Room 2	

Room 3

Room 4 2

Room 5 3

Room 6

Note: Walls joined by model lines are considered continuous line segments. Single model lines are ignored.

After getting `IList<IList<BoundarySegment>>`, get the BoundarySegment by iterating the list.

BoundarySegment

The segments that make the region are represented by the BoundarySegment class; its `ElementId` property returns the id of 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.

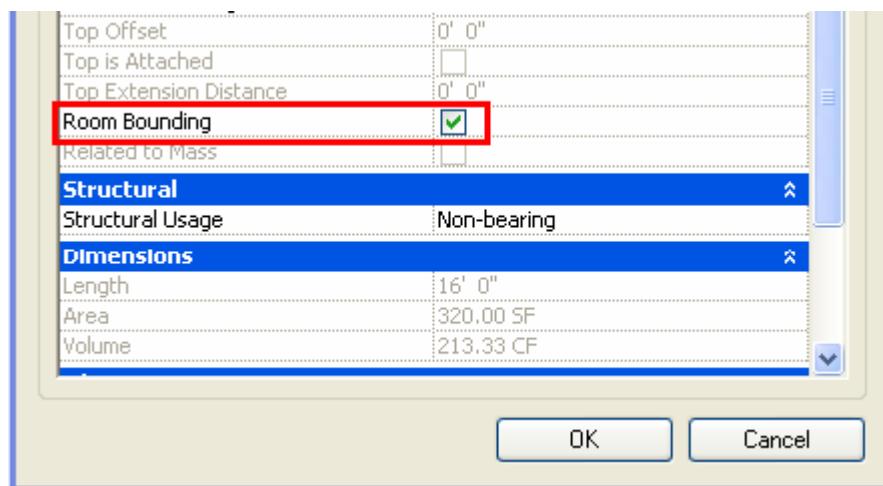


Figure 141: Room Bounding property

Code Region 28-3: Setting room bounding

```
public void SetRoomBounding(Wall wall)
{
}
```

```

Parameter parameter = wall.GetParameter(ParameterTypeId.WallAttrRoomBoundin
g);

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.

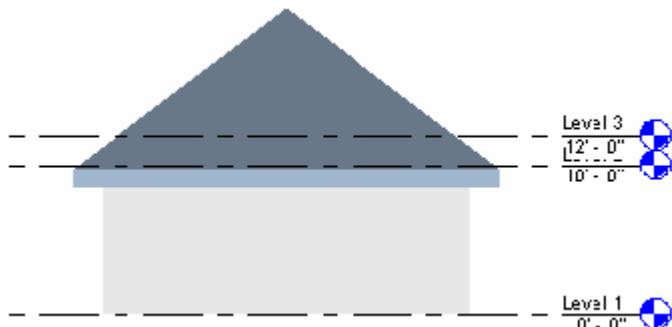


Figure 142: Room created in level 3 view

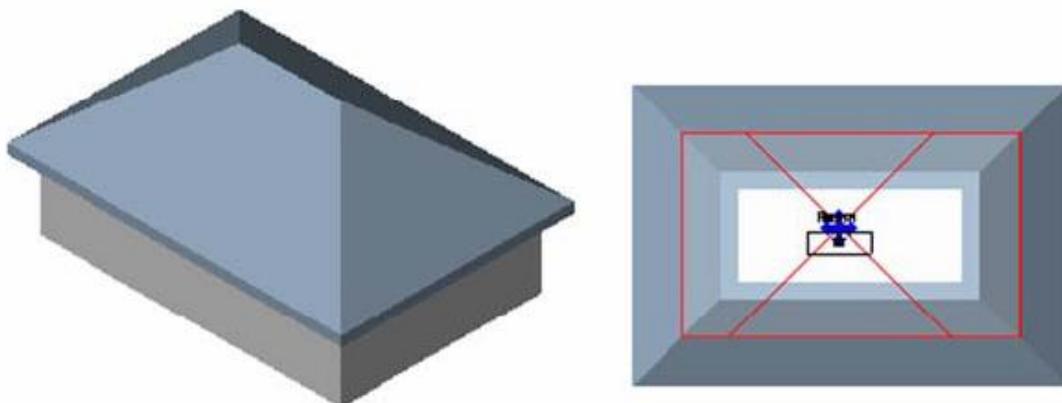


Figure 143: Room boundary formed by roof

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.

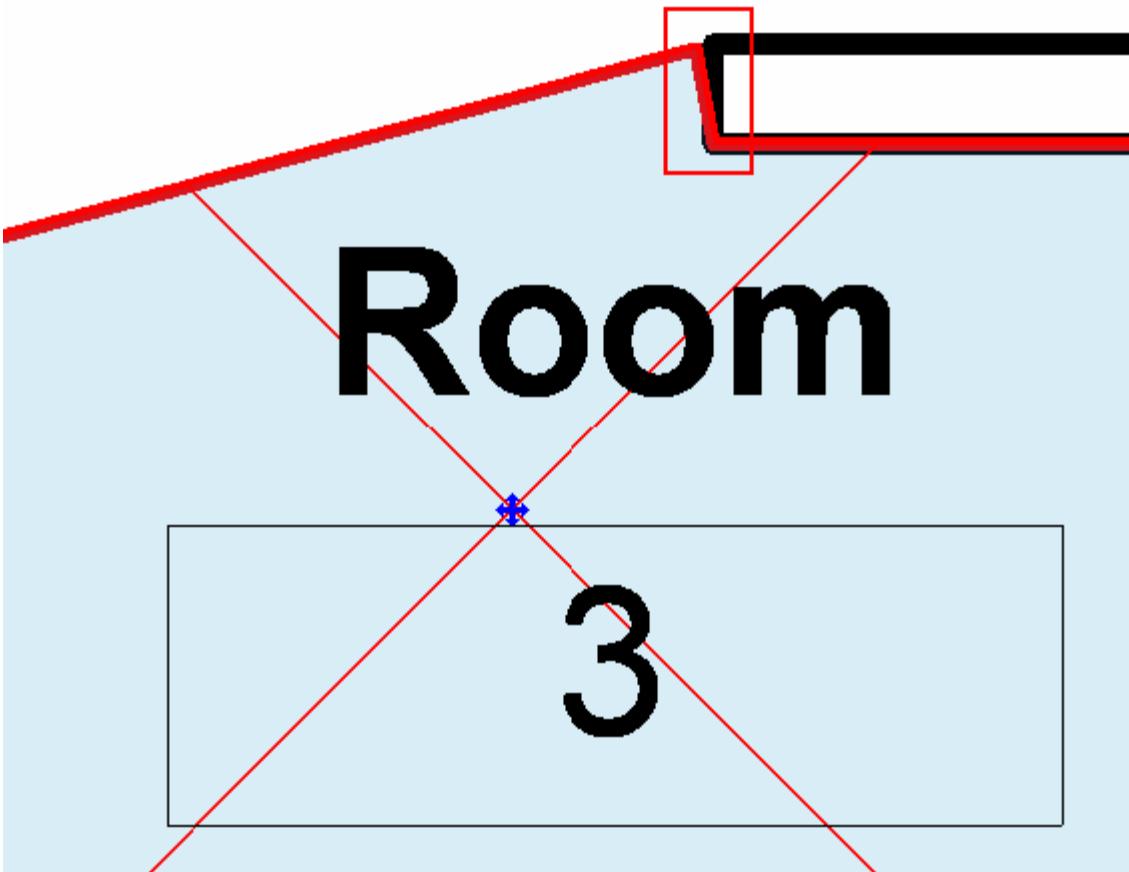


Figure 144: Wall end edge

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.

Boundary and Transaction

When you call `Room.GetBoundarySegments()` 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.

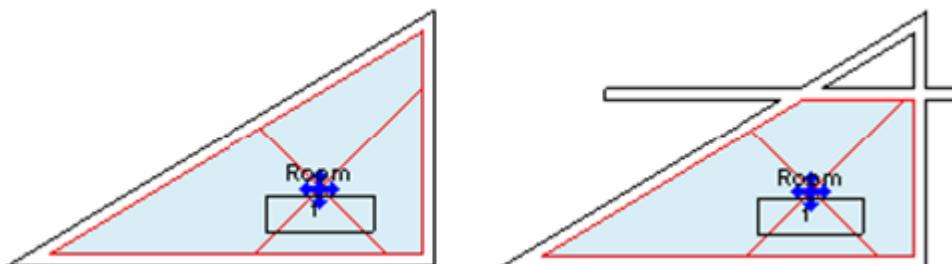


Figure 145: Added wall changes the room boundary

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.GetBoundarySegments(new SpatialElementBoundaryOptions()).First().Count;
    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 = Line.CreateBound(startPos, endPos);

    //Create a new wall and enclose the creating into a single transaction
    using (Transaction transaction = new Transaction(document, "Create Wall"))
    {
        if (transaction.Start() == TransactionStatus.Started)
        {
            Wall wall = Wall.Create(document, line, level.Id, false);
            if (null != wall)
            {
                room.UpdateBoundary();
            }
        }
    }
}
```

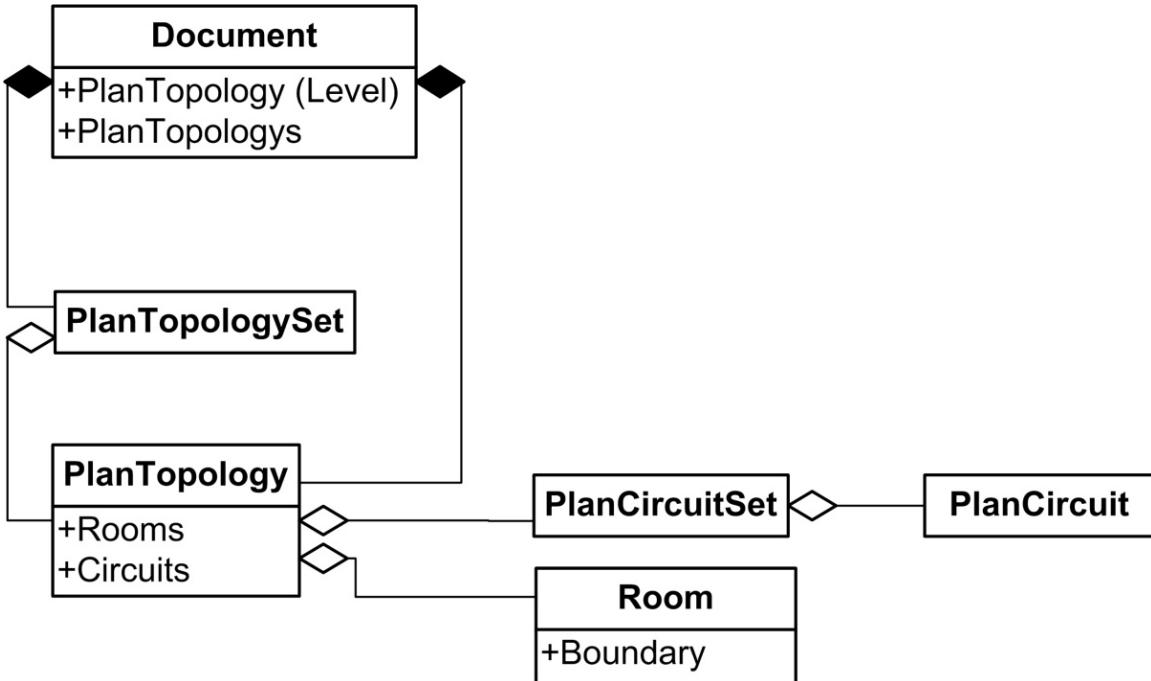
```
        if (TransactionStatus.Committed == transaction.Commit())  
        {  
            //Get the new size  
  
            size = room.GetBoundarySegments(new SpatialElementBoundaryOptions()).First().Count;  
  
            message += "\nRoom boundary size after wall: " + size;  
  
            TaskDialog.Show("Revit", message);  
        }  
    }  
    else  
    {  
        transaction.Rollback();  
    }  
}  
}  
}
```

For more details, see [Transactions](#).

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 146: Room and Plan Topology diagram**

The `PlanCircuit.SideNum` property returns the circuit side number, while `SpatialElement.GetBoundarySegments()` returns an `IList<IList<Autodesk.Revit.DB.BoundarySegment>>`, whose `Count` is different from the circuit side number.

- `SpatialElement.GetBoundarySegments()` 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.

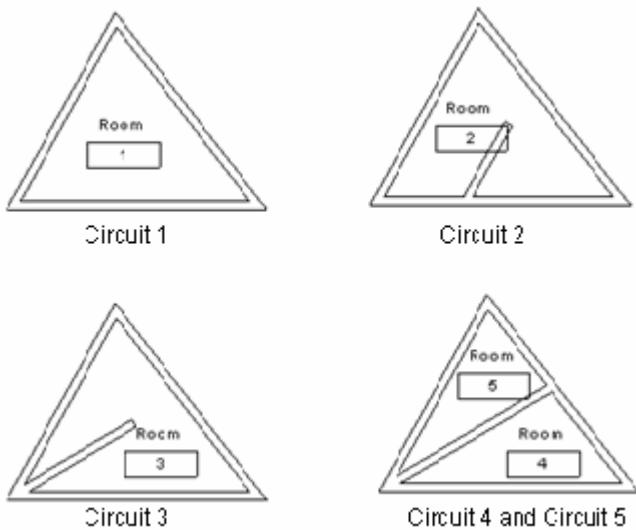
**Figure 147: Compare room boundary with PlanCircuit**

Table 57: Compare Room Boundary with PlanCircuit

Circuit	Circuit.SideNum	<code>IList<IList<Autodesk.Revit.DB.BoundarySegment>>.Count</code> 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)

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.

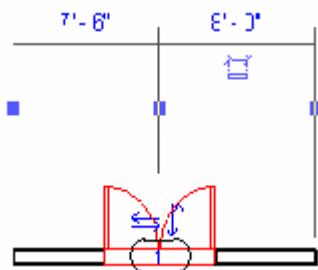


Figure 148: Door 1

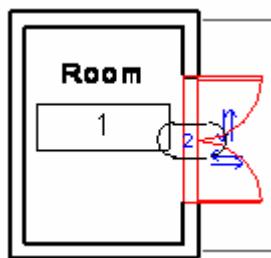


Figure 149: Door 2

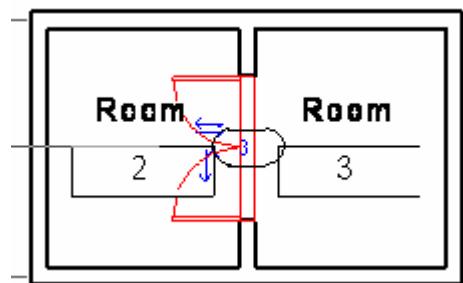


Figure 150: Door 3

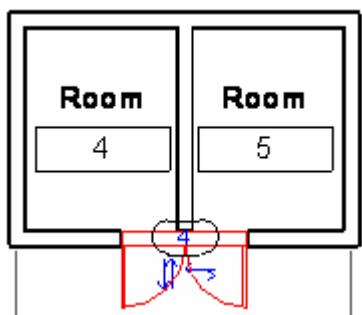


Figure 151: Door 4

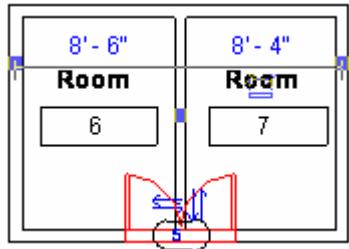


Figure 152: Door 5

Table 58: Door Properties

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

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...
    }
}
```

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:
    using (Transaction t = new Transaction(doc, "Turn on volume calculation"))
    {
        t.Start();
        AreaVolumeSettings settings = AreaVolumeSettings.GetAreaVolumeSetting
s(doc);
        settings.ComputeVolumes = true;
        t.Commit();
    }

    roominfo += "Vol: " + room.Volume + "\n";
    roominfo += "Area: " + room.Area + "\n";
    roominfo += "Perimeter: " + 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.

4.2 Civil Alignments API

Revit provides support for Civil Alignments and their associated annotations. Alignments are imported from InfraWorks as a part of the workflow to transfer Civil Structures. The API supports read of alignment properties and geometric information, along with read/write and create of associated annotations. All new classes for the Alignments API are exposed through a different assembly in the Revit installation, located at Addins\CivilAlignments

\Autodesk.CivilAlignments.DBApplication.dll

`Autodesk.Revit.DB.Infrastructure.Alignment` represents an alignment and can be used to find alignments in a document, and to query a particular alignment's properties and to analyze alignment geometry. This object is not an Element, but the underlying Element can be obtained from this object if needed.

`Autodesk.Revit.DB.Infrastructure.AlignmentStationLabel` represents an alignment station label annotation and can be used to find such labels in a document as well as to create and modify such labels. This object is not an Element, but the underlying Element (which is a `SpotDimension` instance) can be obtained from this object if needed.

- `Autodesk.Revit.DB.Infrastructure.AlignmentStationLabelOptions`
- `Autodesk.Revit.DB.Infrastructure.AlignmentStationLabelSetOptions` provide options for creating a single alignment label or for creating a set of alignment labels.

4.3 MEP Engineering

To support MEP engineering features of the Revit software, the 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
- Access to Mechanical Settings
- Managing Routing Preferences

4.3.1 MEP Element Creation

MEP Elements can be created using the Revit API.

Many elements related to duct, pipe and electrical systems can be created using the following methods available in the Autodesk.Revit.Creation.Document class:

- NewFlexDuct
- NewFlexPipe
- NewMechanicalSystem
- NewPipingSystem
- NewCrossFitting
- NewElbowFitting
- NewTakeoffFitting
- NewTeeFitting
- NewTransitionFitting
- NewUnionFitting

Other MEP elements, such as pipes, can only be created using the static Create() method of its corresponding class. Some MEP elements, such as ducts, can be created by a static method of the corresponding class (i.e. Duct) or by a method of the Autodesk.Revit.Creation.Document class. For these elements, the static Create() method is preferred.

- Duct
- FlexDuct
- Pipe
- FlexPipe
- PipingSystem
- Wire

4.3.1.1 Create 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 using the Pipe.Create() method. New flex pipes, ducts and flex ducts can all be created similarly.

Code Region: Creating a new Pipe using static Create() method

```
public Pipe CreateNewPipe(Document document, ElementId systemTypeId, ElementId levelId)
{
    // 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 = Pipe.Create(document, systemTypeId, pipeType.Id, levelId, p1, p2);
    }

    return pipe;
}
```

{

The code region below demonstrates how to create a FlexPipe using the static FlexPipe.Create() method. Pipes, ducts and flex ducts can all be created between two points similarly.

Code Region: Creating a new FlexPipe using static Create() method

```
public FlexPipe CreateFlexPipe(Document document, Level level)

{
    // find a pipe type

    FilteredElementCollector collector = new FilteredElementCollector(document);

    collector.OfClass(typeof(FlexPipeType));

    ElementId pipeTypeId = collector.FirstElementId();

    // find a pipe system type

    FilteredElementCollector sysCollector = new FilteredElementCollector(document);

    sysCollector.OfClass(typeof(PipingSystemType));

    ElementId pipeSysTypeId = sysCollector.FirstElementId();

    FlexPipe pipe = null;

    if (pipeTypeId != ElementId.InvalidElementId && pipeSysTypeId != ElementId.InvalidElementId)

    {
        // create flex pipe with 3 points

        List<XYZ> points = new List<XYZ>();

        points.Add(new XYZ(0, 0, 0));
```

```
    points.Add(new XYZ(10, 10, 0));

    points.Add(new XYZ(10, 0, 0));

    pipe = FlexPipe.Create(document, pipeSysTypeId, pipeTypeId, level.Id,
points);

}

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: Changing pipe diameter

```
public void ChangePipeSize(Pipe pipe)

{
    Parameter parameter = pipe.GetParameter(ParameterTypeId.RbsPipeDiameterParam);

    string message = "Pipe diameter: " + parameter.AsValueString();

    parameter.Set(0.5); // set to 6

    // Regenerate the document before trying to read a parameter that has been edited

    pipe.Document.Regenerate();

    message += "\nPipe diameter after set: " + parameter.AsValueString();
```

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

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, 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: 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  
    ElementId levelId = ElementId.InvalidElementId;  
    Connector connector1 = null, connector2 = null;  
    ConnectorSetIterator csi = null;  
    ICollection<ElementId> selectedIds = uiDocument.Selection.GetElementIds();  
  
    Document document = uiDocument.Document;  
    // First find the selected equipment and get the correct connector  
    foreach (ElementId id in selectedIds)  
    {  
        Element e = document.GetElement(id);  
        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;

                levelId = family.LevelId;

                break;

            }

        }

    }

}

// next find the second selected item to connect to

foreach (ElementId id in selectedIds)

{

    Element e = document.GetElement(id);

    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 && levelId != ElementId.Inva
lidElementId)

{
```

```

// find a duct type

FilteredElementCollector collector = new FilteredElementCollector(uiD
document.Document);

collector.OfClass(typeof(DuctType));

// Use Linq query to make sure it is one of the rectangular duct type
s

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 = Duct.Create(document, ductType.Id, levelId, connector1, co
nector2);

}

return duct;
}

```

Lining and Insulation

Pipe and duct insulation and lining can be added as separate objects associated with ducts and pipes. The ids of insulation elements associated with a duct or pipe can be retrieved using the static method `InsulationLiningBase.GetInsulationIds()` while the ids of lining elements can be retrieved using the static method `InsulationLiningBase.GetLiningIds()`.

To create new insulations associated with a given duct, pipe, fitting, accessory, or content use the corresponding static method: `DuctInsulation.Create()` or `PipeInsulation.Create()`.

DuctLining.Create() can be used to create a new instance of a lining applied to the inside of a given duct, fitting or accessory.

4.3.1.2 *Creating Wires*

New electrical wires can be created using the Revit API.

The static Wire.Create() allows for new wires to be created in the document. The Create() method requires the id of the view in which the newly created wire will be visible. It must be the id of a floor plan or reflected ceiling plan view. The WiringType for the wire can be either Arc, for wiring that is concealed within walls, ceilings, or floors, or Chamfer, for wiring that is exposed.

The location of the wire is specified by a list of XYZ points defining the vertices of the wire, and optionally a start and/or end connector. The endpoint connectors can be null, however, if the start connector is specified, the connector's origin will be added to the wire's vertices as the start point. Likewise, if the end connector is specified, the connector's origin will be added to the wire's vertices as the end point. The static method Wire.AreVertexPointsValid() will check a list of XYZ points and start and end connectors to ensure they are suitable for a wire.

The shape of the wire is determined by it's wiring type and the total number of points supplied via the vertex points and endpoint connectors. If the wiring type is WiringType.Arc:

- If there are 2 total points supplied, the wire is a straight-line wire.
- If there are 3 total points supplied, the wire is a circular arc wire.
- If there are 4 or more points, the wire is a spline wire.

If the wiring type is WiringType.Chamfer, a polyline wire will be created connecting all the points.

The following example creates a new straight-line wire in the active view with no connectors specified.

Code Region: Creating a new wire

```
public Wire CreateWire(Document document)
{
    Wire wire = null;

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

```

    IList<Element> wireTypes = collector.OfCategory(BuiltInCategory.OST_Wire)
e).WhereElementIsElementType().ToElements();

    WireType wireType = wireTypes.First() as WireType;

    if (wireType != null)
    {

        IList<XYZ> wireVertices = new List<XYZ>();
        wireType.Add(new XYZ(0, 0, 0));
        wireType.Add(new XYZ(2, 0, 0));

        wire = Wire.Create(document, wireType.Id, document.ActiveView.Id, WiringType.Arc, wireType, null, null);
    }

    return wire;
}

```

Connectors

To connect a wire to elements after creation, call `Wire.ConnectTo()`, passing in a start and end connector. If the wire is already connected when this method is used, the old connection will be disconnected and the wire will be connected to the new target.

Vertices

Once a wire is created, vertices can be retrieved using the `Wire.GetVertex()` method. This method takes an index of the requested vertex which should be between 0 and `Wire.NumberOfVertices` (which includes the start and end points of the wire).

Use `Wire.AppendVertex()` to add a vertex to the end of the list, or `Wire.InsertVertex()` to add a vertex at a specific point in the list. The `Wire.IsVertexPointValid()` method checks if the given vertex point can be added to this wire. `IsVertexPointValid()` will return false if the point cannot be added because there is already a vertex at this position on the view plane (within tolerance). Note that a vertex cannot be inserted before the start vertex if the start vertex already connects to an element. Similarly, a vertex cannot be appended to the end of the list if the end point is already connected to an element.

`Wire.RemoveVertex()` will remove a vertex from the list. If the wire vertex is already connected to an element, this method will fail to remove the vertex. In order to remove this vertex, it should be disconnected first, then removed, and then reconnected (if required).

4.3.1.3 *Placeholders*

Placeholder ducts and pipes

The Revit API provides the ability to put placeholder elements into a system when the exact design of the layout is not yet known. Using placeholder ducts and pipes can allow for a well-connected system while the design is still unknown, and then which can then be elaborated in the final design at a later stage.

The two static methods `Duct.CreatePlaceholder()` and `Pipe.CreatePlaceholder()` create placeholder elements. The `IsPlaceholder` property of `Duct` and `Pipe` indicates whether they are a placeholder element or not.

When ready to create actual ducts and pipes from the placeholders, use the `MechanicalUtils.ConvertDuctPlaceholders()` and `PlumbingUtils.ConvertPipePlaceholders()` methods to convert a set of placeholder elements to ducts and pipes. Once conversion succeeds, the placeholder elements are deleted. The new duct, pipe and fitting elements are created and connections are established.

4.3.1.4 *Creating Systems*

Create electrical, mechanical and piping systems.

`MechanicalSystem` and `PipingSystem` have static overloaded `Create()` methods to create new mechanical or piping systems. This is the preferred method for creating new MEP Systems. The simplest `Create()` overload for both classes creates a new system in a given document with a given type Id (which should be the Id for a `DuctSystemType` for a `MechanicalSystem` or the Id of a `PipeSystemType` for a `PipingSystem`). Both classes have a second `Create()` overload that also takes a name for the system. Once created, elements can be added to the system using the `MEPSYSTEM.Add()` method.

`MechanicalSystem` and `PipingSystem` can also be created from the `Creation.Document` class using `NewMechanicalSystem()` and `NewPipingSystem()`. `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()`.

Electrical systems can be created using the `ElectricalSystem.Create` method, which has two overloads. One creates a new `ElectricalSystem` element from an unused `Connector`. The other creates a new `ElectricalSystem` element from a set of electrical components. Both overloads require an `ElectricalSystemType`.

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: Creating a new mechanical system

```
public void CreateSystem(UIDocument uiDocument)
{
    // 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;

    ICollection<ElementId> selectedIds = uiDocument.Selection.GetElementIds();
    Document document = uiDocument.Document;
    foreach (ElementId id in selectedIds)
    {
        Element e = document.GetElement(id);
        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 equipm
ent
            }
        }
    }
}
```

```
        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 = uiDocument.Document.Create.NewMechanicalSystem(baseCo
nnector, connectorSet, DuctSystemType.SupplyAir);

}

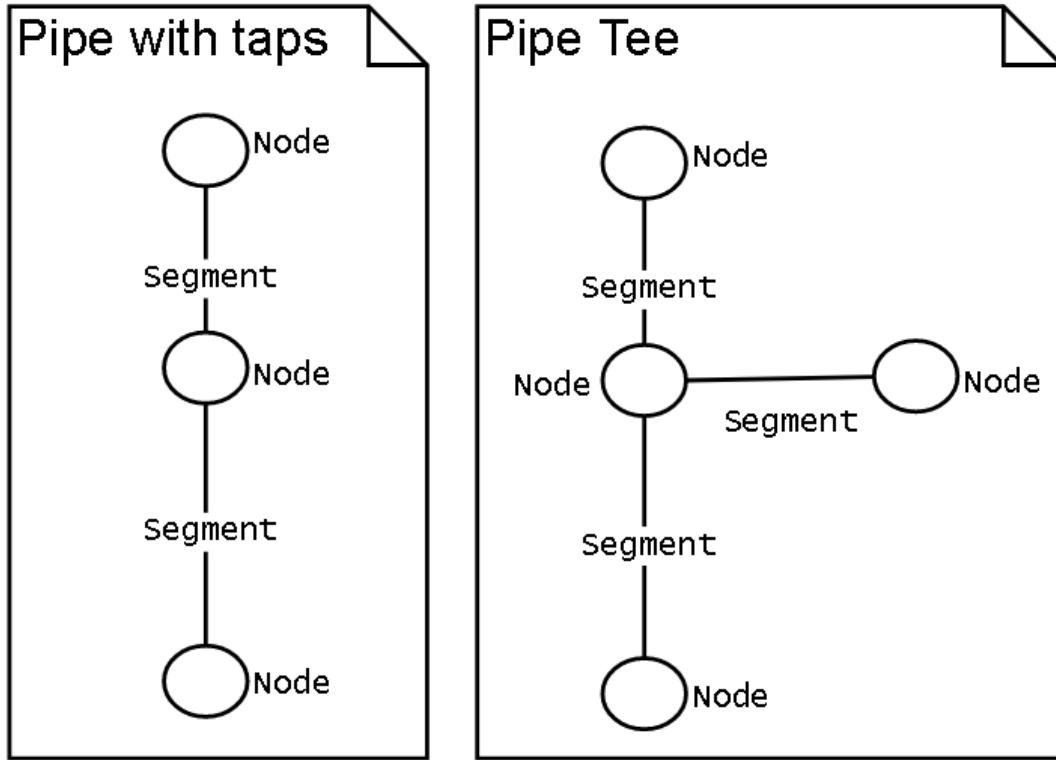
}
```

4.3.2 MEP Analytical Model

The MEP analytical model facilitates network-based flow calculation for piping, fabrication piping, and duct work.

MEPAnalyticalModelCell

By attaching the MEPAnalyticalModelCell to the applicable elements such as Pipe, Duct, FamilyInstance, and FabricationPart, these elements provide a generic data structure for MEP flow and pressure drop analysis, as shown in the diagram below.



Iterating the flow network

While each element is assigned to one `MEPAnalyticalModel` data cell, multiple elements are connected together to form a network. The `MEPNetworkIterator` is used to visit each segment on the network.

For more information, see the SDK sample `NetworkPressureLossReport`.

4.3.3 MEP Systems

`MEPSYSTEM` is the base class for electrical, mechanical and piping systems in Revit MEP.

`ElectricalSystem`, `MechanicalSystem` and `PipingSystem` all derive from the `MEPSYSTEM` class. The base class has some common functionality across system types, such as adding elements to the system or finding the base panel or equipment of the system. A few methods in the base class only apply to HVAC and plumbing systems, such as the `DivideSystem()` method which divides the physical networks in the system and creates a new system for each network.

The derived classes have additional methods and properties specific to the system type.

MEPSection

The `MEPSYSTEM` class has a `SectionsCount` property which returns the number of sections for the system. An `MEPSection` object can be obtained using either the `GetSectionByIndex()`

method or the GetSectionByNumber() method. Although these methods are in the base MEPSYSTEM class, the MEPSection class represents duct and pipe sections and is mainly for pressure loss calculation. It is a series of connected elements (segments - ducts or pipes, fittings, terminals and accessories) which can be obtained from the GetElementIds() method. All section members should have same flow analysis properties: Flow, Size, Velocity, Friction and Roughness.

The segment length, pressure drop and loss coefficient for each element in the section can vary, so methods are available in MEPSection to get these values given a specific element id for an element in the section. The coefficient for ducts is the loss coefficient. For pipes this is the same as the friction factor.

Calculations

Some properties of MEP systems are calculated by Revit. Both MechanicalSystem and PipingSystem have an IsWellConnected property which indicates if the system is well connected or not. If the system is not well connected, parameters which need to be calculated are invalid.

For mechanical and piping systems, some values are calculated based on properties of the sections of the system. The MEPSYSTEM.GetCriticalPathSectionNumbers() method returns a list of the critical path section numbers in order of the direction of flow and PressureLossOfCriticalPath() gets the total pressure loss of the sections in the critical path.

The GetFlow() and GetStaticPressure() methods available from MechanicalSystem and PipingSystem get the flow and static pressure for the system.

PipingSystem has additional calculated properties: GetFixtureUnits() and GetVolume()

Note: Due to the way these calculated properties are handled internally by Revit, they do not support dynamic model update. However, other system properties that are not calculated do support dynamic model update.

4.3.4 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 image shows the two types of physical connectors - end connections and curve connectors.

Connector Creation

Several static methods exist to create connectors. They require a reference to a planar face that will host the connector and, optionally, an edge of that planar face that defines the connector location.

These connectors are created in the family document, and their data is exposed on elements in a project via `FamilyInstance.MEPModel.ConnectorManager`.

- `CreateCableTrayConnector`
- `CreateConduitConnector`
- `CreateDuctConnector`
- `CreateElectricalConnector`
- `CreatePipeConnector`

Connectors on `MEPCurve` and `FabricationPart` are created by default when a new `MEPCurve` element is created and cannot be rehosted. Their data are accessed via `MEPCurve.ConnectorManager` or `FabricationPart.ConnectorManager`.

Connector Modification

The `ConnectorElement.ChangeHostReference` method changes the connector host reference by allowing you to specify a reference to a different plane reference, and optionally to an edge. If only a plane reference is specified, the connector position will be able to subsequently move along the plane. If an edge is also specified, then the connector location is fixed in place.

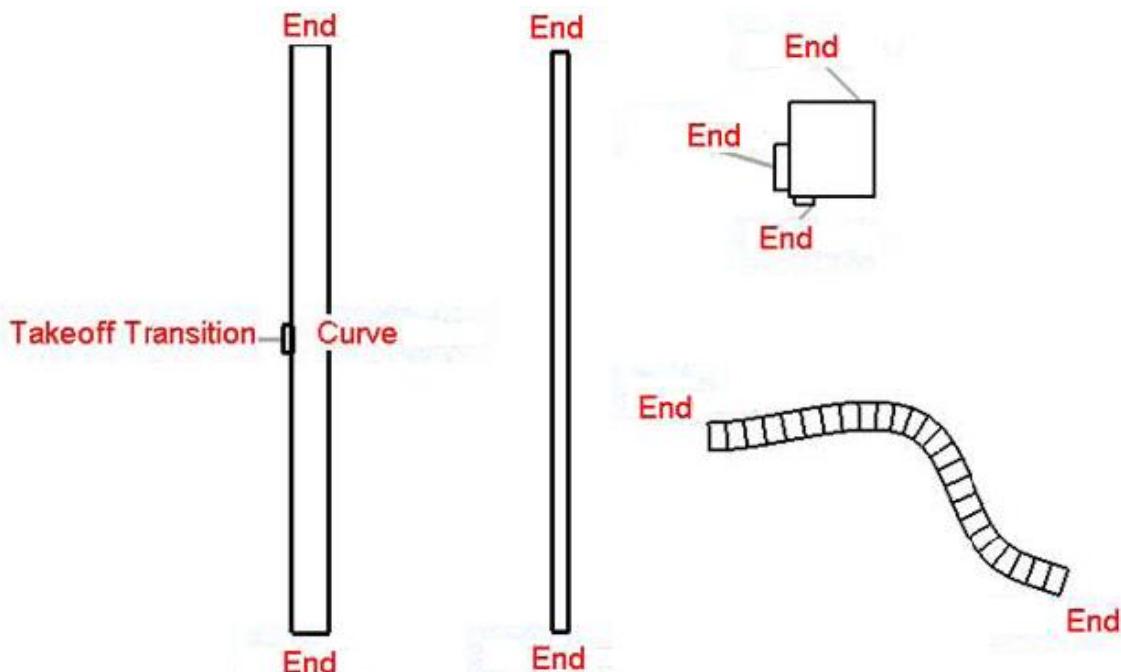


Figure 167: Physical connectors

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(Autodesk.Revit.DB.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.End ||
                        connected.ConnectorType == ConnectorType.Curve ||
                        connected.ConnectorType == ConnectorType.Physical)
                    {
                        message += "\nConnector is connected to: " + connected.Owner.Name;
                        message += "\nConnection type is: " + connected.ConnectorType;
                    }
                }
            }
        }
    }
}
```

```
        }

    }

}

else

{

    message += "\nConnector is not connected to anything.';

}

TaskDialog.Show("Revit", message);

}

}
```

The following dialog box is the result of running this code example on the connector from a piece of plumbing equipment.

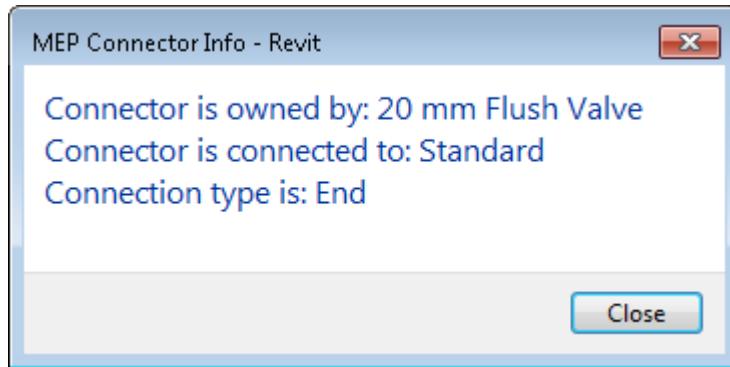


Figure 168: Connector Information

4.3.5 MEP Fabrication Detailing

Several Revit API classes work together to provide the ability to add fabrication components to a Revit document.

Fabrication detailing classes

Before you can place fabrication parts in an MEP model, you must specify a fabrication configuration and load fabrication services into the model. There are a number of classes in the Revit API to facilitate this process including:

- **FabricationConfiguration** - Contains information about the fabrication configuration settings used by the project.
- **FabricationConfigurationInfo** - Contains information about the properties of a FabricationConfiguration.
- **ConfigurationReloadInfo** - Contains results from reloading a FabricationConfiguration.
- **ConnectionValidationInfo** - Contains connection-related warnings generated by reloading a FabricationConfiguration.
- **FabricationService** - Part of the fabrication configuration and defines what FabricationServiceButtons can be used.
- **FabricationServiceButton** - Defines what items to use for different size-based conditions.
- **FabricationPart** - Represents a fabrication component in Revit.
- **FabricationPartType** - Defines the type of a FabricationPart.
- **FabricationRodInfo** - Gives support rod information for a FabricationPart.
- **FabricationHostedInfo** - Contains hosting information for a FabricationPart and provides the ability to disconnect from the host.
- **FabricationConnectorInfo** - Contains information about the connectors of a FabricationPart.
- **FabricationUtils** - General utility methods in the Autodesk Revit product for fabrication.
- **FabricationDimensionDefinition** - Contains information about a fabrication dimension.

The primary classes involved in MEP Fabrication are covered in more detail below. Sample code can be found in the Revit SDK in the FabricationPartLayout sample.

Fabrication configuration

Using the FabricationConfiguration class, users can get and set the fabrication configuration settings for the document. They can also load and unload services, reload the fabrication configuration, get loaded services, get fabrication specifications, get material and insulation information from the configuration, and get connector information.

There is only one fabrication configuration for the document and you can get it using the static FabricationConfiguration.GetFabricationConfiguration() method. To change the configuration, call the overloaded SetConfiguration() method, passing in a FabricationConfigurationInfo object which contains the information about the FabricationConfiguration. One overload of the SetConfiguration() method will set the configuration with the global profile, and one takes a profile name and sets the configuration with that specific profile. (The FabricationConfiguration.GetProfile() method returns the name of the current profile in use and the FabricationConfigurationInfo.GetProfiles() method returns all profile names associated with the configuration.)

The ReloadConfiguration() method reloads the fabrication configuration from its source fabrication configuration. This must be done prior to loading fabrication services.

Use the GetAllLoadedServices() method to get all loaded services or GetAllUsedServices() to get only used fabrication services. A service is in use if any fabrication part in the service is created by the user. Both methods return a list of FabricationService objects.

LoadServices() and UnloadServices() can be used to load and unload a list of fabrication services, respectively.

The FabricationConfiguration class also has methods to get configuration data abbreviations. The GetMaterialAbbreviation() returns the abbreviation of the material or the insulation or the double wall material. GetSpecificationAbbreviation() returns the specification abbreviation for the given specification and GetInsulationSpecificationAbbreviation() will return the abbreviation for the given insulation specification.

Fabrication services

Fabrication services are part of the fabrication configuration and define what fabrication service buttons can be used. The PaletteCount property returns the number of palettes in the service. Using the index of the palettes, you can call GetPaletteName() to get the name of the palette. The method GetButtonCount() will return the number of buttons in a specified palette and the actual buttons can be retrieved by calling GetButton() with a specified palette index and button index.

Fabrication service button

The FabricationServiceButton class contains information about a fabrication button. A fabrication service button defines an item that can be used to define a FabricationPart, possibly subject to a list of size-based specific conditions. Fabrication service buttons are part of a fabrication service.

Fabrication parts

Using the FabricationPart class, users can create, place, move and align fabrication parts in a Revit model. Users can also get or set the dimensions of the fabrication part, and get the fabrication hosted information and support rod information.

The overloaded static FabricationPart.Create() method creates a new fabrication part based on a fabrication service button. The overloaded static CreateHanger() method creates a hanger on another fabrication part. The static AlignPartByConnectors() method will move and align a fabrication part by one of its connectors to align with another connector.

FabricationParts can also be created from design elements using the DesignToFabricationConverter class. The Convert() method will convert a set of MEP design elements into fabrication parts. Successfully created FabricationParts can be obtained using the GetConvertedFabricationParts() methods. If the Convert() method indicates a partial failure, the GetPartialConvertFailureResults() method will return a list of possible failures. For partial failure types (e.g. InvalidConnections) there is a corresponding method of the DesignToFabricationConverter class to retrieve a list of the ElementIds of the FabricationParts with that error type (e.g. GetConvertedFabricationPartsWithInvalidConnections()).

Methods also exist to get and set the value of the fabrication dimension, to get the host element information and to get the fabrication rod

information. `FabricationPart.GetInsulationLiningGeometry()` returns any insulation or liner geometry for a fabrication part. If there is no insulation or liner applied the return value will be null.

Connections and position

`FabricationPart` has a number of methods to attach to connectors or to change the position of the `FabricationPart`. The static `StretchAndFit()` method supports the operation to stretch the fabrication part from the specified connector and fit to the target routing end. The routing end is indicated as a `FabricationPartRouteEnd` object, which can be obtained from the static `FabricationPartRouteEnd.CreateFromConnector()` or `FabricationPartRouteEnd.CreateFromCenterline()` methods. If the `StretchAndFit()` method fails, it will return a `FabricationPartFitResult` enumeration that provides more details on the failure.

Other methods to modify the `FabricationPart` include `Reposition()`, `RotateConnectedPartByConnector()` and `PlaceAsTap()`.

Product lists

Some `FabricationPart` elements, such as purchased duct and pipe fittings, have a "product list". The product list entries represent a catalog of available sizes for the selected part. The `ProductListEntry` property specifies the product list entry of the `FabricationPart`. If the `IsProductList()` method returns false, the `ProductListEntry` will be -1.

To get the list of product entries for the `FabricationPart`, use the `GetProductListEntryCount()` and `GetProductListEntryName()` methods. Prior to changing the `ProductListEntry` for the `FabricationPart`, call `IsProductListEntryCompatibleSize()` to check to see if this part can be changed to the specified product list entry without altering any connected dimensions.

Exporting fabrication job files

`FabricationPart.SaveAsFabricationJob()` writes a fabrication job to disk in the MAJ file format. The exported file will contain the fabrication parts included in the input. It takes an options class, `FabricationSaveJobOptions`, allowing for the possibility of including holes in fabrication parts where branches meet the main straight.

Load and unload of one-off fabrication parts from loose item files

`Autodesk.Revit.DB.FabricationItemFile` contains information about one-off items that can be loaded into a fabrication configuration.

`Autodesk.Revit.DB.FabricationItemFolder` may contain nested `FabricationItemFolders` and a list of `FabricationItemFiles`.

The members:

- `FabricationConfiguration.LoadItemFiles()`
- `FabricationConfiguration.UnloadItemFiles()`
- `FabricationConfiguration.GetAllLoadedItemFiles()`
- `FabricationConfiguration.GetAllUsedItemFiles()`

- `FabricationConfiguration.CanUnloadItemFiles()`
- `FabricationConfiguration.AreItemFilesLoaded()`
- `FabricationConfiguration.GetItemFolders()`

allow control over the loading and unloading of item files into the configuration.

`FabricationPart.Create(Document, FabricationItemFile, ElementId)` creates a `FabricationPart` from an item file.

Version history of parts

`Autodesk.Revit.DB.FabricationVersionInfo` gives the information about different versions of fabrication data, including the reason why the data was changed.

`FabricationPart.GetVersionHistory()` returns a list of `FabricationVersionInfo` classes that describe the history of the changes made to the part. The most recent changes are first in the list.

Part swap out information

`Autodesk.Revit.DB.ReloadSwapOutInfo` gives information about a part that was swapped out during reload.

The members:

- `ConfigurationReloadInfo.OutOfDatePartCount`
- `ConfigurationReloadInfo.GetOutOfDatePartStatus()`

identify the parts that had newer versions found during a reload and which Revit attempted to swap out.

Centerline length

`FabricationPart.CenterlineLength` returns the length of the fabrication part's centerline.

4.3.6 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 overloaded static methods provided by the `ConnectorElement` class are:

- `CreateCableTrayConnector`
- `CreateConduitConnector`
- `CreateDuctConnector`

- CreateElectricalConnector
- CreatePipeConnector

Each of the methods above has a second overload that takes an additional Edge parameter that allows creation of connector elements centered on internal loops of a given face. 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)
    {
        Solid geoSolid = geoObject as Solid;
        if (null != geoSolid)
        {
            foreach (Face geoFace in geoSolid.Faces)
            {
                if (geoFace is PlanarFace)
                {
```

```
    planarFaces.Add(geoFace as PlanarFac  
e);  
  
    }  
  
}  
  
}  
  
}  
  
}  
  
if (planarFaces.Count > 1)  
{  
    // Create the Supply Hydronic pipe connector  
  
    ConnectorElement connSupply = ConnectorElement.CreatePipeConn  
ector(uiDocument.Document,  
  
        PipeSystemType.SupplyHydronic,  
  
        planarFaces[0].Reference);  
  
    Parameter param = connSupply.GetParameter(ParameterTypeId.Conn  
ectorRadius);  
  
    param.Set(1.0); // 1' radius  
  
    param = connSupply.GetParameter(ParameterTypeId.RbsPipeFlowDi  
rectionParam);  
  
    param.Set(2);  
  
    // Create the Return Hydronic pipe connector  
  
    ConnectorElement connReturn = ConnectorElement.CreatePipeCon  
ector(uiDocument.Document,  
  
        PipeSystemType.ReturnHydronic,  
  
        planarFaces[1].Reference);
```

```
        param = connReturn.GetParameter(ParameterTypeId.ConnectorRadius);
        param.Set(0.5); // 6" radius
        param = connReturn.GetParameter(ParameterTypeId.RbsPipeFlowDirectionParam);
        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.

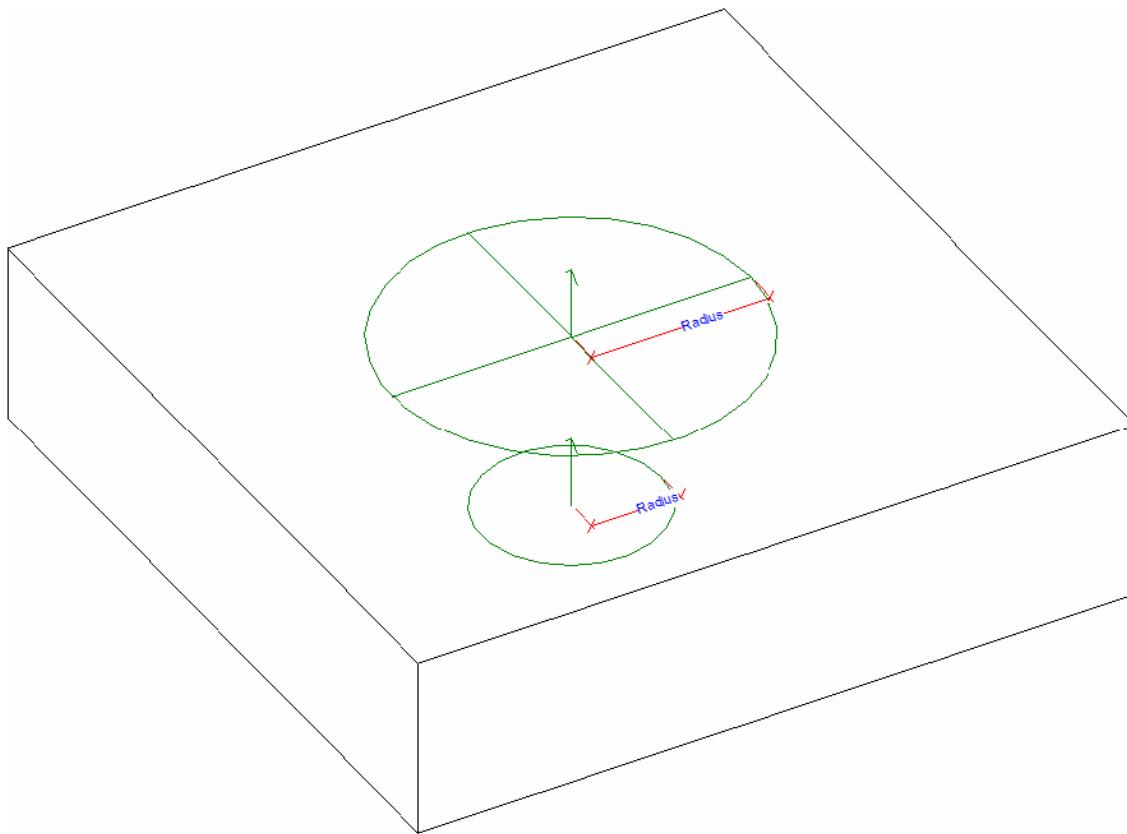
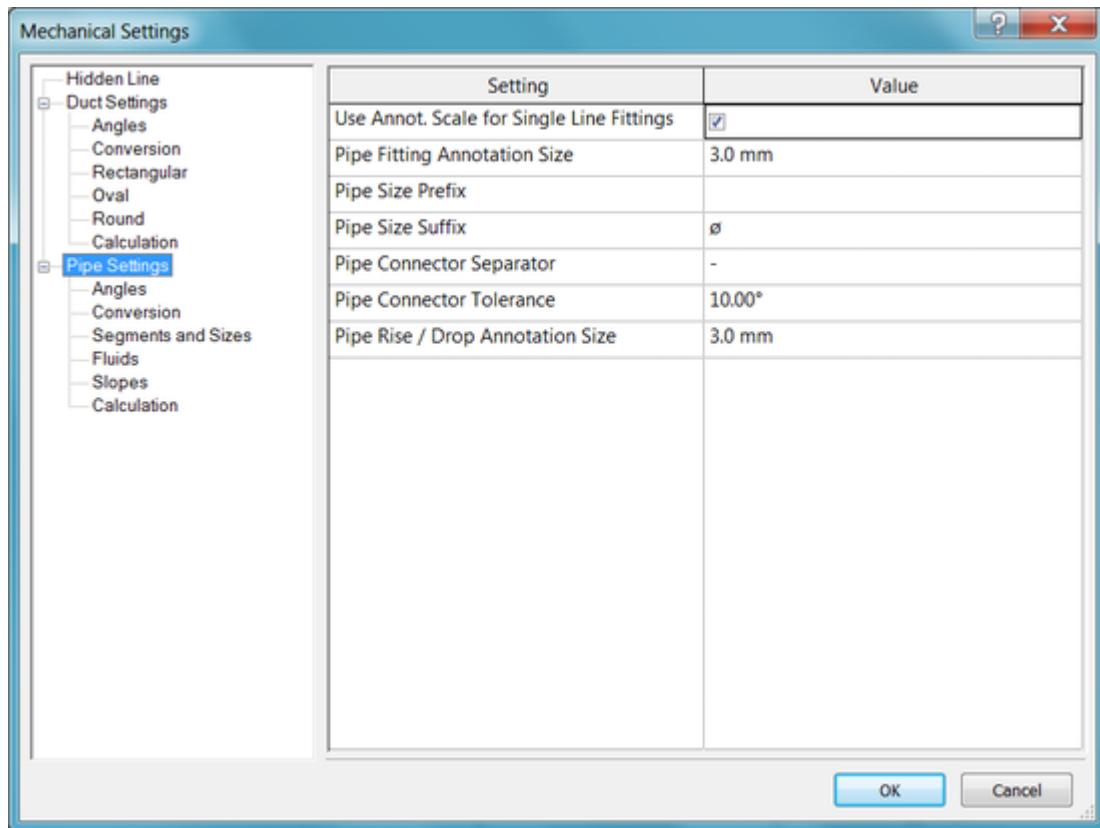


Figure 169: Two connectors created on an extrusion

4.3.7 Mechanical Settings

Many of the settings available on the Manage tab under MEP Settings - Mechanical Settings are also available through the Revit API.

Pipe Settings



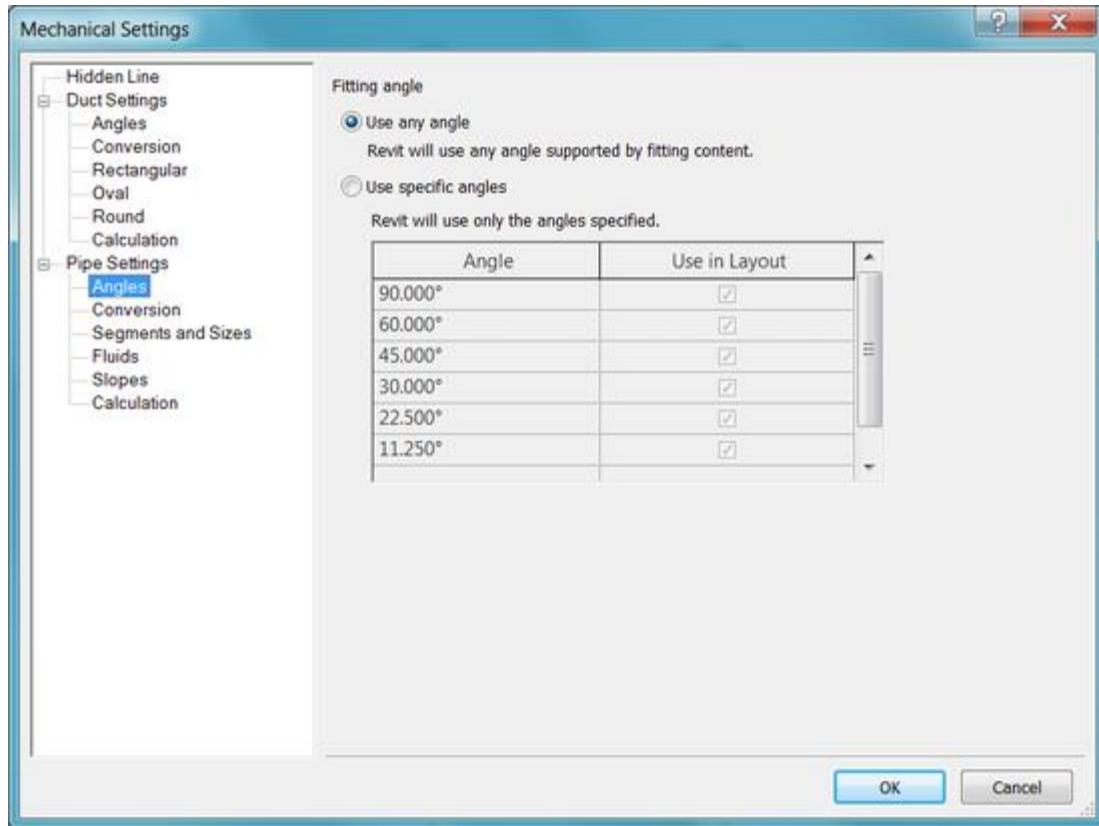
Pipe Settings

The PipeSettings class provides access to the settings shown above, such as Pipe Size Suffix and Pipe Connector Tolerance. There is one PipeSettings object per document and it is accessible through the static method PipeSettings.GetPipeSettings().

Fitting Angles

Fitting angle usage settings for pipes are available from the following property and methods of the PipeSettings class:

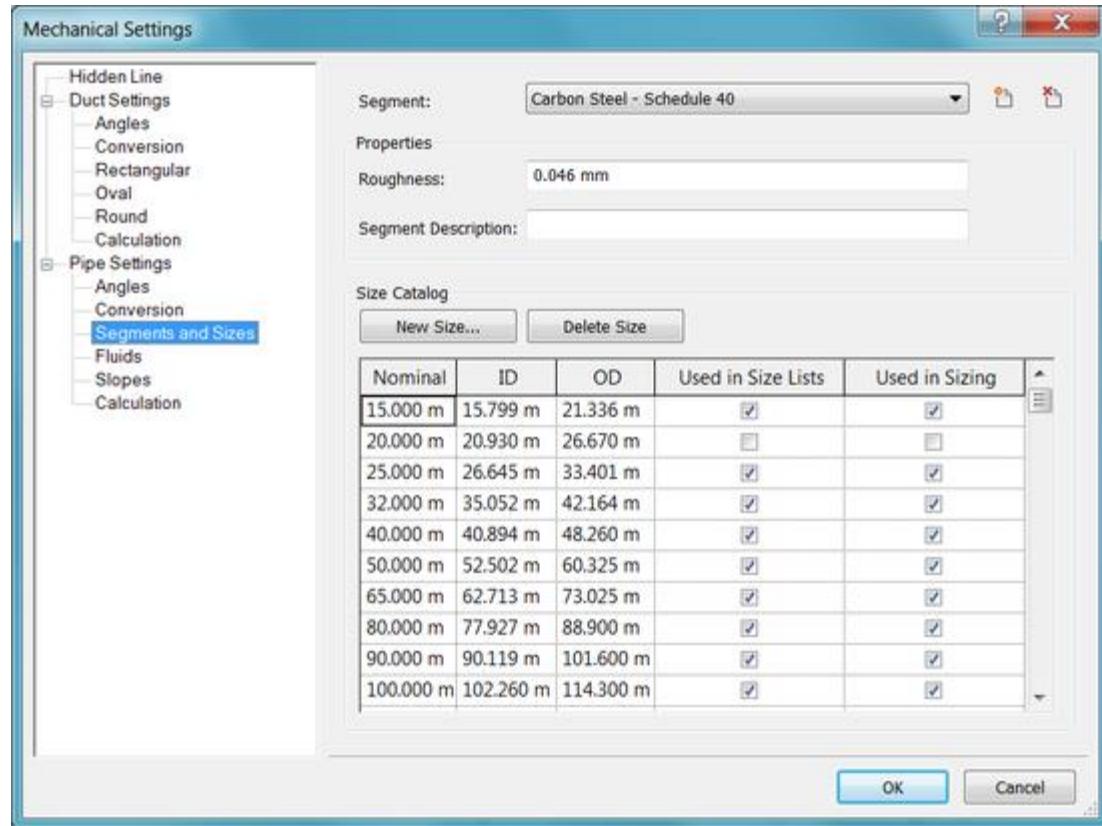
- PipeSettings.FittingAngleUsage
- PipeSettings.GetSpecificFittingAngles()
- PipeSettings.GetSpecificFittingAngleStatus()
- PipeSettings.SetSpecificFittingAngleStatus()



Pipe Fitting Angles

Segments and Sizes

The settings available in the UI under Pipe Settings - Segments and Sizes are available as well.



Segments and Sizes

This information is available through the Segment and MEPSize classes. A Segment represents a length of MEPCurve that contains a material and set of available sizes. The pipe sizes are represented by the MEPSize class. The Segments available can be found using a filter. The following example demonstrates how to get some of the information in the dialog above.

Code Region: Traversing Pipe Sizes in Pipe Settings

```
public void PipeSizes(Document document)
{
    FilteredElementCollector collectorPipeType = new FilteredElementCollector
    (document);

    collectorPipeType.OfClass(typeof(Segment));

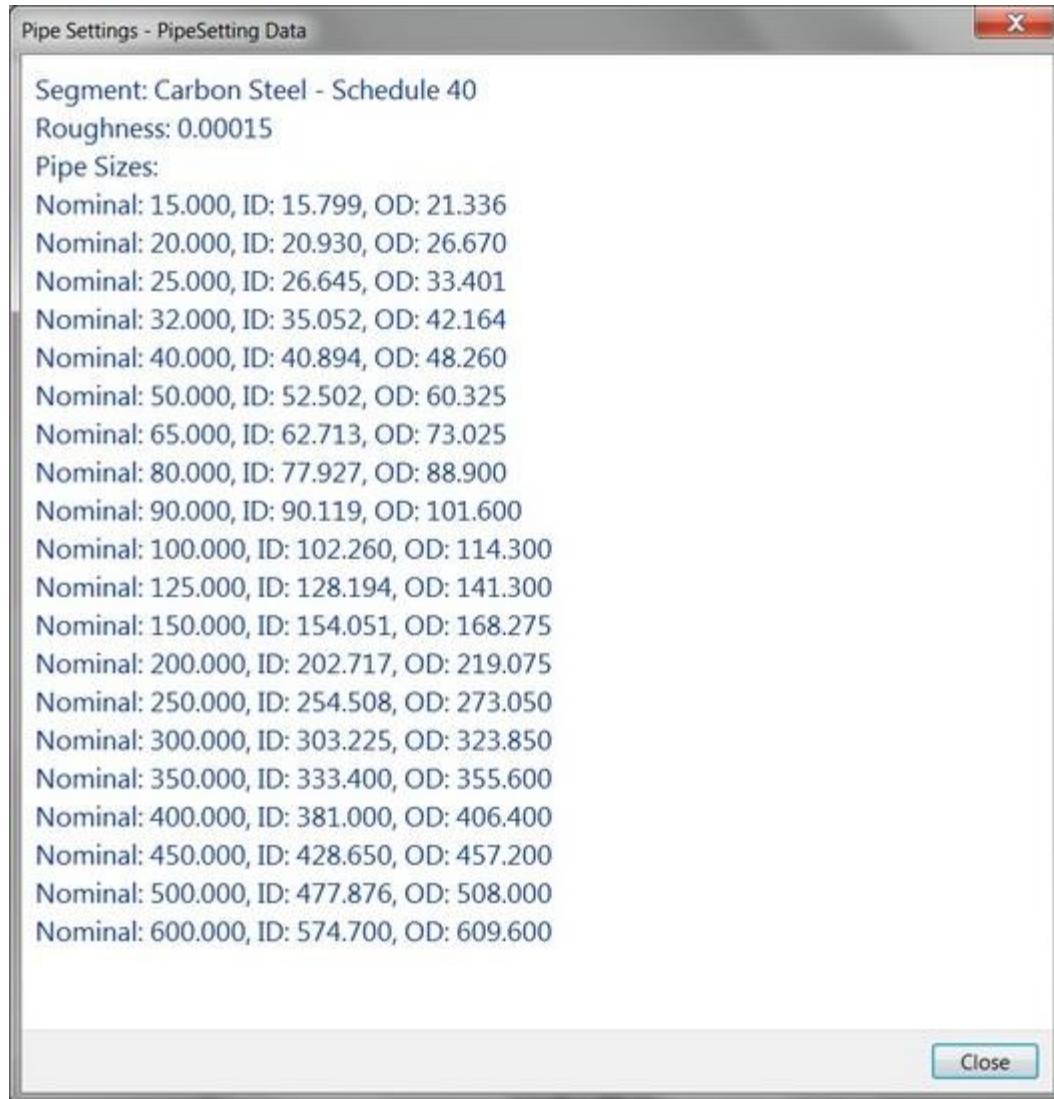
    IEnumerable<Segment> segments = collectorPipeType.ToElements().Cast<Segme
    nt>();
```

```
foreach (Segment segment in segments)
{
    StringBuilder strPipeInfo = new StringBuilder();
    strPipeInfo.AppendLine("Segment: " + segment.Name);

    strPipeInfo.AppendLine("Roughness: " + segment.Roughness);

    strPipeInfo.AppendLine("Pipe Sizes:");
    double dLengthFac = 304.8; // used to convert stored units from ft to mm for display
    foreach (MEPSIZE size in segment.GetSizes())
    {
        strPipeInfo.AppendLine(string.Format("Nominal: {0:F3}, ID: {1:F3}, OD: {2:F3}",
            size.NominalDiameter * dLengthFac, size.InnerDiameter * dLengthFac, size.OuterDiameter * dLengthFac));
    }

    TaskDialog.Show("PipeSetting Data", strPipeInfo.ToString());
    break;
}
```



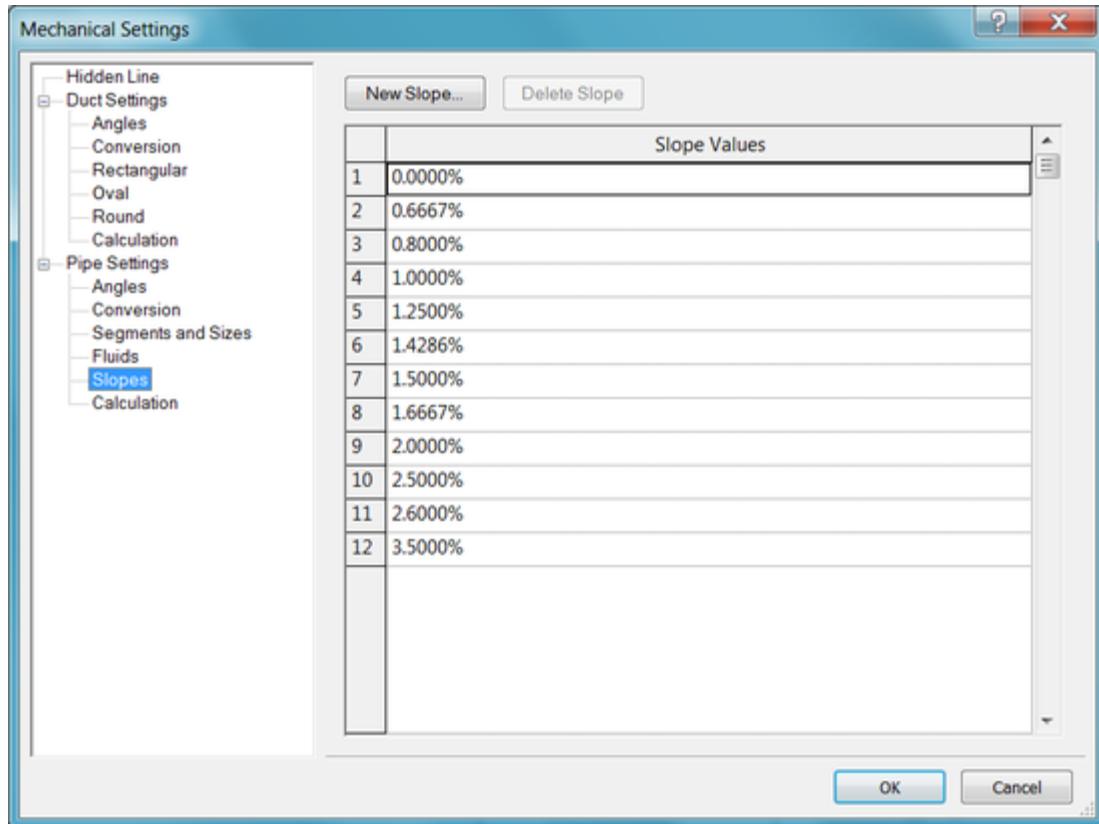
Output of previous example

To add new sizes to the list, use the Segment.AddSize() method. Use Segment.RemoveSize() to remove a size by nominal diameter.

Slopes

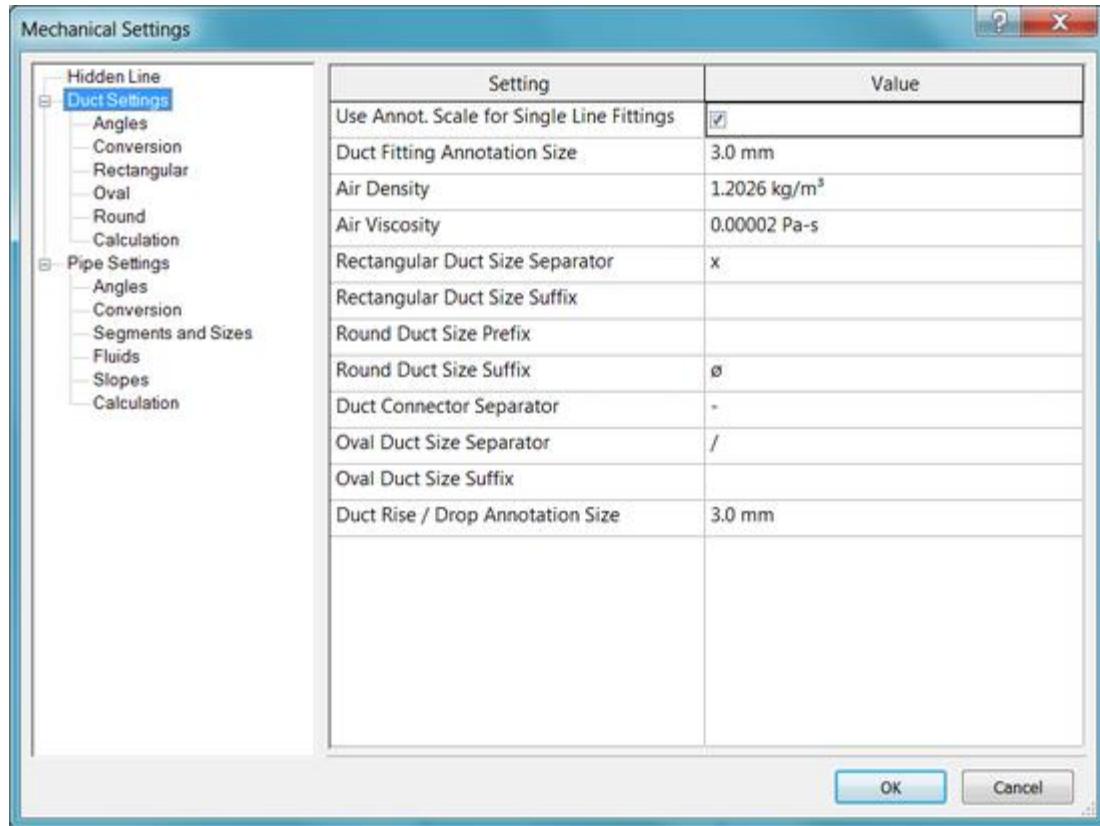
The PipeSettings class also provides access to the slope values available in the UI under Pipe Settings - Slopes. Use GetPipeSlopes() to retrieve a list of slope values.

PipeSettings.SetPipeSlopes() provides the ability to set all the slope values at once, while PipeSettings.AddPipeSlope() adds a single pipe slope. Revit stores the slope value as a percentage (0-100).



Pipe Slope Values

Duct Settings



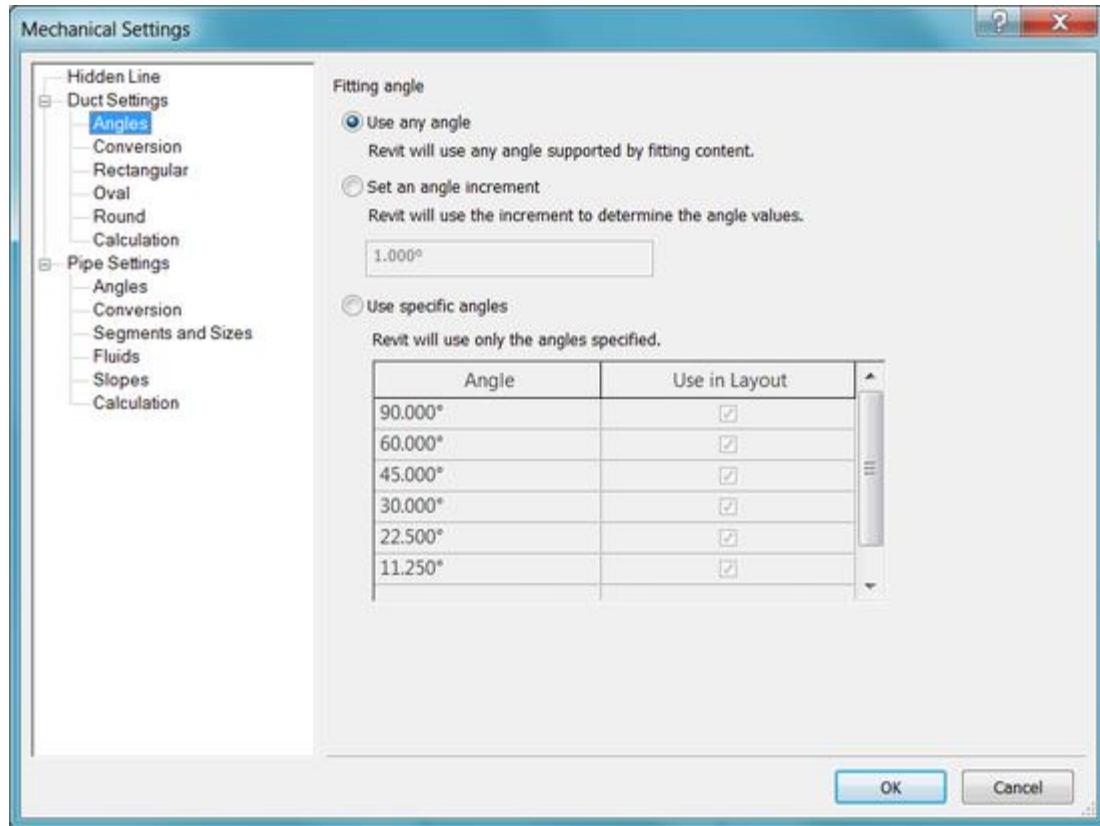
Duct Settings

The `DuctSettings` class provides access to the settings shown above, such as Duct Fitting Annotation Size and Air Density. There is one `DuctSettings` object per document and it is accessible through the static method `DuctSettings.GetDuctSettings()`.

Duct Fitting Angles

Fitting angle usage settings for ducts are available from the following property and methods of the `DuctSettings` class:

- `DuctSettings.FittingAngleUsage`
- `DuctSettings.GetSpecificFittingAngles()`
- `DuctSettings.GetSpecificFittingAngleStatus()`
- `DuctSettings.SetSpecificFittingAngleStatus()`



Duct Fitting Angles

MEP Hidden Line Settings

The `MEPHiddenLineSettings` class represents the settings of the mechanical hidden line display (e.g. ducts and pipes). It can be obtained from the static method: `MEPHiddenLineSettings.GetMEPHiddenLineSettings(Document)`. It offers the following properties:

- `MEPHiddenLineSettings.DrawHiddenLine`
- `MEPHiddenLineSettings.LineStyle`
- `MEPHiddenLineSettings.InsideGap`
- `MEPHiddenLineSettings.OutsideGap`
- `MEPHiddenLineSettings.SingleLineGap`

4.3.8 Electrical Analysis for Preliminary Design

Electrical engineers can estimate the building load throughout the distribution system without placing physical electrical families in the model, enabling conceptual analysis and planning.

Overview

- Create area-based load boundary lines based on one or more lines and a specified level using `CurveElement.CreateAreaBasedLoadBoundaryLine` OR `CurveElement.CreateAreaBasedLoadBoundaryLines`
- For a given level and phase, check if there are any empty plan circuits that do not have electrical load areas with `ElectricalLoadAreaData.HasCircuitsWithoutElectricalLoadAreas`
- If this returns true, create electrical load areas on all the empty plan circuits of the given level using `ElectricalLoadAreaData.CreateElectricalLoadAreas`
- Because electrical load areas are special kind of `SpatialElement`, you can find all existing and any newly created ones using a `FilteredElementCollector`
- Finally, create Area Based Loads by creating a Zone with `Zone.CreateAreaBasedLoad` and a `AreaBasedLoadData` then add the Electrical Load Area to the `AreaBasedLoadData`. Note that one Load Area can be included by more than one Area Based Loads - for example Area Based Load 1 can include Load Area X and Area Based Load 2 can also include Load Area X.

Connections between an `AreaBasedLoadData` object and other nodes can be managed with the methods:

- `ConnectToUpstreamNode(ElementId upstreamNodeId)`
- `DisconnectFromUpstreamNode()`
- `CanDisconnectFromUpstreamNode()`
- `CanConnectToUpstreamNode(ElementId upstreamNodeId)`
- `GetUpstreamNodeId()`

```
public void CreateAreaBasedElectricalLoads(Document doc, List<Curve> lines, Level
level, Phase phase)

{
    IEnumerable<SpatialElement> electricalLoadAreas;

    using (Transaction transaction = new Transaction(doc, "Create Electrical Load
Areas"))

    {
        transaction.Start();

        //Create Area Based Load Boundary Lines.

        CurveElement.CreateAreaBasedLoadBoundaryLines(doc, lines, level.Id);

        if (ElectricalLoadAreaData.HasCircuitsWithoutElectricalLoadAreas(doc, lev
el.Id, phase.Id))

        {
    }
```

```

//Create Electrical Load Areas.

ISet<ElementId> createdElectricalLoadAreaIds = ElectricalLoadAreaData
a.CreateElectricalLoadAreas(doc, level.Id, phase.Id);

}

electricalLoadAreas = new FilteredElementCollector(doc)

.OfClass(typeof(SpatialElement))

.WherePasses(new ElementLevelFilter(level.Id))

.WherePasses(new ElementPhaseStatusFilter(phase.Id, ElementOnPhase
Status.None))

.Cast<SpatialElement>()

.Where(x => x.SpatialElementType == SpatialElementType.Electrical
LoadArea);

transaction.Commit();

}

using (Transaction transaction = new Transaction(doc, "Create Area Based Load
s"))

{
    transaction.Start();

    //Create Area Based Load on each Load Area.

    foreach (SpatialElement electricalLoadArea in electricalLoadAreas)

    {
        Zone areaBasedLoad = Zone.CreateAreaBasedLoad(doc, "AreaBasedLoad1",
level.Id, phase.Id);

        AreaBasedLoadData areaBasedLoadData = areaBasedLoad.GetDomainData() as
AreaBasedLoadData;

        areaBasedLoadData.AddElectricalLoadArea(electricalLoadArea.Id);
    }
}

```

```

        areaBasedLoadData.Voltage = UnitUtils.ConvertToInternalUnits(100, UnitTypeId.Volts);

    }

    transaction.Commit();

}

}

```

Defining Electrical Analytical Loads

When defining loads, you can define area-based loads and point-based equipment loads.

Area-based loads

Area-based loads let you define a closed region and indicate power requirements based on power/area density. For example, lighting in 2nd floor office spaces is 2w/ft², while lighting in conference rooms is 3w/ft², and general power across the entire floor is 3.5 w/ft².

An Area Based Load is composed by one or more Electrical Load Areas. One Electrical Load Area can be included by more than one Area Based Loads. Geometrically, an Electrical Load Area is a minimized enclosed region surrounded by area-based load boundary lines. In the Revit API, Electrical Load Area elements are represented as SpatialElement objects with their property SpatialElementType = ElectricalLoadArea. The SpatialElement.GetSpatialElementDomainData() method gets the SpatialElementDomainData for a given spatial element, and then down-casts it as ElectricalLoadAreaData to GetAreaBasedLoadIds() of it.

The Electrical Load Area elements can be created by [CreateElectricalLoadAreas\(\)](#). This function creates electrical load areas on all the empty plan circuits of the given level.

Point-based equipment loads

Equipment loads allow you to capture load requirements associated with major equipment components, such as elevators, chillers, or any other component beyond the general power density-based loads.

Equipment load elements can be created by ElectricalAnalyticalNode.Create() with the parameter ElectricalAnalyticalNodeType as EquipmentLoad. The [AnalyticalEquipmentLoadData](#) class provides access to this data.

When defining the distribution system, you can create Electrical Analytical Equipment (busses, transformers, transfer switches, and power sources), and interconnect these components. You can then associate area and equipment loads with the distribution system components to sum load throughout the distribution system.

The [AnalyticalEquipmentLoadData.LoadSet](#) property returns the element id of the electrical analytical load set of the analytical equipment load. This [ElectricalAnalyticalLoadSet](#) has members:

- `GetLoadIds()` to get the equipment load ids in the LoadSet
- `QuantityOnStandBy` to get or set the number of Equipment Loads that are not operational at any time
- `TotalQuantity` to get the total count of the equipment loads in the LoadSet

Electrical Analytical Nodes

The `ElectricalAnalyticalNode` class represents an electrical analytical node under the Analytical Power Distribution in the System Browser. The type of this node can be any of the value of the `ElectricalAnalyticalNodeType` enum:

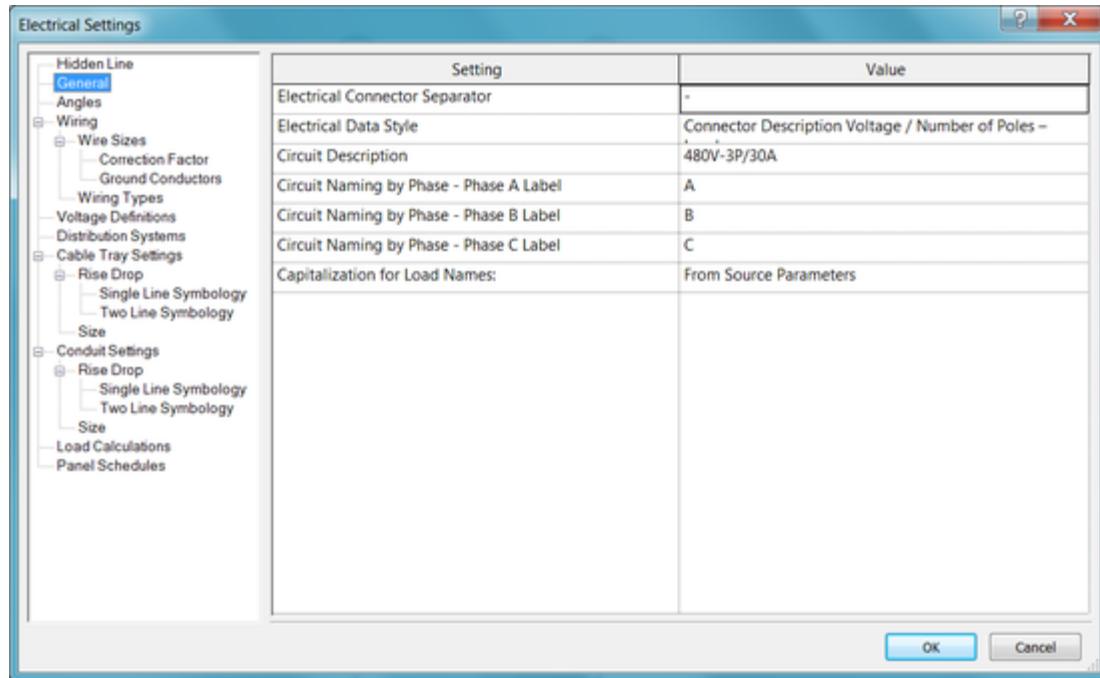
- Power Source
- Bus
- Transformer
- Transfer Switch
- Equipment Load

The static method `ElectricalAnalyticalNode.Create` is used to create these nodes. `GetAllDownstreamLoadIds()` gets all the descendant Electrical Analytical Load ids of the node.

Electrical Analytical Load Sets

4.3.9 Electrical Settings

Some of the settings available on the Manage tab under MEP Settings - Electrical Settings are also available through the Revit API.



Electrical Settings

The `ElectricalSetting` class provides access to different electrical settings, such as fitting angles, wire types, and voltage types. There is one `ElectricalSetting` object per document and it is accessible through the static method `ElectricalSetting.GetElectricalSettings()`.

General Settings

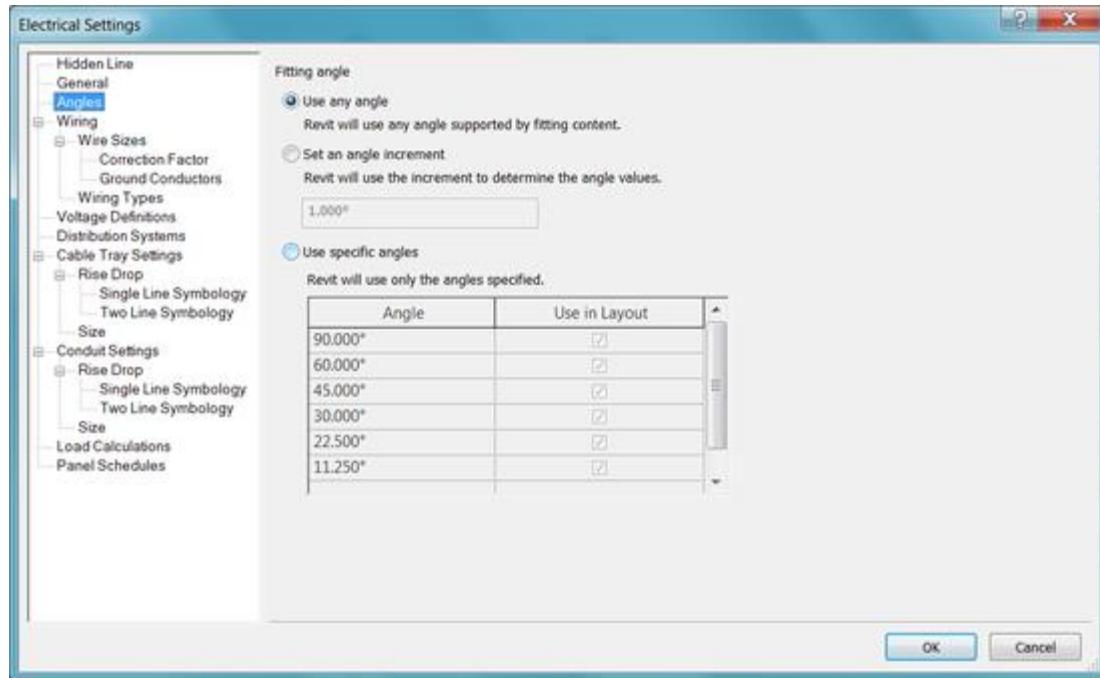
The following general settings are available as properties of the `ElectricalSetting` class:

- `CircuitSequence` - Accesses the circuit sequence numbering schema
- `CircuitNamePhaseA` - Accesses the circuit naming by phase (Phase A Label).
- `CircuitNamePhaseB` - Accesses the circuit naming by phase (Phase B Label).
- `CircuitNamePhaseC` - Accesses the circuit naming by phase (Phase C Label).

Fitting Angles

Fitting angle settings for cable trays and conduits are available from the following methods of the `ElectricalSetting` class:

- `GetSpecificFittingAngles()`
- `GetSpecificFittingAngleStatus()`
- `SetSpecificFittingAngleStatus()`



Fitting Angles

Other Electrical Settings

Properties of the `ElectricalSetting` class provide access to:

- Distribution System Types
- Voltage Types
- Wire Conduit Types
- Wire Material Types
- Wire Types

Methods also are available to add or remove from the project distribution system types, voltate types, wire material types and wire types.

Circuit Naming

`Autodesk.Revit.DB.Electrical.CircuitNamingScheme` represents a scheme used for electrical circuit naming. Significant methods on this class allow schemes to be created and allowing access to the list of combined parameters associated with the scheme:

- `CircuitNamingScheme.Create()`
- `CircuitNamingScheme.GetCombinedParameters()`
- `CircuitNamingScheme.SetCombinedParameters()`

`Autodesk.Revit.DB.Electrical.CircuitNamingSchemeSettings` represents the circuit naming scheme settings object in a project document. Members on this class include:

- `CircuitNamingSchemeSettings.GetCircuitNamingSchemeSettings()`
- `CircuitNamingSchemeSettings.CircuitNamingSchemeld`

4.3.10 Routing Preferences

Routing preferences are accessible through the `RoutingPreferenceManager` class. An instance of this class is available from a property of the `MEPCurveType` class. Currently, only `PipeType` and `DuctType` support routing preferences.

The `RoutingPreferenceManager` manages all rules for selecting segment types and sizes as well as fitting types based on user selection criteria. The `RoutingPreferenceRule` class manages one segment or fitting preference and instances of this class can be added to the `RoutingPreferenceManager`. Each routing preference rule is grouped according to what type of routing item it manages. The type is represented by the `RoutingPreferenceRuleGroupType` and includes these options:

Member name	Description
Undefined	Undefined group type (default initial value)
Segments	Segment types (e.g. pipe stocks)
Elbows	Elbow types
Junctions	Junction types (e.g. takeoff, tee, wye, tap)
Crosses	Cross types
Transitions	Transition types (Note that the multi-shape transitions may have their own groups)
Unions	Union types that connect two segments together
MechanicalJoints	Mechanical joint types that connect fitting to fitting, segment to fitting, or segment to segment
TransitionsRectangularToRound	Multi-shape transition from the rectangular profile to the round profile
TransitionsRectangularToOval	Multi-shape transition from the rectangular profile to the oval profile
TransitionsOvalToRound	Multi-shape transition from the oval profile to the round profile

Each routing preference rule can have one or more selection criteria, represented by the `RoutingCriterionBase` class, and the derived type `PrimarySizeCriterion`. `PrimarySizeCriterion` selects fittings and segments based on minimum and maximum size constraints.

The `RoutingConditions` class holds a collection of `RoutingCondition` instances. The `RoutingCondition` class represents routing information that is used as input when determining if a routing criterion, such as minimum or maximum diameter, is met. The `RoutingPreferencesManager.GetMEPPartId()` method gets a fitting or segment id based on a `RoutingPreferenceRuleGroupType` and `RoutingConditions`.

The following example gets all the pipe types in the document, gets the routing preference manager for each one, then gets the sizes for each segment based on the rules in the routing preference manager.

Code Region: Using Routing Preferences

```
private List<double> GetAvailablePipeSegmentSizesFromDocument(Document document)
{
    System.Collections.Generic.HashSet<double> sizes = new HashSet<double>();

    FilteredElementCollector collectorPipeType = new FilteredElementCollector(document);
    collectorPipeType.OfClass(typeof(PipeType));

    IEnumerable<PipeType> pipeTypes = collectorPipeType.ToElements().Cast<PipeType>();

    foreach (PipeType pipeType in pipeTypes)
    {
        RoutingPreferenceManager rpm = pipeType.RoutingPreferenceManager;

        int segmentCount = rpm.GetNumberOfRules(RoutingPreferenceRuleGroupType.Segments);

        for (int index = 0; index != segmentCount; ++index)
        {
```

```

        RoutingPreferenceRule segmentRule = rpm.GetRule(RoutingPreference
RuleGroupType.Segments, index);

        Segment segment = document.GetElement(segmentRule.MEPPartId) as S
egment;

        foreach (MEPSIZE size in segment.GetSizes())

        {

            sizes.Add(size.NominalDiameter); //Use a hash-set to remove
duplicate sizes among Segments and PipeTypes.

        }

    }

}

List<double> sizesSorted = sizes.ToList();

sizesSorted.Sort();

return sizesSorted;
}

```

4.4 Structural Engineering

The following sections describe API features that only pertain to the structural engineering features of Revit:

- Structural Model Elements - Discusses specific Elements and their properties that only relate to the structural engineering features of Revit.
- AnalyticalModel - Discusses analytical model-related classes such as AnalyticalModel, RigidLink, and AnalyticalModelSupport.
- AnalyticalLink - Discusses creating new analytical links between analytical beams and columns.
- Loads - Discusses Load Settings and three kinds of Loads.
- Your Analysis Link - Provides suggestions for API users who want to link Revit to certain Structural Analysis applications.

This chapter contains some advanced topics. If you are not familiar with the Revit Platform API, read the basic sections first, such as [Getting Started](#), [Elements Essentials](#), [Parameters](#), and so on.

4.4.1 Structural Model Elements

Structural Model Elements are, literally, elements that support a structure such as columns, rebar, trusses, and so on. The following section describe how to manipulate these elements.

The model elements included in this section are specific to the structural engineering features of Revit. For more information about other structural element classes, see the corresponding parts in [Walls](#), [Floors](#), [Ceilings](#), [Roofs](#) and [Openings](#) and [Family Instances](#).

4.4.1.1 *Structural Columns, Beams and Braces*

Structural column, beam, and brace elements are all represented by the FamilyInstance class. 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 by using categories. The category for structural beams and braces is `BuiltInCategory.OST_StructuralFraming`. The category for structural columns is `BuiltInCategory.OST_StructuralColumns`.

Code Region 29-2: Using `BuiltInCategory.OST_StructuralFraming`

```
public void GetBeamAndColumnSymbols(Document document)

{
    List<FamilySymbol> columnTypes = new List<FamilySymbol>();
    List<FamilySymbol> framingTypes = new List<FamilySymbol>();

    FilteredElementCollector collector = new FilteredElementCollector(document);

    ICollection<Element> elements = collector.OfClass(typeof(Family)).ToList();

    foreach(Element element in elements)
    {
        Family family = element as Family;
        Category category = family.FamilyCategory;
        if (null != category)
```

```
{  
    ISet<ElementId> familySymbolIds = family.GetFamilySymbolIds();  
  
    if ((int)BuiltInCategory.OST_StructuralColumns == category.Id.Value)  
    {  
        foreach (ElementId id in familySymbolIds)  
        {  
            FamilySymbol symbol = family.Document.GetElement(id) as FamilySymbol;  
            columnTypes.Add(symbol);  
        }  
    }  
  
    else if ((int)BuiltInCategory.OST_StructuralFraming == category.Id.Value)  
    {  
        foreach (ElementId id in familySymbolIds)  
        {  
            FamilySymbol symbol = family.Document.GetElement(id) as FamilySymbol;  
            framingTypes.Add(symbol);  
        }  
    }  
  
}  
  
string message = "Column Types: ";  
foreach (FamilySymbol familySymbol in columnTypes)  
{
```

```
    message += "\n" + familySymbol.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.

BeamSystem

BeamSystem provides full access and edit ability to beam systems. 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.

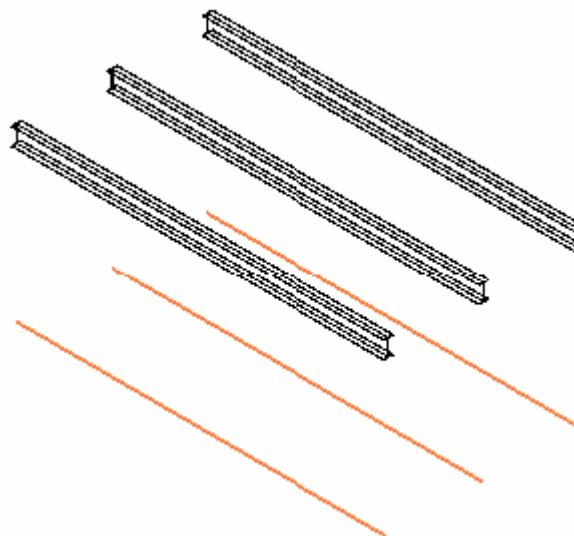


Figure 156: Change BeamSystem elevation

4.4.1.2 Trusses

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;

    using (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);

            XYZ zDirection = new XYZ(0, 0, 1);

            Plane plane = Plane.Create(new Frame(origin, xDirection, yDirection, zDirection));

            SketchPlane sketchPlane = SketchPlane.Create (document, plane);

            //new base Line - use line that spans two selected columns

            AnalyticalMember frame1 = (AnalyticalMember)document.GetElement(A
nalyticalToPhysicalAssociationManager.GetAnalyticalToPhysicalAssociationManag
er(document).GetAssociatedElementId(column1.Id));

            XYZ centerPoint1 = (frame1.GetCurve() as Line).GetEndPoint(0);

            AnalyticalMember frame2 = (AnalyticalMember)document.GetElement(A
nalyticalToPhysicalAssociationManager.GetAnalyticalToPhysicalAssociationManag
er(document).GetAssociatedElementId(column2.Id));
```

```
XYZ centerPoint2 = (frame2.GetCurve() as Line).GetEndPoint(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 = Line.CreateBound(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 trus
s's base line

Autodesk.Revit.DB.View view = document.ActiveView;

// Get a truss type for the truss

FilteredElementCollector collector = new FilteredElementCollector
(document);

collector.OfClass(typeof(FamilySymbol));
collector.OfCategory(BuiltInCategory.OST_Truss);
```

```

    TrussType trussType = collector.FirstElement() as TrussType;

    if (null != trussType)
    {
        truss = Truss.Create(document, trussType.Id, sketchPlane.Id,
baseLine);
        transaction.Commit();
    }
    else
    {
        transaction.Rollback();
        throw new Exception("No truss types found in document.");
    }
}

return truss;
}

```

4.4.1.3 Reinforcement

The Revit API provides classes to manage reinforcement, such as rebar or fabric sheets, in valid hosts such as concrete columns, beams, walls, foundations, and structural floors.

4.4.1.3.1 Rebar

The Rebar class represents rebar used to reinforce suitable elements, such as concrete beams, columns, slabs or foundations.

Shape Driven and Free Form Rebar

There are two kind of rebar – Shape Driven and Free Form. The Shape Driven Rebar is a Rebar the is driven by a shape. The Free Form rebar is driven by curves. Free Form Rebar can be constructed in two ways: from curves (and it will not have constraint) and the second option with

a server that will allow the rebar to have constraints. The server will also allow the API user the possibility to implement how the rebar curves are calculated based on the constraints. This allows API developers to create their own kind of Rebar.

Distribution Path

RebarHandlePositionData.GetDistributionPath() gets the distribution path currently stored in the rebar.

For a free form rebar set the distance between two consecutive bars may be different if it is calculated between different points on bars. The distribution path is an array of curves with the property that based on these curves the set was calculated to respect the layout rule and number of bars or spacing.

Creating rebar

Free Form rebar can be created with the Rebar.CreateFreeForm method. Shape Driven rebar objects can be created using these static Rebar methods. The method `RebarHostData.IsReferenceContainedByAValidHost()` identifies if an element that contains the given reference can host reinforcement.

Name	Description
<pre>public static Rebar Rebar.CreateFromCurv es(Document doc, RebarStyle style, RebarBarType rebarType, RebarHookType startHook, RebarHookType endHook, Element host, XYZ norm, IList<Curve> curves, RebarHookOrientation startHookOr ient, RebarHookOrientation endHookOrie nt,</pre>	Creates a new instance of a Rebar element within the project. All curves must belong to the plane defined by the normal and origin.

```

    bool useExistingShapeIfPossible,
    bool createNewShape
);

```

```

public static Rebar Rebar.CreateFromRebarShape(
    Document doc,
    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.

```

public static Rebar Rebar.CreateFromCurvesAndShape(
    Document doc,
    RebarShape rebarShape,
    RebarBarType rebarType,
    RebarHookType startHook,
    RebarHookType endHook,
    Element host,
    XYZ norm,
    IList<Curve> curves,
);

```

Creates a new instance of a Rebar element within the project. The instance will have the default shape parameters from the RebarShape. All curves must belong to the plane defined by the normal and origin.

```

        RebarHookOrientation startHookOr
ient,
        RebarHookOrientation endHookOrie
nt
);

```

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. The third version creates rebar from an array of curves and based on a RebarShape.

When using the CreateFromCurves() or CreateFromCurvesAndShape() method, the parameters RebarBarType and RebarHookType are available in the RebarBarTypes and RebarHookTypes properties of the Document.

The following code illustrates how to create Rebar with a specific layout.

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);
    // create rebar 9' long
    XYZ rebarLineEnd = new XYZ(origin.X, origin.Y, origin.Z + 9);
    Line rebarLine = Line.CreateBound(origin, rebarLineEnd);

    // Create the line rebar
    IList<Curve> curves = new List<Curve>();
    curves.Add(rebarLine);
}

```

```

    Rebar rebar = Rebar.CreateFromCurves(document, 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 as fixed number, with 10 bars, distribution path length of 1.5'
        // with bars of the bar set on the same side of the rebar plane as indicated by normal
        // and both first and last bar in the set are shown

        rebar.GetShapeDrivenAccessor().SetLayoutAsFixedNumber(10, 1.5, true, true, true);
    }

    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:

Rebar Layout Rule

Value	0	1	2	3	4
Description	Single	Fixed Number	Maximum Spacing	Number with Spacing	Minimum Clear Spacing

Rebar.GetShapeDrivenAccessor().ScaleToBox() takes the shape and scales it to fit the box defined by an origin and two vectors that define the edges of a rectangle.

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.

Free Form Rebar Space Event

The API allows you to control the behavior of free form rebar when the user pushes the <space> key by using these methods:

- `RebarFreeFormAccessor.CycleCounter()` - This property Identifies the cycle counter. It can be zero or a positive number. Its value is changed when the free form Rebar element is selected and the user press Space key, or-through the setter of this property or- by the server if it considers that the counter reaches the maximum value and reset it (set it to 0). property can be accessed just for Rebars that are controlled by a server.
- `RebarUpdateCurvesData.CycleCounterChanged()` - True if the cycle counter was changed, false otherwise. The cycle counter value is changed when the free form Rebar element is selected and the user press Space key -or- by through `RebarRebarFreeFormAccessor.CycleCounter` property or- by the server if it considers that the counter reaches the maximum value and reset it (set it to 0).
- `RebarUpdateCurvesData.GetCycleCounter()` - Gets the cycle counter that is stored in the rebar.
- `RebarUpdateCurvesData.SetCycleCounter()` - Sets the cycle counter to a specific value during the execution of the server. The actual value will be set into the Rebar element after the free form server curve computation is completed.

Aligned Free form Rebar

Aligned Free form rebar can be aligned to the distribution path, to keep the bars vertical, parallel, or perpendicular to a face.

- `RebarFreeFormAccessor.hasValidAlignedServer()` - Returns true if the current rebar is created with the Aligned Free Form rebar server, false otherwise.
- `RebarFreeFormAccessor.AlignedFreeFormSetOrientationOptions()` - Orientation options for an Aligned Free Form Rebar set
- `RebarFreeFormAccessor.AlignedFreeFormSetOrientationOptions` - The enum has the following values.
 - `AlignedToDistributionPath` - Cutting planes are perpendicular to the distribution path.
 - `Vertical` - Cutting planes are oriented vertically, X or Y vector being Z axis.
 - `ParallelToFace` - Cutting planes are parallel to a selected planar face.
 - `PerpendicularToFace` - Cutting planes are oriented perpendicular to a selected planar face, X or Y vector being the face normal.

Copying / Propagating Rebar to a new host

Methods in the `RebarPropagation` class allow rebar to be copied from one host element to a different host element with similar geometry. It copies the source rebars, aligns them to the destination face based on their alignment with the source face, and adapts them to destination host. The propagation is done based on a destination host element where the new rebar will be hosted or a destination face.

Moving individual rebar in a Rebar Set

The methods `Rebar.MoveBarInSet()`, `Rebar.GetBarIndexFromReference()`, `Rebar.GetMovedBarTransform()`, and `Rebar.ResetMovedBarTransform()` allow an application to move an individual bar and to read and reset the transform of the individual bar in a Rebar Set.

Removing individual bars from a Rebar Set

`Rebar.SetBarIncluded(bool, int)` allows you to designate that the bar at a given index will be included or excluded. Setting `Rebar.IncludeFirstBar(bool)` and `Rebar.IncludeLastBar(bool)` are equivalent to removing the bar at the first or last position index.

`Rebar.DoesBarExistAtPosition(int)` can be used to find if the bar at any position index is included or excluded.

Numbering

Rebar is one of the categories of elements whose numbering can be controlled via the Revit API. The `NumberingSchema` and `NumberingSchemaType` classes can be used to define how rebar elements are to be organized for the purpose of numbering/tagging them. Each `NumberingSchema` controls numbering of elements of one particular kind. Instances of `NumberingSchema` are also elements and there is always only one of each type in every Revit document. Available types of all built-in numbering schemas are enumerated in `NumberingSchemaTypes` class.

Elements (e.g. Rebar) belonging to a particular schema (e.g. `NumberingSchemaTypes.StructuralNumberingSchemas.Rebar`) are organized and numbered in sequences. A sequence is a collection of elements that share the same numbering partition as defined by their respective values of the Partition parameter (`NUMBER_PARTITION_PARAM`). A numbering sequence must contain at least one element. In other words, a sequence is established once there is at least one element of which the partition parameter has a value that differs from other elements (in the same numbering schema). If the last element is removed (deleted or moved to a different sequence) then the empty sequence ceases to exist.

Elements get assigned to sequences either upon their creation (based on the then current numbering partition value), by explicitly modifying the Partition parameter of an element, or by using the `AssignElementsToSequence()` method. The `AssignElementsToSequence()` method is preferred over explicitly changing the Partition parameter, because the method applies changes to sequences and element numbers immediately, while changed parameters only go into effect after the current transaction is closed.

In addition to directly or indirectly changing the Partition parameter of elements, numbering sequences can be reorganized by using methods of the `NumberingSchema` class. The `MoveSequence()` method moves all elements of an existing sequence to a new sequence that does not exist yet in the schema, thus effectively renaming the Partition parameter on all the affected elements. The `AppendSequence()` method removes all elements from one sequence and appends them to elements of another existing sequence while applying the matching policy. The method `MergeSequences()` takes elements of all specified sequences and moves them all into a newly created sequence. All the merged elements will be renumbered and matched as needed based on the matching algorithm.

The sample below uses the MoveSequence() method to swap numbers for Rebar in two numbering sequences.

Code Region: Swap numbers

```
/// <summary>
/// This method uses multiple moving operations to swap numbers
/// for Rebars in two numbering sequences. The sequences are
/// identified by the names of two numbering partitions.
/// </summary>

/// <param name="document">Document to modify</param>
/// <param name="part1">Name of the partition of one numbering sequence</para-
m>
/// <param name="part2">Name of the partition of another numbering sequence</para-
m>

private void SwapNumberingSequences(Document document, string part1, string p-
art2)
{
    // Obtain a schema object for a particular kind of elements
    NumberingSchema schema = NumberingSchema.GetNumberingSchema(document, Numb-
eringSchemaTypes.StructuralNumberingSchemas.Rebar);

    using (Transaction transaction = new Transaction(document))
    {
        // Changes to numbering sequences must be made inside a transaction
        transaction.Start("Swap Numbering Sequences");

        // We will use a temporary partition for the swap operation,
        // for the move operation only works if the target partition
        // does not exist yet in the same numbering schema.
    }
}
```

```
// (We assume this TEMPORARY partition does not exist.)  
  
string tempPartition = "TEMPORARY";  
  
  
// Step 1  
  
// First we move all elements from one sequence into  
// a partition we know does not exist. This action will  
// create the temporary partition and remove the original  
// one (part1).  
  
schema.MoveSequence(part1, tempPartition);  
  
  
// Step 2  
  
// With the sequence in partition 'part1' removed  
// we can now move elements from the second sequence to it.  
// This action will re-create a sequence in partition 'part1'  
// and remove the sequence in partition 'part2'  
  
schema.MoveSequence(part2, part1);  
  
  
// Step 3  
  
// Finally, we can move elements 'parked' in the temporary  
// sequence to partition 'part2', for that partition was  
// removed in the previous step and thus can now be created  
// again. The temporary partition will be automatically  
// removed upon completing this step.  
  
schema.MoveSequence(tempPartition, part2);  
  
  
transaction.Commit();
```

```
    }  
}
```

Elements in different sequences are numbered independently, meaning that there may be elements with the same number in two sequences even though the elements are different. Likewise, there may be perfectly identical elements in two or more sequences bearing different numbers. However, within each one numbering sequence any two identical elements will always have the same number, while different elements will never have the same number within a numbering sequence.

Enumerable elements are always numbered automatically upon their creation. Each new element will get an incrementally higher number. However, new elements that match existing elements within the same sequence will get the same number assigned. Elements will keep their assigned numbers as long as it is possible. This means, for example, that if some previously created rebar elements get deleted, all remaining elements (within the same numbering sequence) will keep their numbers, which may result in gaps in the respective numbering sequence. Gaps can be removed by invoking RemoveGaps() for sequences in which gaps are not desired.

The following example consolidates the numbers on Rebar elements by removing any remaining gaps in numbering sequences and setting the start number of each sequence so numbers in sequences do not overlap.

Code Region: Consolidate Rebar numbers

```
private void ConsolidateRebarNumbers(Document document)  
{  
    // Obtain a schema object for a particular kind of elements  
    NumberingSchema schema = NumberingSchema.GetNumberingSchema(document, NumberingSchemaTypes.StructuralNumberingSchemas.Rebar);  
  
    // Collect the names of partitions of all the numbering sequences currently contained in the schema  
    IList<string> sequences = schema.GetNumberingSequences();  
  
    using (Transaction transaction = new Transaction(document))  
    {
```

```
// Changes to numbers must be made inside a transaction
transaction.Start("Consolidate Rebar Numbers");

// First we make sure numbers in all sequences are consecutive
// by removing possible gaps in numbers. Note: RemoveGaps does
// nothing for a sequence where there are no gaps present.

// We also want to find what the maximum range of numbers is
// of all the sequences (the one the widest span of used numbers)
int maxRange = 0;

foreach (string name in sequences)
{
    schema.RemoveGaps(name);

    // Here we use First() from the Linq extension.
    // There is always at least one range in every sequence,
    // and after gaps are closed there is exactly one range.

    IntegerRange range = schema.GetNumbers(name).First();
    int rangeSpan = 1 + (range.High - range.Low);
    if (rangeSpan > maxRange)
    {
        maxRange = rangeSpan;
    }
}
```

```
// Next we give sequences different start numbers  
  
// starting with 100 and then stepping by at least  
  
// the maximum range we found in the previous step  
  
int startNumber = 100;  
  
  
// We round the range up to the closest 100  
  
int step = 100 * (int)((maxRange + 99) / 100.0);  
  
  
foreach (string name in sequences)  
{  
  
    schema.ShiftNumbers(name, startNumber);  
  
    startNumber += step;  
  
}  
  
  
transaction.Commit();  
}  
}
```

Numbers are stored as values of a numbering parameter on each numbered element. The Id of the parameter is obtained by querying the NumberingSchema.NumberingParameterId property. The value of the number can be obtained by querying the parameter for the respective numbered element. The value is read-only and thus cannot be set; it is always computed based on relations of elements across numbering partitions and the matching policy within the numbering sequence of each element.

Even though numbers are always assigned automatically to all elements of a schema, the method ChangeNumber() gives the programmer a way to explicitly overwrite a specific number as long as the new number is unique in the numbering sequence. The caller specifies a number to be changed and a new value that is to be applied, providing the value does not exist yet in the same numbering sequence.

Distribution type

The Rebar.DistributionType property can be used to modify the type of a rebar set. Rebar sets can be Uniform or VaryingLength. For a uniform distribution type: all bars parameters are the same as the first bar in set. For a varying length distribution type: bars parameters can vary (primarily in length) taking in consideration the constraints of the first bar in set.

The Rebar.GetParameterValueAtIndex() method gets the parameter value for a bar at the specified index. Accepts only values between 0 and NumberOfBarPositions-1. If the DistributionType is Uniform then the returned ParameterValue is the same no matter the index. If the DistributionType is VaryingLength then the returned ParameterValue is evaluated at the given index.

Bar Type Diameter

The options class BarTypeDiameterOptions allows creation of a new set of diameter values for a RebarBarType. It can be used when copying the diameter information as a bulk of data from one RebarBarType to another.

The diameter options can be set for a RebarBarType with RebarBarType.SetBarTypeDiameters() which sets all input diameters from the input BarTypeDiameterOptions in the current RebarBarType.

Constraints

The RebarConstraint class represents a constraint on a handle of a rebar element.

For Shape Driven Rebar Constraints, Each handle on a rebar is defined by a plane, and can be constrained along the direction perpendicular to the plane. Rebar constraints work by locking the handle planes to planar references, or 'targets.'

For Free Form Rebar Constraints, each handle of the Rebar can be constrained to multiple host faces or to the face cover

The RebarConstraintsManager provides information about the constraints (RebarConstraints) acting on the shape handles (RebarConstrainedHandles) of a Rebar element. It can also be used to modify the constraints.

To identify the direction for a positive offset value for a Rebar-to-Rebar constraint RebarConstraint, use `GetPositiveOffsetDirectionForToOtherRebarConstraint()` which returns the positive offset direction vector. This is available only for constraints of type RebarConstraintType.ToOtherRebar. This is the same vector shown in the UI as an arrow representing the direction of a positive offset value.

Geometry

The methods `Rebar.GetTransformedCenterlineCurves()` and `RebarInSystem.GetTransformedCenterlineCurves()` return the centerline curves for a given bar, where the geometry of the curves are in the actual transformed position. The `BarPositionTransform` (representing the relative position of any individual bar in the set - a translation along the distribution path) and `MovedBarTransform` (representing the movement of the bar relative to its default position along the distribution path) will be applied to the returned curves.

Freeform Rebar

The properties: `RebarFreeFormAccessor.RebarStyle` and `RebarFreeFormAccessor.StirrupTieAttachmentType` provide read and write access to the corresponding properties of freeform rebar elements.

The method: `RebarFreeFormAccessor.SetReportedShape()` changes the rebar shape of a freeform rebar that is currently using the `RebarWorkInstructions.Straight` option to the provided rebar shape.

Rebar conversion

The methods `AreaReinforcement.ConvertRebarInSystemToRebars()` and `PathReinforcement.ConvertRebarInSystemToRebars()` convert all of the `RebarInSystem` elements owned by the input element into equivalent `Rebar` elements.

4.4.1.3.2 Rebar Couplers

Rebar couplers are used to connect adjacent rebar.

Creating couplers

Couplers can connect two adjacent rebar or cap the end of a single rebar. Couplers are represented by the `RebarCoupler` class. The static `Create()` method can be used to create new couplers.

The following example creates a coupler to connect two adjacent single rebar bars. The `Create()` method needs the `ElementId` of a coupler type and the `RebarReinforcementData` for the two rebars to connect.

Code Region: Creating a rebar coupler

```
private RebarCoupler CreateCoupler(Document doc, List<Rebar> bars)
{
    RebarCoupler coupler = null;

    // if we have at least 2 bars, create a coupler between them

    if (bars.Count > 1)

    {
        // get a type id for the Coupler
    }
}
```

```
    ElementId defaultTypeId = doc.GetDefaultFamilyTypeId(new ElementId(BuiltInCategory.OST_Coupler));

    if (defaultTypeId != ElementId.InvalidElementId)
    {
        // Specify the rebar and ends to couple
        RebarReinforcementData rebarData1 = RebarReinforcementData.Create(bars[0].Id, 0);
        RebarReinforcementData rebarData2 = RebarReinforcementData.Create(bars[1].Id, 1);

        RebarCouplerError error;
        coupler = RebarCoupler.Create(doc, defaultTypeId, rebarData1, rebarData2, out error);
        if (error != RebarCouplerError.ValidationSuccessfully)
        {
            TaskDialog.Show("Revit", "Create Coupler failed: " + error.ToString());
        }
    }

    // Use a coupler to cap the other end of the first bar
    RebarReinforcementData rebarData = RebarReinforcementData.Create(bars[0].Id, 1);
    RebarCoupler.Create(doc, defaultTypeId, rebarData, null, out error);
    if (error != RebarCouplerError.ValidationSuccessfully)
    {
        TaskDialog.Show("Revit", "Create Coupler failed: " + error.ToString());
    }
}
```

```

    }

}

return coupler;
}

```

If the coupler cannot be created, a `RebarCouplerError` enum will be returned. Some reasons the creation may fail include bars not touching, different layouts between the rebar sets, or the bars may be the wrong diameter for the coupler.

The `CouplerLinkTwoBars()` method will return true if the coupler connects two rebars, or false if only caps one rebar.

The `Rebar` class has a `GetCouplerId()` method to get the id of the a coupler attached to either end of the rebar.

Coupler location and Angle

To get the location point or points (in the case of a rebar set), call the `GetPointsForPlacement()` method, which returns a list of XYZ points indicating where the coupler (or couplers) are placed. The `GetCouplerPositionTransform()` method will return a transform representing the relative position of the coupler at a specified index in the set. The index should be between 0 and the coupler quantity - 1. `GetCouplerQuantity()` will return the quantity of couplers in the set.

`RebarCoupler.RotationAngle` identifies the rotation angle of the coupler around its axis.

Numbering

`RebarCoupler` is one of the categories of elements whose numbering can be controlled via the Revit API. The `NumberingSchema` and `NumberingSchemaType` classes can be used to define how coupler elements are to be organized for the purpose of numbering/tagging them. Each `NumberingSchema` controls numbering of elements of one particular kind. Instances of `NumberingSchema` are also elements and there is always only one of each type in every Revit document. Available types of all built-in numbering schemas are enumerated in `NumberingSchemaTypes` class.

Elements (e.g.`RebarCoupler`) belonging to a particular schema (e.g. `NumberingSchemaTypes.StructuralNumberingSchemas.RebarCoupler`) are organized and numbered in sequences. A sequence is a collection of elements that share the same numbering partition as defined by their respective values of the `Partition` parameter (`NUMBER_PARTITION_PARAM`). A numbering sequence must contain at least one element. In other words, a sequence is established once there is at least one element of which the partition parameter has a value that differs from other elements (in the same numbering schema). If the last element is removed (deleted or moved to a different sequence) then the empty sequence ceases to exist.

End Treatments

The end treatment for a rebar bar comes from the coupler attached to it. The end treatment type for both ends of the coupler is specified by the coupler family type.

The overloaded `EndTreatmentType.Create()` method can be used to create a new `EndTreatmentType` with or without a string to specify the end treatment name. The `CreateDefaultEndTreatmentType()` method creates a new `EndTreatmentType` with a default name.

To access the end treatment type for a `RebarCoupler`, use the `BuiltInParameter COUPLER_MAIN_ENDTREATMENT` to set End Treatment 1 and `COUPLER_COUPLED_ENDTREATMENT` to set End Treatment 2 for the `FamilySymbol` representing the rebar coupler type. The example below creates a new `EndTreatmentType` and assigns it to End Treatment 1 for the coupler type. An existing `EndTreatmentType` can also be used.

Code Region: Change EndTreatmentType for RebarCoupler

```
private void NewEndTreatmentForCouplerType(Document doc, ElementId couplerType
eId)
{
    EndTreatmentType treatmentType = EndTreatmentType.Create(doc, "Custom");

    FamilySymbol couplerType = doc.GetElement(couplerTypeId) as FamilySymbol;

    Parameter param = couplerType.GetParameter(ParameterTypeId.CouplerMainEnd
treatment);

    param.Set(treatmentType.Id);
}
```

An end treatment can be defined in a `RebarShape` to define the shapes that have specific treatments at the ends. The end treatment in the `RebarShape` is used for shape recognition. The user defines the shapes according to their specifications and when a coupler is placed on a bar, it automatically searches for the right `RebarShape`, if the `ReinforcementSettings.RebarShapeDefinesEndTreatments` property is set to true. Note that this property can only be set before any rebar or reinforcement are added to the document.

Code Region: Set and EndTreatmentType for a RebarShape

```
private bool SetEndTreatmentType(Document doc, RebarShape rebarShape)

{
    bool set = false;

    // check if end treatments are defined by rebar shape

    ReinforcementSettings settings = ReinforcementSettings.GetReinforcementSettings(doc);

    if (settings.RebarShapeDefinesEndTreatments)

    {
        EndTreatmentType treatmentType = EndTreatmentType.Create(doc, "Flame Cut");

        rebarShape.SetEndTreatmentTypeId(treatmentType.Id, 0);

        ElementId treatmentTypeId = EndTreatmentType.CreateDefaultEndTreatmentType(doc);

        rebarShape.SetEndTreatmentTypeId(treatmentTypeId, 1);

        set = true;
    }

    else
    {
        try
        {
            // can only be changed if document contains no rebars, area reinforcement or path reinforcement

            settings.RebarShapeDefinesEndTreatments = true;
        }
        catch (Exception e)
        {
    }
```

```

        // cannot change the settings value
        TaskDialog.Show("Revit", e.Message);
    }
}

return set;
}

```

The RebarShape class has methods to determine if it has end treatments, and methods to get and set the EndTreatmentType for both ends of the RebarShape.

Note that Rebar Couplers can't be placed on Area, Path, or Rebar Container.

Code Region: Get end treatments for Rebar

```

private void ListEndTreatments(Document doc, List<Rebar> bars)
{
    StringBuilder info = new StringBuilder();

    for (int n = 0; n < bars.Count; n++)
    {
        // get end treatment for both ends of bar
        for (int i = 0; i < 2; i++)
        {
            ElementId treatmentTypeId = bars[n].GetEndTreatmentTypeId(i);

            if (treatmentTypeId != ElementId.InvalidElementId)
            {
                EndTreatmentType treatmentType = doc.GetElement(treatmentType
Id) as EndTreatmentType;

                info.AppendLine(string.Format("End treatment for bar {0} end
{1}: {2}", n, i, treatmentType.EndTreatment));
            }
        }
    }
}

```

```

        }

    }

}

TaskDialog.Show("Revit", info.ToString());
}

```

4.4.1.3.3 Area and Path Reinforcement

The Revit API provides classes representing area and path reinforcement in the structural features of Revit.

Find the `AreaReinforcementCurves` for `AreaReinforcement` by calling the `GetBoundaryCurveIds()` method which returns an `IList` of `ElementIds` that represent `AreaReinforcementCurves`.

While the `AreaReinforcement.GetBoundaryCurveIds()` method returns a set of `ElementIds` representing `AreaReinforcementCurves`, which have a property that returns a `Curve`, the `PathReinforcement.GetCurveElementIds()` method returns a collection of `ElementIds` that represent `ModelCurves`. There is no way to flip the `PathReinforcement` except by on creation using the `PathReinforcement.Create()` method. `PathReinforcement` can only be created using an array of curves.

For more details about retrieving an Element's Geometry, refer to [Geometry](#).

Note: Project-wide settings related to area and path reinforcement are accessible from the [ReinforcementSettings](#) class.

The API provides access to the layers of `Area Reinforcement` elements and allows the individual lines exposed through those layers to be moved and removed.

The overloaded `AreaReinforcement.Create()` method provides two ways to create new `AreaReinforcement`: based on a host boundary or from an array of curves. The Major Direction of the area reinforcement can be set when creating a new `AreaReinforcement` using either of the overloaded `Create()` methods, but the `AreaReinforcement.Direction` property is read-only.

Creating area reinforcement

```
AreaReinforcement CreateAreaReinforcementInWall(Wall wall, Autodesk.Revit.DB.Document document)
```

```

{

    // Get the wall analytical profile whose curves will define the boundary
    // of the area reinforcement

    AnalyticalPanel analytical = (AnalyticalPanel)document.GetElement(AnalyticalToPhysicalAssociationManager.GetAnalyticalToPhysicalAssociationManager(document)

        .GetAssociatedElementId(wall.Id));

    if (null == analytical)

    {

        throw new Exception("Can't get AnalyticalModel from the selected wall");
    }

}

IList<Curve> curves = analytical.GetOuterContour().Cast<Curve>().ToList()
();

//define the Major Direction of AreaReinforcement,
//we get direction of first Line on the Wall as the Major Direction

Line firstLine = (Line)(curves[0]);

XYZ majorDirection = new XYZ(
    firstLine.GetEndPoint(1).X - firstLine.GetEndPoint(0).X,
    firstLine.GetEndPoint(1).Y - firstLine.GetEndPoint(0).Y,
    firstLine.GetEndPoint(1).Z - firstLine.GetEndPoint(0).Z);

// Obtain the default types

ElementId defaultRebarBarTypeId = document.GetDefaultElementTypeId(ElementTypeGroup.RebarBarType);

ElementId defaultAreaReinforcementTypeId = document.GetDefaultElementTypeId(ElementTypeGroup.AreaReinforcementType);

ElementId defaultHookTypeId = ElementId.InvalidElementId;
}

```

```

    // Create the area reinforcement

    AreaReinforcement rein = AreaReinforcement.Create(document, wall, curves,
        majorDirection, defaultAreaReinforcementTypeId, defaultRebarBarTypeId, defau
        ltHookTypeId);

    return rein;
}

```

Creating path reinforcement

The overloaded static method PathReinforcement.Create() method provides two ways to create path reinforcement. Both create path reinforcement in a host object from an array of curves, but one will use the default rebar shape while the other takes a Rebar Shape id as a parameter. The example below uses the default rebar shape.

Creating path reinforcement

```

PathReinforcement CreatePathReinforcement(Autodesk.Revit.DB.Document document,
    Wall wall)

{
    // Create a geometry line in the selected wall as the path

    List<Curve> curves = new List<Curve>();

    LocationCurve location = wall.Location as LocationCurve;

    XYZ start = location.Curve.GetEndPoint(0);

    XYZ end = location.Curve.GetEndPoint(1);

    curves.Add(Line.CreateBound(start, end));

    // Obtain the default types

    ElementId defaultRebarBarTypeId = document.GetDefaultElementTypeId(Elemen
    tTypeGroup.RebarBarType);

```

```

    ElementId defaultPathReinforcementTypeId = document.GetDefaultElementType
    Id(ElementTypeGroup.PathReinforcement);

    ElementId defaultHookTypeId = ElementId.InvalidElementId;

    // Begin to create the path reinforcement

    PathReinforcement rein = PathReinforcement.Create(document, wall, curves,
    true, defaultPathReinforcementTypeId, defaultRebarBarTypeId, defaultHookType
    Id, defaultHookTypeId);

    if (null == rein)

    {

        throw new Exception("Create path reinforcement failed.");

    }

    // Give the user some information

    TaskDialog.Show("Revit", "Create path reinforcement succeed.");

    return rein;

}

```

When specifying the rebar shape id for a new PathReinforcement, if there are no rebar shapes in the project or you are not initially concerned with the rebar shape, you can use the static PathReinforcement method GetOrCreateDefaultRebarShape() to obtain a valid rebar shape for use with PathReinforcement. If you would like to check whether an existing rebar shape is valid for use with path reinforcement, you can call the static method PathReinforcement.IsValidRebarShapeId().

New shape types may be queried, or assigned to the path reinforcement by using the PathReinforcement properties PrimaryBarShapeId and AlternatingBarShapeId. The static method IsValidRebarShapeId() can be used to determine if you have a valid shape before attempting to set the shape id on a path reinforcement object. Note that before attempting to set alternating bars, the alternating bars parameter must be enabled in the Path Reinforcement by setting PATH_REIN_ALTERNATING BuiltInParameter to true.

The orientation of the primary and alternating bars may also be queried, or set through the properties PrimaryBarOrientation and AlternatingBarOrientation which take a value from the ReinforcementBarOrientation enumeration. You may check whether an orientation is valid for a

particular path reinforcement object by calling the class method `IsValidPrimaryBarOrientation()` or `IsValidAlternatingBarOrientation()`.

You may query the state of the alternating layer by calling the `IsAlternatingLayerEnabled()` method. The alternating layer is controlled via the built in parameter `PATH_REIN_ALTERNATING` on the path reinforcing element.

The API provides access to move, include, or remove individual bars for `RebarInSystem` elements that are owned by `PathReinforcement`.

4.4.1.3.4 Fabric Reinforcement

Fabric reinforcement is a layer of fabric sheets made of welded fabric wire and hosted in concrete slabs or walls.

Fabric sheets are typically created in a pattern which results in welded wire sheets overlapping each other to provide continuity of load transfer from one sheet to the next. Sheets may be in one or more layers within a concrete element.

Fabric Sheets are typically specified by a grid spacing and a wire size for each direction. For example: 6x6 - W2.9/W2.9 in US Imperial units (or 152x152-MW18.7/MW18.7 in SI units) would be interpreted as a grid of wires at 6" on center with a wire area of 2.9 hundredths of an inch squared (0.029 sq-in).

In the Revit API, these sheets are represented by the `FabricSheet` class and they are defined by a `FabricSheetType` class which controls the number and spacing of wires in each direction, either by a pattern or custom definition. `FabricSheetType` reference wires as `FabricWireItems` that are associated with a `FabricWireType`.

Fabric Area

The `FabricArea` class is a container for fabric sheets. When created, fabric sheets are automatically generated for the `FabricArea` based on the `FabricSheetType` referenced on creation. The `FabricArea` class has an overloaded `Create()` method to create a `FabricArea` based either on the host boundary or from an array of curves. `FabricArea` must be hosted by a structural floor, wall, foundation slab or a part created from a structural layer of one of these types.

The example below demonstrates how to create a new `FabricArea` and get a list of the resulting `FabricSheets`.

Code Region: Create a fabric area

```
private FabricArea CreateNewFabricArea(Document document, Element wall)
{
}
```

Code Region: Create a fabric area

```
FabricArea system = null;

// create default types if they aren't already in the model
ElementId fabricAreaTypeId = FabricAreaType.CreateDefaultFabricAreaType(document);

ElementId fabricSheetTypeId = FabricSheetType.CreateDefaultFabricSheetType(document);

system = FabricArea.Create(document, wall, new XYZ(1, 0, 0), fabricAreaTypeId, fabricSheetTypeId);

// call regenerate to generate fabric sheets in fabric area
document.Regenerate();

// get the list of elementIds for the sheets automatically generated in the fabric area
IList<ElementId> sheetIds = system.GetFabricSheetElementIds();

TaskDialog.Show("Revit", string.Format("{0} fabric sheets created", sheetIds.Count));

return system;
}
```

Creating fabric sheets

Fabric sheets can be hosted by a container (represented by the `FabricArea` class), or can exist as single fabric sheets. Single fabric sheets are hosted elements and must be hosted by a structural floor, structural wall, structural foundation slab, or a part created from a structural layer of one of these types. Bent fabric sheets can also be hosted in structural beams, columns and braces.

The `FabricSheet` class provides an overloaded static `Create()` method for creating new single fabric sheets in the model. One overload of `FabricSheet.Create()` creates a flat fabric sheet and

requires a reference to the document in which the fabric sheet will be created, a reference to the host element which will host the fabric sheet and the ElementId of the fabric sheet type to create.

Another overload of FabricSheet.Create() can be used to create bent fabric sheets, which is not possible in the Revit user interface. It requires the same input as above, plus a CurveLoop that defines the bending path. This bending path is a profile that defines the bending shape of the fabric sheet. Fabric wires have an allowable bend radii so you may provide this CurveLoop profile with or without fillets. In other words, if a U shaped profile is desired, only 3 lines must be specified and the fabric sheet created will be created with the appropriate bend radii at the corners. If the provided profile has no hard corners, (I.e., the curve has a tangent at each point, except the end) no fillets will be created in the final fabric sheet.

The provided CurveLoop is intended to define the centerline of the bent wire. You may specify whether the bend is in the major wires or the minor wires with the BentFabricBendDirection property of the FabricSheet. You may also specify the location of straight wires with respect to bent wires with the BentFabricStraightWiresLocation property.

Fabric sheet type

A FabricSheet is associated with a FabricSheetType, which is used in the generation of wires for the sheet in the major and minor directions. The FabricSheetLayoutPattern for the wires in a FabricSheetType can be defined as:

- ActualSpacing - spacing of rebars is fixed
- FixedNumber - spacing of rebars is adjustable, but the number of bars is constant
- MaximumSpacing - number of rebars depends on length of rebar set with a maximum spacing constraint specified
- NumberWithSpacing - spacing and number of rebars are fixed
- QuantitativeSpacing - multiple groups of wires with a specific spacing and diameter For the first 4 options, FabricSheetType has methods to set the layout pattern with given information for either the minor or major direction, such as SetMajorLayoutAsFixedNumber() or SetMinorAsActualSpacing(). The layout pattern can be retrieved with the properties MajorLayoutPattern and MinorLayoutPattern.

QuantitativeSpacing is for custom patterns. Use SetLayoutAsCustomPattern() to set the major and minor layout patterns to QuantitativeSpacing. If the layout pattern for the FabricSheetType is custom, the MajorDirectionWireType, MinorDirectionWireType, MajorSpacing and MinorSpacing properties do not apply. The SetLayoutAsCustomPattern() method has parameters for the start overlaps for the major and minor directions (both end overhangs are read only and computed internally), as well as a list of FabricWireItems for each direction.

Code Region: Fabric sheet with custom wire pattern

```
private FabricSheet CreateCustomFabricSheet(Document document, Element wall)
```

Code Region: Fabric sheet with custom wire pattern

```
{  
  
    if (FabricSheet.IsValidHost(wall) == false)  
  
        return null;  
  
  
    // Create a new type for custom FabricSheet  
  
    ElementId fabricSheetTypeId = FabricSheetType.CreateDefaultFabricSheetTyp  
e(document);  
  
    FabricSheetType fst = document.GetElement(fabricSheetTypeId) as FabricShe  
etType;  
  
  
    // Create some fabric wire types  
  
    ElementId idWireType1 = FabricWireType.CreateDefaultFabricWireType(docume  
nt);  
  
    FabricWireType wireType1 = document.GetElement(idWireType1) as FabricWire  
Type;  
  
    wireType1.WireDiameter = 3.5 / 12.0;  
  
  
    ElementId idWireType2 = FabricWireType.CreateDefaultFabricWireType(docume  
nt);  
  
    FabricWireType wireType2 = document.GetElement(idWireType1) as FabricWire  
Type;  
  
    wireType2.WireDiameter = 2.0 / 12.0;  
  
  
    // Create the wires for the custom pattern  
  
    IList<FabricWireItem> majorWires = new List<FabricWireItem>();  
  
    IList<FabricWireItem> minorWires = new List<FabricWireItem>();  
  
    FabricWireItem item = FabricWireItem.Create(2.0 / 12.0, 1, idWireType1,  
0);
```

Code Region: Fabric sheet with custom wire pattern

```
majorWires.Add(item);

majorWires.Add(item);

item = FabricWireItem.Create(1.5 / 12.0, 10.0 / 12.0, idWireType2, 0);

majorWires.Add(item);

item = FabricWireItem.Create(3.0 / 12.0, 1, idWireType2, 0);

minorWires.Add(item);

item = FabricWireItem.Create(3.0 / 12.0, 10.0 / 12.0, idWireType2, 0);

minorWires.Add(item);

fst.SetLayoutAsCustomPattern(6.0 / 12.0, 4.0 / 12.0, minorWires, majorWires);

FabricSheet sheet = FabricSheet.Create(document, wall, fabricSheetTypeID);

// Regeneration is required before setting any property to object that was created in the same transaction.

document.Regenerate();

AnalyticalMember member = (AnalyticalMember)document.GetElement(AnalyticalToPhysicalAssociationManager.GetAnalyticalToPhysicalAssociationManager(document)

    .GetAssociatedElementId(wall.Id));

sheet.PlaceInHost(wall, member.GetTransform());

// Give the user some information

TaskDialog.Show("Revit", string.Format("Flat Fabric Sheet ID='{0}' created successfully.", sheet.Id.IntegerValue));
```

Code Region: Fabric sheet with custom wire pattern

```
    return sheet;  
}
```

Single fabric sheets

When fabric sheets are created, they are not yet placed in their host. You must call the method `FabricSheet.PlaceInHost()` and pass in the host element as well as a transform which describes the final position of the fabric sheet. The element passed for the host must support hosting fabric sheets. Note that only bent fabric sheets can be hosted in beams, columns or braces, while both bent and flat fabric sheets can be hosted in structural floors, walls and foundation slabs, or parts created from a structural layer of one of these types.

Code Region: Creating a bent fabric sheet

```
private FabricSheet CreateBentFabricSheet(Document document, Element column)  
{  
    if (FabricSheet.IsValidHost(column) == false)  
        return null;  
  
    CurveLoop bendingProfile = new CurveLoop();  
    Line line1 = Line.CreateBound(new XYZ(2, 0.8, 0), new XYZ(6, 0.8, 0));  
    Line line2 = Line.CreateBound(new XYZ(6, -0.8, 0), new XYZ(4, -2, 0));  
    Arc arc = Arc.Create(line1.GetEndPoint(1), line2.GetEndPoint(0), new XYZ  
    (7, 0, 0));  
    bendingProfile.Append(line1);  
    bendingProfile.Append(arc);  
    bendingProfile.Append(line2);
```

```

ElementId fabricSheetTypeId = document.GetDefaultElementType( ElementTypeGroup.FabricSheetType);

FabricSheetType fst = document.GetElement(fabricSheetTypeId) as FabricSheetType;

fst.SetMajorLayoutAsFixedNumber(12.0, 1.0, 0.8, 4);

fst.SetMinorLayoutAsMaximumSpacing(10.0, .75, .90, 3.0);

FabricSheet bentFabricSheet = FabricSheet.Create(document, column.Id, fabricSheetTypeId, bendingProfile);

// Regeneration is required before setting any property to object that was created in the same transaction.

document.Regenerate();

bentFabricSheet.BentFabricBendDirection = BentFabricBendDirection.Major;

AnalyticalMember amc = (AnalyticalMember)document.GetElement(AnalyticalToPhysicalAssociationManager.GetAnalyticalToPhysicalAssociationManager(document)

    .GetAssociatedElementId(column.Id));

bentFabricSheet.PlaceInHost(column, amc.GetTransform());

// Give the user some information

TaskDialog.Show("Revit", string.Format("Bent Fabric Sheet ID='{0}' created successfully.", bentFabricSheet.Id.IntegerValue));

return bentFabricSheet;
}

```

Bent fabric sheets

The `IsBent` property indicates if the fabric sheet is a bent fabric sheet or not. It is not possible to convert a Fabric Sheet between flat and bent.

You may obtain the curves for a bent sheet or set new curves by calling GetBendProfile() or SetBendProfile() respectively.

Code Region: Change the bend profile of a fabric sheet

```
private void ModifyBentFabricSheet(Document document, FabricSheet bentFabricSheet)
{
    CurveLoop newBendingProfile = CurveLoop.CreateViaOffset(bentFabricSheet.GetBendProfile(), 0.5, new XYZ(0, 0, -1));
    bentFabricSheet.SetBendProfile(newBendingProfile);

    // Give the user some information
    TaskDialog.Show("Revit", string.Format("Bent Fabric Sheet ID='{0}' modified successfully.", bentFabricSheet.Id.IntegerValue));
}
```

The GetSegmentParameterIdsAndLengths() method will return a dictionary with segment parameter ids as the keys and length as the values. These correspond to segments of a bent fabric sheet (like A, B, C, D etc.), but are not in alphabetical or any other order. (An empty dictionary is returned for flat fabric sheets.) To set the length of a bent fabric sheet segment, use SetSegmentLength().

Code Region: Get segment ids and lengths

```
private void GetBentFabricSheetData(FabricSheet fabricSheet)
{
    string fabricNumber = fabricSheet.FabricNumber;
    IDictionary<ElementId, double> idsAndLengths = fabricSheet.GetSegmentParameterIdsAndLengths(true);
    StringBuilder displayInfo = new StringBuilder();
    displayInfo.AppendLine(string.Format("Parameter Ids and segment lengths for FabricSheet {0}:", fabricNumber));
```

Code Region: Get segment ids and lengths

```

foreach (ElementId key in idsAndLengths.Keys)
{
    displayInfo.AppendLine(string.Format("Parameter Id: {0}, Length: {1}",
    " ", key, idsAndLengths[key]));
}

TaskDialog.Show("Revit", displayInfo.ToString());
}

```

The FabricNumber property used in the example above, specifies the numerical parameter assigned to the fabric sheet and any sheet of the same type, dimension, material, shape, and partition.

4.4.1.3.5 Rebar Containers

Rebar Containers are elements which represent an aggregation of rebars in one host. This element can only be created via the API.

The advantages of using the RebarContainer element are:

- Defining new types of rebar distributions not possible with the Revit user interface
- Improve rebar performance by combining multiple rebar sets into the definition of a single element
- The bars don't react to the host's geometry modifications

Each RebarContainer contains a collection (empty when first created) of RebarContainerItem objects. RebarContainerItem provides properties and methods similar to that of a Rebar element.

Creating a new RebarContainer

You may use the static method RebarContainer.Create() which requires the Document in which you want to create a new RebarContainer, the host Element which will host the new RebarContainer and the ElementId of the RebarContainerType which will be assigned to the new RebarContainer.

The following example creates a new rebar container and specifies that any items in the container will be presented as subelements in schedules and tags.

Code Region: Creating a rebar container

```
RebarContainer CreateRebarContainer(Autodesk.Revit.DB.Document document, FamilyInstance beam)

{
    // Create a new rebar container

    ElementId defaultRebarContainerTypeId = RebarContainerType.CreateDefaultRebarContainerType(document);

    RebarContainer container = RebarContainer.Create(document, beam, defaultRebarContainerTypeId);

    // Any items for this container should be presented in schedules and tags
    // as separate subelements

    container.PresentItemsAsSubelements = true;

    return container;
}
```

Filling a RebarContainer

Once you have a new RebarContainer object, you can fill it up with RebarContainerItems by calling one of the following methods on the RebarContainer object:

- AppendItemFromRebar() - Appends a RebarContainerItem to the collection using properties of the Rebar element used to create it.
- AppendItemFromCurves() - Appends a RebarContainerItem to the collection from a list of curves using properties from the specified RebarBarType.
- AppendItemFromRebarShape() - Appends a RebarContainerItem to the collection using properties from the specified RebarBarType and RebarShapeType.
- AppendItemFromCurvesAndShape() - Appends a RebarContainerItem to the collection from a list of curves using properties from the specified RebarBarType and RebarShapeType. Note that in creating the container items, you will often be required to specify the normal of the plane in which the curves exist. This normal also defines the direction of multiple rebar creation if you set the layout rules to anything other than single. If you set the

RebarContainerItem to a FixedNumber layout rule, for instance, those additional bars will be distributed along a line perpendicular to the plane you specify when creating the RebarContainerItem. In this sense the Normal also serves as the direction of distribution for RebarContainerItems with layout rules applied.

Accessing RebarContainerItems

Use the method RebarContainer.Contains() to query whether the particular RebarContainerItem exists in the RebarContainer.

RebarContainer provides methods to remove individual items, clear all items, get specific items and return the count of RebarContainerItems in the container. The SetItemHiddenStatus() method controls whether specific items in the container are hidden in a specific view or not, and IsItemHidden() identifies if an item is hidden in a view.

Code Region: Adding items to a rebar container

```
void AddItemsToRebarContainer(RebarContainer container, FamilyInstance beam,
RebarBarType barType, RebarHookType hookType)

{
    // Define the rebar geometry information - Line rebar
    LocationCurve location = beam.Location as LocationCurve;
    XYZ origin = location.Curve.GetEndPoint(0);

    // create rebar along the length of the beam
    XYZ rebarLineEnd = location.Curve.GetEndPoint(1);
    Line line = Line.CreateBound(origin, rebarLineEnd);
    XYZ normal = new XYZ(1, 0, 0);
    Curve rebarLine = line.CreateOffset(0.5, normal);

    // Create the line rebar
    IList<Curve> curves = new List<Curve>();
    curves.Add(rebarLine);
```

```

    RebarContainerItem item = container.AppendItemFromCurves(RebarStyle.Stand
ard, barType, hookType, hookType, normal, curves, RebarHookOrientation.Right,
RebarHookOrientation.Left, true, true);

    if (null != item)

    {
        // set specific layout for new rebar as fixed number, with 10 bars, d
istribution path length of 1.5'

        // with bars of the bar set on the same side of the rebar plane as in
dicated by normal

        // and both first and last bar in the set are shown

        item.SetLayoutAsFixedNumber(10, 1.5, true, true, true);

    }

    // Hide the new item in the active view

    container.SetItemHiddenStatus(container.Document.ActiveView, item.ItemInd
ex, true);
}

```

RebarContainerType

Rebar Containers will have types. These types are agnostic to the individual RebarContainerItems contained in them but will afford access to the Container's parameters at the type level. It is a required argument for creation of a new RebarContainer. Obtain a default by calling the static method RebarContainerType.CreateDefaultRebarContainerType() to obtain the ElementId of a RebarContainerType if the desired one does not exist in the database.

RebarContainer parameters management

Because the RebarContainer may contain several different configurations of individual RebarContainerItems, the overall RebarContainer parameters will be derived from the contained RebarContainerItem members if possible. If the parameter exists on all RebarContainerItems and it is the same across all RebarContainerItems in the RebarContainer then the RebarContainer parameter will show this value. If they are different, or do not exist on all RebarContainerItems in the Container, the parameter will display without value.

To allow you to manage this and specify necessary parameters for the RebarContainer as a whole, you may request the RebarContainerParameterManager from the RebarContainer calling the method GetParametersManager() which will return an object you can use to override

the parameters of the RebarContainer. As an example, TotalLength in a RebarContainer with multiple RebarContainerItems with differing lengths, will show an empty parameter for TotalLength. You can call the RebarContainerParametersManager method AddOverride() which has 4 overloads so that you can override the value of any of four value types of parameters (ElementId, Double, Integer, String). Use the method IsRebarContainerParameter() to determine if the parameter is a RebarContainer parameter before attempting to add an override for the parameter.

You can also add a shared parameter as an override to a RebarContainer by calling the method AddSharedParameterAsOverride() and passing in the id of the shared parameter. Check to see that the parameter has not already been added to the RebarContainer before calling this method.

You may remove overrides by calling the method RemoveOverride() or remove all overrides by calling ClearOverrides().

Individual overridden parameters may be set to be modifiable or read only by calling the methods SetOverriddenParameterModifiable() or SetOverriddenParameterReadOnly() and query the same with the IsOverriddenParameterModifiable() method.

Lastly, to check if a RebarContainer parameter is overridden, call the method IsParameterOverridden() passing in the ElementId of the parameter in question.

Code Region: RebarContainer parameters management

```
void ManageParameters(RebarContainer container)
{
    RebarContainerParameterManager paramManager = container.GetParametersManager();

    Parameter aParam = container.LookupParameter("A");
    if (aParam != null)
    {
        paramManager.AddOverride(aParam.Id, 0.4);
        paramManager.SetOverriddenParameterModifiable(aParam.Id);
    }
}
```

RebarContainerItems

Rebar container items offer many of the same properties as do Rebar elements. They are a much lighter representation of a modeled rebar and unlike the Rebar class which is derived from Element, RebarContainerItem is derived from the IDisposable Interface.

You may redefine the RebarContainerItem represented by any RebarContainerItem by calling the class methods available on RebarContainerItem objects:

- SetFromRebar()
- SetFromCurves()
- SetFromRebarShape()
- SetFromCurvesAndShape() These methods mimic the methods which were used on RebarContainer to create the RebarContainerItem in the first place and have identical or nearly identical arguments which can be used to modify any RebarContainerItem in the RebarContainer.

You may query the RebarBarType ID with the class property BarTypeId. In order to apply a different RebarBarType to the RebarContainerItem you must use the SetFrom methods on the RebarContainerItem (i.e. SetFromCurves() or SetFromRebar()).

4.4.1.3.6 Reinforcement Settings

Several settings regarding reinforcement in the model are controlled at the document level and are accessed through the ReinforcementSettings class for the document.

These settings include reinforcement rounding document level overrides, hosting of rebar elements on path and area reinforcement, as well as tag abbreviations for reinforcement tagging.

Access the document reinforcement settings object by calling the static ReinforcementSettings.GetReinforcementSettings() which takes a reference to the document and returns the corresponding reinforcement settings object.

Reinforcement rounding overrides

You may access settings for document level reinforcement rounding overrides by using the methods GetRebarRoundingManager() or GetFabricRoundingManager()

Hosting rebar on area and path reinforcing

The property HostStructuralRebar controls whether Revit will automatically host real rebar on path and area reinforcement. Before setting this property to true, you must verify that there are no path or area reinforcing objects currently in the document by using a FilteredElementCollector.

Include hooks in rebar shape definition

The RebarShapeDefinesHooks property controls whether rebar shape definitions include hooks in the definition. This maps to the UI property "Include hooks in Rebar Shape definition" setting

in Reinforcement Settings. Note that this property may not be changed if the model contains: rebar elements, area or path reinforcing, or rebar containers. Use a FilteredElementCollector to query the model for the existence of any of these elements before attempting to set this property.

Include end treatments in rebar shape definition

The RebarShapeDefinesEndTreatments property indicates if end treatments are defined by the RebarShape. This property can only be set if there are no rebars, area reinforcements or path reinforcements.

Reinforcement abbreviation tags

You may access as well as apply new rebar tag abbreviations. Engineers typically specify rebar with certain abbreviations (E.g., NF = near face, E.W. = each way). There are 16 different abbreviation tags stored by the ReinforcementSettings object and are indexed by the enumeration ReinforcementAbbreviationTagType.

Query current abbreviation tags by calling the method GetReinforcementAbbreviationTag() which will return a string value of the current abbreviation corresponding to the tag type passed to the method. You may also retrieve all of the abbreviation tags associated with either path or area reinforcement by calling the method GetReinforcementAbbreviationTags().

Set new abbreviations by calling the method SetReinforcementAbbreviationTag() passing a ReinforcementAbbreviationTagType to specify for which tag type you wish to set the new abbreviation.

Varying Rebar Set

The NumberVaryingLengthRebarsIndividually property of ReinforcementSettings modifies the way varying length bars are numbered (individually or as a whole). If true, then each bar in a varying rebar set will be assigned a rebar number. Otherwise, the whole rebar set will be assigned a unique rebar number and each bar within the set will be assigned a suffix that is unique within the set.

The ReinforcementSettings.RebarVaryingLengthNumberSuffix property is a unique string identifier used for a bar within a variable length rebar set.

4.4.1.3.7 Reinforcement Rounding

Reinforcement rounding is important for preparation of construction documents. You can use numerical rounding of rebar lengths to simplify organization and annotation when tagging, filtering, or scheduling rebar.

Revit enables overriding of three different reinforcement parameters: Fabric Sheet lengths, Rebar total lengths, and Rebar segment lengths. Rebar segment lengths are the individual segments which make up a rebar shape.

The application of Rebar and/or Fabric Sheet rounding is controlled initially at the Document level. If rounding is not enabled at the document level, the rounding for Reinforcement Length

(structural unit settings) will be applied to rebar lengths in schedules and tags. If the Document level reinforcement settings are overridden then the rebar settings control rounding for rebars and fabric sheets and may be overridden further at the type and element level.

Overriding reinforcement rounding settings

The ReinforcementSettings class provides methods to obtain either the fabric or rebar rounding managers by calling GetFabricRoundingManager() or GetRebarRoundingManager() respectively. These rounding managers represent the document level rounding overrides and will control rounding for any types or elements which are not otherwise overridden by accessing the corresponding rounding manager for the type or element in question.

Reinforcement rounding override is enabled by setting the ReinforcementRoundingManager.IsActiveOnElement property to true at the document level. This property must be set to true before attempting to override the reinforcement rounding settings for the document, a type, or an element.

The ReinforcementRoundingManager has two derived classes, FabricRoundingManager and RebarRoundingManager, to handle overrides for these different types of elements. FabricRoundingManager is used for managing reinforcement rounding override settings used by FabricSheetType and FabricSheet elements while RebarRoundingManager is used for managing reinforcement rounding override settings used by RebarBarTypes, Rebar and RebarInSystem elements.

To override reinforcement rounding settings for a rebar type, call Rebar.GetReinforcementRoundingManager() to get a reference to a RebarRoundingManager. Once you have access to the RebarRoundingManager, to override the rounding settings, first set RebarRoundingManager.IsActiveOnElement to true before attempting to set any of the rounding properties.

RebarRoundingManager.TotalLengthRounding is the rounding increment that will be used in rounding. (e.g., 1" means that the actual length will be rounded to an increment of 1") RebarRoundingManager.TotalLengthRoundingMethod takes a RoundingMethod enum which has the following possible values:

- Nearest - Standard rounding: round to nearest
- Up - Round up
- Down - round down

RebarRoundingManager also provides access to the Segment length rounding with properties RebarRoundingManager.SegmentLengthRounding and RebarRoundingManager.SegmentLengthRoundingMethod.

To override reinforcement settings on a fabric sheet type or a fabric sheet element, calling GetReinforcementRoundingManager() on the corresponding class will return a FabricRoundingManager. The FabricRoundingManager class has similar functionality to the RebarRoundingManager for controlling rounding for fabric sheets.

Querying an element's current reinforcement rounding settings

You may access information about what is currently controlling the reinforcement rounding for any element by querying the following properties:

`RebarRoundingManager.ApplicableReinforcementRoundingSource` (for rebar) or `FabricRoundingManager.ApplicableReinforcementRoundingSource` (for fabric sheets). You may also query current rounding settings by using the `ApplicableTotalLengthRoundingMethod` and `ApplicableTotalLengthRounding` properties, available from both types of rounding managers .

4.4.1.3.8 Annotation

The `MultiReferenceAnnotation` can be used to label and dimension Rebar elements, and are labeled in the user interface as "Multi-rebar annotations". The members of this class include:

- Create - creates a new `MultiReferenceAnnotation` based on a document, view, and an instance of the `MultiReferenceAnnotationOptions` class
- `AreElementsValidForMultiReferenceAnnotation` - validates if the input elements match the element category id for the `MultiReferenceAnnotationType`.
- `is3DViewValidForDimension` - Returns True if the view is suitable for placing the `MultiReferenceAnnotation`
 - If the `DimensionStyle` is `LinearFixed`, it cannot be created in a 3D View.
 - If the `DimensionStyle` is `Linear`, it cannot be created in a 3D View if the view direction is perpendicular to the current work plane normal.
 - Returns true if the `ownerViewId` is not a 3D view.

The references of the annoation and placement options are specified in the `MultiReferenceAnnotationOptions` class. Its members include:

- `DimensionLineDirection` - The direction vector of the dimension line
- `SetElementsToDimension` - The elements referenced by the dimension
- `SetAdditionalReferencesToDimension` - Sets the additional references which the dimension will witness. The additional references to dimension cannot come from the same category as the `MultiReferenceAnnotationType`'s reference category.
- `ReferencesDontMatchReferenceCategory` - Verifies that all of the references belongs to elements which don't match the reference category required by the `MultiReferenceAnnotationType`.
- `GetAdditionalReferencesToDimension` - Gets the additional references which the dimension will witness

Code Region: Create MultiReferenceAnnotation

```
public void CreateAnnotation(Document doc)
{
}
```

Code Region: Create MultiReferenceAnnotation

```

    MultiReferenceAnnotationOptions opt = new MultiReferenceAnnotationOptions
    (
        new FilteredElementCollector(doc)
            .OfClass(typeof(MultiReferenceAnnotationType))
            .Cast<MultiReferenceAnnotationType>()
            .FirstOrDefault();

        opt.DimensionLineDirection = XYZ.BasisX;
        opt.DimensionPlaneNormal = XYZ.BasisZ;
        opt.DimensionStyleType = DimensionStyleType.Linear;
        opt.SetElementsToDimension(new FilteredElementCollector(doc, doc.ActiveView.Id).OfClass(typeof(Rebar)).ToElementIds());
    }

    using (Transaction t = new Transaction(doc, "MultiReferenceAnnotation"))
    {
        t.Start();
        MultiReferenceAnnotation mra = MultiReferenceAnnotation.Create(doc, doc.ActiveView.Id, opt);
        t.Commit();
    }
}

```

4.4.1.4 Boundary Conditions**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 : ";

    boundaryConditions.GetBoundaryConditionsType();

    switch (boundaryConditions.GetBoundaryConditionsType())
    {
        case BoundaryConditionsType.Point:
            XYZ point = boundaryConditions.Point;

            message += "\nThis BoundaryConditions is a Point Boundary Conditions.";
            message += "\nLocation point: (" + point.X + ", "
                + point.Y + ", " + point.Z + ")";

            break;

        case BoundaryConditionsType.Line:
            message += "\nThis BoundaryConditions is a Line Boundary Conditions.";

            Curve curve = boundaryConditions.GetCurve();

            // Get curve start point

            message += "\nLocation Line: start point: (" + curve.GetEndPoint(0).X + ", "
                + curve.GetEndPoint(0).Y + ", " + curve.GetEndPoint(0).Z
                + ")";

            // Get curve end point

            message += "; end point:(" + curve.GetEndPoint(1).X + ", "
                + curve.GetEndPoint(1).Y + ", " + curve.GetEndPoint(1).Z
                + ")";
    }
}
```

```
        break;

    case BoundaryConditionsType.Area:

        message += "\nThis BoundaryConditions is an Area Boundary Conditions.";
        IList<CurveLoop> loops = boundaryConditions.GetLoops();
        foreach (CurveLoop curveLoop in loops)
        {
            foreach (Curve areaCurve in curveLoop)
            {
                // Get curve start point
                message += "\nCurve start point:( " + areaCurve.GetEndPoint(0).X + ", "
                           + areaCurve.GetEndPoint(0).Y + ", " + areaCurve.GetEndPoint(0).Z + ")";

                // Get curve end point
                message += "; Curve end point:( " + areaCurve.GetEndPoint(1).X + ", "
                           + areaCurve.GetEndPoint(1).Y + ", " + areaCurve.GetEndPoint(1).Z + ")";

            }
            break;
        default:
            break;
    }

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

4.4.1.5 *Slabs*

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:

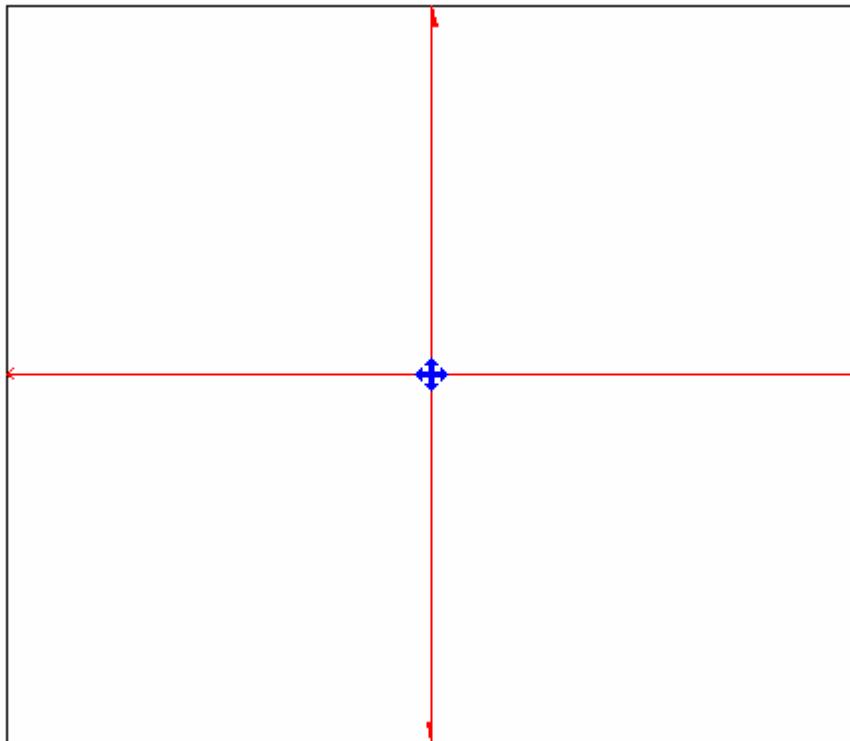


Figure 157: Slab span directions

When using `Floor.Create()` 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:

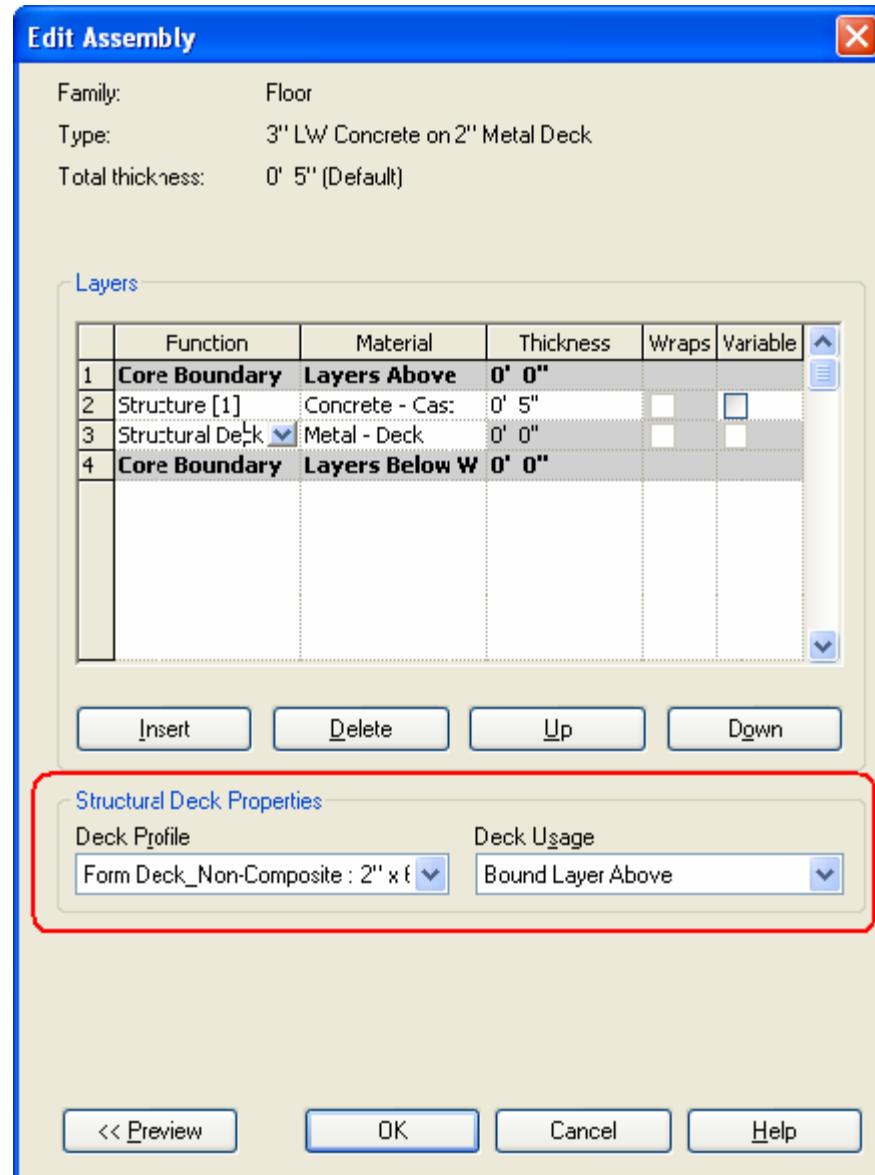


Figure 158: Floor CompoundStructuralLayer properties

4.4.2 Analytical Model

In structural engineering, an analytical model is the engineering description of a structural physical model. The Analytical Model is an essential part of BIM data and is subject to collaborative workflows within engineering teams and across project teams. The Analytical Model implementation in Revit enables engineers to:

- Have freedom of analytical modeling to reflect their individual design decisions with respect to structural elements and buildings/structures seen as whole systems.
- Analytically represent any types of structure

- Create consistent Analytical Models enabling structural analysis jobs from Revit models
- Perform full bi-directional workflow with analysis software and capture model modifications made there
- Preserve Analytical Model from unexpected changes if needed
- Create multiple analytical models reflecting diverse analysis types and configurations
- Replicate the ease of analytical modeling like in dedicated structural analysis software, combined with the power of parametric and collaboration enabling BIM platform.

AnalyticalElement

AnalyticalElement represents the base class for all analytical objects.

- Transform GetTransform() - Returns the transform which reflects Analytical Element orientation.
- AnalyzeAs AnalyzeAs - This represents the Analyze As parameter assigned to Analytical Element.
- Reference GetReference(ArchitecturalModelSelector selector) - Returns a reference to a given curve within the Analytical Element.
- ElementId MaterialId - Defines the Material Id for the Analytical Element.

AnalyticalMember

AnalyticalMember represents a linear element in the structural analytical model.

- AnalyticalMember Create(Document document, Curve curve) - Method which creates a new instance of an Analytical Member within the project
- AnalyticalStructuralRole StructuralRole - The structural role assigned to the Analytical Member
- Curve GetCurve() - Returns the curve of the Analytical Member.
- void SetCurve(Curve curve) - Sets the curve for the Analytical Member. This method disconnects elements from other analytical elements (if the end nodes are in the same position). If the user wants to move the corner, and keep the connection, there are other ways for achieving that like ElementTransformUtils.moveElements.
- bool IsValidCurve(Curve curve) - Verifies if the curve is valid for an Analytical Member
- void FlipCurve() - Flips the ends of the Analytical Member
- StructuralSectionShape StructuralSectionShape - The structural section shape of the Analytical Member (read only)
- ElementId SectionType - The id of the type from the structural Family assigned to the Analytical Member
- double CrossSectionRotation - Cross section rotation of the Analytical Member

AnalyticalPanel

AnalyticalPanel represents a surface in the structural analytical model.

- AnalyticalPanel Create(Document document, CurveLoop curveLoop) - Method which creates a new instance of an Analytical Panel within the project.
- CurveLoop GetOuterContour() - Returns the Curve Loop that defines the geometry of the Analytical Surface element
- bool IsCurveLoopValid(CurveLoop profile) - Checks if curve loop is valid for Analytical Panel
- To modify Analytical Panel geometry, users should use SketchEditScope framework.
- void StartWithNewSketch(ElementId elementId) - Starts a sketch edit mode for an element which, at this moment, doesn't have a sketch. Another way of editing geometry is SetOuterContour(CurveLoop outerContour) - Sets the Curve Loop that defines the geometry of the Analytical Surface element Like for AnalyticalMember, setting the contour for analytical panel will break the connection with other analytical elements. If the user wants to move the corner, and keep the connection, there are other ways for achieving that like ElementTransformUtils.MoveElements.
- `ISet<ElementId> GetAnalyticalOpeningsIds()` - Returns the Analytical Openings Ids of the Analytical Panel
- ElementId SketchId - Sketch associated to this Revit element
- AnalyticalStructuralRole StructuralRole - Structural role assigned to the Analytical Panel

AnalyticalOpening

AnalyticalOpening is an element which represents an opening in an Analytical Panel.

- AnalyticalOpening Create(Document doc, CurveLoop curveLoop, ElementId panelId) - Method which creates a new instance of an Analytical Opening within the project
- CurveLoop GetOuterContour() - Returns the Curve Loop that defines the geometry of the Analytical Surface element
- bool IsCurveLoopValidForAnalyticalOpening(CurveLoop loop, Document document, ElementId panelId) - Checks if curve loop is valid for Analytical Opening To modify Analytical Opening geometry, you should use SketchEditScope framework. Another way to modify Analytical Opening geometry is: void SetOuterContour(CurveLoop outerContour) - Sets the Curve Loop that defines the geometry of the Analytical Surface element.
- ElementId PanelId - ElementId of the host Analytical Panel.
- ElementId SketchId - Sketch associated to this Revit element

AnalyticalToPhysicalAssociationManager

AnalyticalToPhysicalAssociationManager manages the associations between analytical and physical elements. In the past solution, the elements itself knew about each other and the user had no control over it (the association could not be modified). With this new approach, the association can be edited. Revit supports 1-1 association and elements cannot be part of multiple associations at the same time.

- AnalyticalToPhysicalAssociationManager
GetAnalyticalToPhysicalAssociationManager(Document doc) Returns the AnalyticalToPhysicalAssociationManager for this document
- void AddAssociation(ElementId analyticalElementId, ElementId physicalElementId) - Adds a new association between an analytical element and a physical element.

- void RemoveAssociation(ElementId elementId) - This method will remove the association for the element with the given ElementId.
- ElementId GetAssociatedElementId(ElementId elementId) - Returns id of the element which is associated with the given ElementId.
- bool HasAssociation(ElementId id) - Verifies if the element has already defined an association

AnalyticalNodeData

The AnalyticalNodeData class holds information about connection status of analytical nodes.

- AnalyticalNodeData GetAnalyticalNodeData(Element element) - Returns AnalyticalNodeData associated with this element, if it exists.
- AnalyticalNodeConnectionStatus GetConnectionStatus() - Returns the Connections Status for an Analytical Node.

Loads

All loads in the analytical model have a host. For each type of load - point, line, area - there are two options:

- "Constrained on host"
 - The load is created for the entire shape of the host analytical element
 - The load is always hosted on the analytical element
 - The load follows the host analytical element shape and adjust its shape accordingly
- "Custom"
 - The load can be placed anywhere on the host analytical element
 - The load is always hosted on the analytical element
 - The load moves with the host but keeps its shape

Methods to create loads and validate their hosts are:

Load Base:

- LoadBase.IsConstrainedOnHost - checks if the load is constrained on host (is created on the entire shape of the host).

Line Load:

- LineLoad.Create(Document document, ElementId hostEleId, XYZ forceVector1, XYZ momentVector1, LineLoadType symbol) - Creates a new hosted line load within the project.
- LineLoad.Create(Document document, ElementId hostEleId, int curveIndex, XYZ forceVector1, XYZ momentVector1, Structure.LineLoadType symbol) - Creates a new hosted line load within the project.
- LineLoad.Create(Document document, ElementId hostEleId, Curve curve, XYZ forceVector1, XYZ momentVector1, LineLoadType symbol) - Creates a new custom line load within the project.

- `LineLoad.IsValidHostId(Document doc, ElementId hostEleId)` - Indicates if the provided element id can host line loads.

Area Load:

- `AreaLoad.Create(Document doc, ElementId hostEleId, XYZ forceVector1, AreaLoadType symbol)` - Creates a new hosted area load within the project.
- `AreaLoad.Create(Document document, ElementId hostEleId, IList<CurveLoop> loops, XYZ forceVector, AreaLoadType symbol)` - Creates a new custom area load within the project.
- `AreaLoad.Create(Document document, ElementId hostEleId, IList<CurveLoop> loops, IList<XYZ> forceVectors, IList<int> refPointCurveIndexes, IList<int> refPointCurveEnds, AreaLoadType symbol)` - Creates a new custom area load within the project.
- `AreaLoad.IsValidHostId(Document doc, ElementId hostEleId)` - Indicates if the provided element id can host area loads.
- `AreaLoad.AreCurveLoopsValid(IList<CurveLoop> loops)` - Checks if curve loops are valid for creating an area load.
- `AreaLoad.AreCurveLoopsInsideHostBoundaries(Document doc, ElementId hostId, IList<CurveLoop> loops)` - Checks if contour loops are inside host boundaries.

Point Loads:

- `PointLoad.Create(Document doc, ElementId hostEleId, AnalyticalElementSelector selector, XYZ forceVector, XYZ momentVector, PointLoadType symbol)` - Creates a new hosted point load within the project.
- `PointLoad.Create(Document document, ElementId hostEleId, XYZ point, XYZ forceVector, XYZ momentVector, PointLoadType symbol)` - Creates a new custom hosted point load within the project using data at point.
- `PointLoad.IsPointInsideHostBoundaries(Document pDoc, ElementId hostId, XYZ point)` - Indicates if the point is inside panel boundaries or is on the member's curve.
- `PointLoad.IsValidHostId(Document doc, ElementId hostEleId)`

4.4.3 Loads

The following sections identify load settings and discuss load limitation guidelines.

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:

Table 60 Load Case Properties and Parameters

Property	BuiltinParameter
Case Number	LOAD_CASE_NUMBER
Nature	LOAD_CASE_NATURE
Category	LOAD_CASE_CATEGORY

The LOAD_CASE_CATEGORY parameter returns an ElementId. The following table identifies the mapping between Category and ElementId Value.

Table 61: Load Case Category

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
Accidental Loads	OST_LoadCasesAccidental
Temperature Loads	OST_LoadCasesTemperature
Seismic Loads	OST_LoadCasesSeismic

Creating loads and load combinations

The following classes have one or more static Create() methods to create corresponding classes:

- LoadUsage
- LoadNature
- LoadCase
- LoadCombination.

- PointLoad
- LineLoad
- AreaLoad

Point, area, and line loads can be created with or without a host element. Each of these classes has a `isValidHostId` method that indicates if a specific element can host that type of load.

Because they are all Element subclasses, they can be deleted using `Document.Delete()`.

Load Combinations are created via the static method `LoadCombination.Create()` which has two overloads. The first takes only a reference to the document in which you want to create the load combination and the string for the name of the new combination. The second takes these arguments plus the `LoadCombinationType` and `LoadCombinationState`. The `LoadCombinationType` can be either `Combination`, for a straight load combination, or `Envelope`, for an envelope of the effects of several load cases or combinations.

The `LoadCombinationState` can be either `Serviceability` or `Ultimate`. Use `Serviceability` if the load combination represents a service load level on the structure. This is typically used in design or checking of member deflections or other serviceability criteria such as Allowable Stress Design methodologies. Use `Ultimate` if the load combination represents an ultimate load state or a factored load state on the structure typically used in Load Resistance Factor Design methodologies.

After a `LoadCombination` is created, you will need to fill it with `LoadComponents` which comprise the load combination and their factors. `LoadComponents` are added to the `LoadCombination` by calling `LoadCombination.SetComponents()` with a list of the components as shown in this code snippet below.

Note: Make sure that the list of components does not refer to itself.

The following example demonstrates how to create a Load Combination and how to find or create Load Cases and Load Natures to use to set the components of the Load Combination.

Code Region: Creating a new LoadCombination

```
LoadCombination CreateLoadCombinationLoadCaseLoadUsageLoadNatureAndLoadCompon
ent(Document document)

{
    // Create a new load combination

    LoadCombination loadCombination = LoadCombination.Create(document, "DL1 +
    RAIN1", LoadCombinationType.Combination, LoadCombinationState.Ultimate);

    if (loadCombination == null)

        throw new Exception("Create new load combination failed.");
```

```
// Get all existing LoadCase

FilteredElementCollector collector = new FilteredElementCollector(document);

ICollection<Element> collection = collector.OfClass(typeof(LoadCase)).ToElements();



// Find LoadCase "DL1"

LoadCase case1 = null;

foreach (Element e in collection)

{

    LoadCase loadCase = e as LoadCase;

    if (loadCase.Name == "DL1")

    {

        case1 = loadCase;

        break;

    }

}

// Get all existing LoadNature

collector = new FilteredElementCollector(document);

collection = collector.OfClass(typeof(LoadNature)).ToElements();



// Find LoadNature "Dead"

LoadNature nature1 = null;

foreach (Element e in collection)

{
```

```
LoadNature loadNature = e as LoadNature;

if (loadNature.Name == "Dead")

{

    nature1 = loadNature;

    break;

}

}

// Create LoadNature "Dead" if not exist

if (nature1 == null)

    nature1 = LoadNature.Create(document, "Dead");

// Create LoadCase "DL1" if not exist

if (case1 == null)

    case1 = LoadCase.Create(document, "DL1", nature1.Id, LoadCaseCategory.Dead);

// Create LoadNature "Rain"

LoadNature nature2 = LoadNature.Create(document, "Rain");

if (nature2 == null)

    throw new Exception("Create new load nature failed.");

// Create LoadCase "RAIN1"

LoadCase case2 = LoadCase.Create(document, "RAIN1", nature2.Id, LoadCaseCategory.Snow);

if (case1 == null || case2 == null)

    throw new Exception("Create new load case failed.");
```

```
// Create LoadComponents - they consist of LoadCases or nested LoadCombination and Factors

List<LoadComponent> components = new List<LoadComponent>();

components.Add(new LoadComponent(case1.Id, 2.0));
components.Add(new LoadComponent(case2.Id, 1.5));

// Add components to combination
loadCombination.SetComponents(components);

// Create LoadUsages

LoadUsage usage1 = LoadUsage.Create(document, "Frequent");
LoadUsage usage2 = LoadUsage.Create(document, "Rare");

if (usage1 == null || usage2 == null)
    throw new Exception("Create new load usage failed.");

// Add load usages to combination
loadCombination.SetUsageId(new List<ElementId>() {usage1.Id, usage2.Id});

// Give the user some information
TaskDialog.Show("Revit", string.Format("Load Combination ID='{0}' created successfully.", loadCombination.Id.Value));

return loadCombination;
}
```

You may also modify the cases, components, natures, etc by using LoadCombination.GetComponents(), making modifications and then calling LoadCombination.SetComponents() again. LoadUsages for the LoadCombination can be modified by calling LoadCombination.GetUsageIds() to get the list of LoadUsage Ids, modifying the list, and calling SetUsageIds() again. The following code sample demonstrates how to modify an existing LoadCombination.

Code Region: Modify a load combination

```
void ModifyLoadCombinationLoadCaseLoadUsageLoadNatureAndLoadComponent(Document document, LoadCombination loadCombination)

{
    // Change name of LoadCombination
    loadCombination.Name = "DL2 + RAIN1";

    // Get any LoadCase from combination
    // Combination can have assigned LoadCase or other (nested) LoadCombination so we need to filter out any LoadCombination
    LoadCase case1 = null;

    IList<ElementId> caseAndCombinationIds = loadCombination.GetCaseAndCombinationIds();

    foreach (ElementId id in caseAndCombinationIds)
    {
        Element element = document.GetElement(id);

        if (element is LoadCase)
        {
            case1 = (LoadCase)element;
            break;
        }
        else if (element is LoadCombination)
        {
            // ...
        }
    }

    // ...
}
```

```
        continue;

    }

}

if (case1 == null)
    throw new Exception("Can't get LoadCase.");

// Change case name and number
case1.Name = "DL2";
if (LoadCase.IsNumberUnique(document, 3))
{
    case1.Number = 3;
}

// Create load nature
LoadNature liveNature = LoadNature.Create(document, "Dead nature");
if (liveNature == null)
    throw new Exception("Create new load nature failed.");

// Change nature category and ID for case
case1.SubcategoryId = new ElementId(BuiltInCategory.OST_LoadCasesDead);
case1.NatureId = liveNature.Id;

//Change factor for case1
IList<LoadComponent> components = loadCombination.GetComponents();
foreach (LoadComponent loadComponent in components)
```

```

    {

        if (loadComponent.LoadCaseOrCombinationId == case1.Id)

        {

            loadComponent.Factor = 3.0;

        }

    }

    loadCombination.SetComponents(components);

    // Remove one usage from combination

    IList<ElementId> usages = loadCombination.GetUsageIds();

    usages.RemoveAt(0);

    loadCombination.SetUsageIds(usages);

    // Give the user some information

    TaskDialog.Show("Revit", string.Format("Load Combination ID='{0}' modified successfully.", loadCombination.Id.Value));

}

```

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, and then copy the corresponding properties and parameters from an existing `LoadCase` (or `LoadNature`).

The following is a minimum sample code to demonstrate the creation of a point load:

Code Region: New PointLoad

```

public void CreatePointLoad(Document document, ElementId hostId)
{

```

```

// Define the location at which the PointLoad is applied.

XYZ point = new XYZ(0, 0, 4);

// Define the 3d force.

XYZ force = new XYZ(0, 0, -1);

// Define the 3d moment.

XYZ moment = new XYZ(0, 0, 0);

PointLoad pointLoad = PointLoad.Create(document, hostId, point, force, moment, null);

}

```

4.4.4 Analysis Link

With Revit, a structural 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 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/en/download/rfem_revit_en.pdf)
- Advance Design, VisualDesign, Arche, EffeL and SuperSTRESS from GRAITEC (www.graitec.com/En/revit.asp)
- Scia Engineer from Nemetschek (www.scia.net/en/software/product-selection/scia-engineer)
- 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/products/product-line/structural-analysis-software/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)
- SPACE GASS from SPACE GASS (www.spacegass.com/revit)

- Robot Structural Analysis Professional from Autodesk (www.autodesk.com/products/robot-structural-analysis)

The key to linking Revit 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 the Revit API and the other for the structural analysis application, shown as follows:

Table 62: Revit and Analysis Application Object Mapping

Revit 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:		

...

4.4.5 Analytical Links

An analytical link is an element connecting 2 separate analytical nodes, which has properties such as fixity state. Analytical links can be created automatically by Revit from analytical beams to analytical columns during modeling based on certain rules. And they can also be created manually, both in the Revit UI and using the Revit API.

In the Revit API, an analytical link is represented by the `AnalyticalLink` class. Fixity values are available from its associated `AnalyticalLinkType`.

The example below demonstrates how to read all of the `AnalyticalLinks` in the document and displays a TaskDialog summarizing the number of automatically generated and manually created `AnalyticalLinks`.

Code Region: Reading AnalyticalLinks

```
public void ReadAnalyticalLinks(Document document)
{
    FilteredElementCollector collectorAnalyticalLinks = new FilteredElementCollector(document);
    collectorAnalyticalLinks.OfClass(typeof(AnalyticalLink));

    IEnumerable<AnalyticalLink> alinks = collectorAnalyticalLinks.ToElements()
        .Cast<AnalyticalLink>();

    int nAutoGeneratedLinks = 0;
    int nManualLinks = 0;

    foreach (AnalyticalLink alink in alinks)
    {
        if (alink.IsAutoGenerated() == true)
            nAutoGeneratedLinks++;
        else
    }
}
```

```

        nManualLinks++;
    }

    string msg = "Auto-generated AnalyticalLinks: " + nAutoGeneratedLinks;
    msg += "\nManually created AnalyticalLinks: " + nManualLinks;
    TaskDialog.Show("AnalyticalLinks", msg);
}

```

The static method AnalyticalLink.Create() creates a new analytical link. Rather than connecting the two elements directly, the connection is created between two Hubs. The Hub class represents a connection between two or more Autodesk Revit Elements.

The following example creates a new analytical link between two selected FamilyInstance objects. It uses a filter to find all Hubs in the model and then the GetHub() method searches the hubs to find one which references the Id of the AnalyticalElement of each FamilyInstance.

Code Region: Creating a new AnalyticalLink

```

public void CreateLink(Document doc, FamilyInstance fi1, FamilyInstance fi2)
{
    FilteredElementCollector hubCollector = new FilteredElementCollector(doc);
    hubCollector.OfClass(typeof(Hub)); //Get all hubs
    ICollection<Element> allHubs = hubCollector.ToElements();
    FilteredElementCollector linktypeCollector = new FilteredElementCollector(doc);
    linktypeCollector.OfClass(typeof(AnalyticalLinkType));
    ElementId firstLinkType = linktypeCollector.ToElementIds().First(); //Get the first analytical link type.

    // Get hub Ids from two selected family instance items
    ElementId startHubId = GetHub(AnalyticalToPhysicalAssociationManager.GetAnalyticalToPhysicalAssociationManager(doc)

```

```
.GetAssociatedElementId(fi1.Id), allHubs);

    ElementId endHubId = GetHub(AalyticalToPhysicalAssociationManager.GetAalyticalToPhysicalAssociationManager(doc)

.GetAssociatedElementId(fi2.Id), allHubs);

    Transaction tran = new Transaction(doc, "Create Link");

    tran.Start();

    //Create a link between these two hubs.

    AnalyticalLink createdLink = AnalyticalLink.Create(doc, firstLinkType, startHubId, endHubId);

    tran.Commit();

}

//Get the first Hub on a given AnalyticalElement element

private ElementId GetHub(ElementId hostId, ICollection<Element> allHubs)

{

    foreach (Element ehub in allHubs)

    {

        Hub hub = ehub as Hub;

        ConnectorManager manager = hub.GetHubConnectorManager();

        ConnectorSet connectors = manager.Connectors;

        foreach (Connector connector in connectors)

        {

            ConnectorSet refConnectors = connector.AllRefs;

            foreach (Connector refConnector in refConnectors)

            {

                if (refConnector.Owner.Id == hostId)

                {
```

```

        return hub.Id;
    }

}

}

return ElementId.InvalidElementId;
}

```

4.4.6 Steel Fabrication

Linking between Revit elements and steel fabrication elements

`Autodesk.Revit.DB.Steel.SteelElementProperties` attaches steel fabrication information to Revit elements. Elements which can have fabrication information include:

- `FamilyInstance` elements (structural beams and columns)
- `StructuralConnectionHandler` elements
- Specific steel connection elements (bolts, anchors, plates, etc). These connection elements will be of type `Element` but with categories related to structural connections, for example:
 - `OST_StructConnectionWelds`
 - `OST_StructConnectionHoles`
 - `OST_StructConnectionShearStuds`
 - `OST_StructConnectionBolts`
 - `OST_StructConnectionAnchors`
 - `OST_StructConnectionPlates`
- Some concrete elements (walls, floors, concrete beams) when they are used as input elements to creation of steel connections.

Use `SteelElementProperties.GetSteelElementProperties()` to obtain the properties if they exist.

The properties contain `SteelElementProperties.UniqueID` which is the id of the object in fabrication terms, and can be used to determine the Steel Core element corresponding to this Revit element, for use with the Advance Steel API.

You can also look up the id for a Revit element using the static method `SteelElementProperties.GetFabricationUniqueId()`.

For Revit elements which do not currently have a fabrication link, it can be added using: `SteelElementProperties.AddFabricationInformationForRevitElements()`

If you have a fabrication id, you can lookup the corresponding Revit element using: SteelElementProperties.GetReference(). This may return a reference to an element or a subelement.

Custom steel connections

StructuralConnectionHandler.CreateGenericConnection() creates a custom StructuralConnectionHandler along with its associated StructuralConnectionHandlerType. Input elements should include structural members and steel connection members, and at least one StructuralConnectionHandler representing the generic connection to replace with the new detailed custom connection.

The methods:

- StructuralConnectionHandlerType.AddElementsToCustomConnection()
- StructuralConnectionHandlerType.RemoveMainSubelementsFromCustomConnection()

provide support for adding or removing steel connection elements in a custom connection.

The properties:

- StructuralConnectionHandler.IsCustom
- StructuralConnectionHandler.IsDetailed
- StructuralConnectionHandlerType.IsCustom
- StructuralConnectionHandlerType.IsDetailed

provide read access to information about the structural connection handler elements.

4.5 Toposolid

The `Autodesk.Revit.DB.Toposolid` class is a solid element that has an assigned type and basic parameters and is eligible for Boolean operations such as cutting with Mass and In-place components. This class is closely related to floors and its points are editable using the same SlabShapeEditor interfaces. In addition, if the Toposolid has a Sketch boundary it can be accessed via the standard SketchEditScope capabilities.

ToposolidType

`Autodesk.Revit.DB.ToposolidType` represents the type of a Toposolid in Autodesk Revit. This class inherits from HostObjAttributes and provides read/write access to the CompoundStructure of the ToposolidType. The `GetCountourSetting()` method provides access to the object that defines the elevations where contours will be drawn.

Creation

`Toposolid.Create` can be used with one of three overloads that allow you to specify either or both:

- An array of planar curve loops that represent the profiles of the toposolid
- An array of points that used to construct the top face of the toposolid

`Toposolid.CreateFromTopographySurface` creates a toposolid element from a host TopographySurface and toposolid sub-divisions from its subregions

Linked Topography

- TopographyLinkType represents a site file brought into the current Revit model as a link.
- TopographyLinkType.Reload() reloads the TopographyLinkType from its current location.

5 Advanced Topics

5.1 Analysis

Related Information

- [Green Building Studio API](#)

5.1.1 Energy Data

The EnergyDataSettings object contains settings for gbXML Export and Heating and Cooling Load Calculations and project level settings for Conceptual Energy Analysis.

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);
    }
}

```

5.1.2 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 `AnalysisResultSchema` class contains all information about one analysis result, such as a description and the names and multipliers of all units for result visualization. Multiple `AnalysisResultSchemas` can be registered with the `SpatialFieldManager`.

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.

Framework Overview

The Analysis Visualization Framework (AVF) enables the display of external calculations in the context of a Revit model. API users do not have to know the details of internal Revit drawing mechanism; all they need to do is (a) to populate correctly data containers with calculated numbers representing the calculated properties and (b) to specify the Revit View and style of the data display. Calculated numbers should represent a certain Spatial Field (function) defined on the Field Domain with the corresponding Field Values.

Field Domain

Field Domain is an array of points that can be defined in any of the following layouts:

- along a curve (one-dimensional), API class `FieldDomainPointsByParameter`;
- on a surface (two-dimensional), API class `FieldDomainPointsByUV`;
- in space (three-dimensional), API class `FieldDomainPointsByXYZ`.

These classes have a common base class `FieldDomainPoints`.

Field Values

`Field Values` is an array of scalar or vector values calculated at each domain point. The corresponding API class is `FieldValues`. It can be constructed by taking an input array of `ValueAtPoint` or `VectorAtPoint` objects. The size of these arrays must be equal to the size of array of points in the corresponding `FieldDomainPoints` object.

Each `ValueAtPoint` or `VectorAtPoint` object can contain one or more measurements calculated at the same point in Field Domain. The restriction is that the number of measurements must be the same for each point in Field Domain (see Figure 1 below).

Each measurement can have a name and a description (see `SpatialFieldManager.SetMeasurementNames()` and `SpatialFieldManager.SetMeasurementDescriptions()`). Only one measurement is displayed at a time in Revit View (see `SpatialFieldManager.CurrentMeasurement` property).

Spatial Field Primitive

`Spatial Field Primitive` represents the pairing of `FieldDomainPoints` and its corresponding `FieldValues` (see Figure 1 below). Each primitive:

- can be associated with a Reference to a Curve or Surface in Revit model geometry (corresponding `FieldDomainPointsByParameter` or `FieldDomainPointsByUV` must be used);
- can be associated with a Curve or Surface (constructed on demand by API user) that are not part of Revit geometry (corresponding `FieldDomainPointsByParameter` or `FieldDomainPointsByUV` must be used);
- can represent free points in space (corresponding `FieldDomainPointsByXYZ` must be used);

Field Domain	Point [1]	Point [2]	..	Point [N-1]	Point [N]
Field Values	ValueAtPoint [1]	ValueAtPoint[2]	..	ValueAtPoint[N-1]	ValueAtPoint[N]
measureme nt [1]	Value [1,1]	Value [2,1]	..	Value [N-1,1]	Value [N,1]
measureme nt [2]	Value [1,2]	Value [2,2]	..	Value [N-1,2]	Value [N,2]
...

Field Domain	Point [1]	Point [2]	..	Point [N-1]	Point [N]
measureme nt [i]			..		
currently displayed	Value [1,i]	Value [2,i]	..	Value [N-1,i]	Value [N,i]
...
measureme nt [M-1]	Value [1,M- 1]	Value [2,M-1]	..	Value [N-1,M- 1]	Value [N,M-1]
measureme nt [M]	Value [1,M]	Value [2,M]	..	Value [N-1,M]	Value [N,M]

Figure 1. Spatial Field Primitive with N domain points and M measurements.

Each Spatial Field (can also be called a Result) consists of a set of Spatial Field Primitives (see `SpatialFieldManager.AddSpatialFieldPrimitive()` and `SpatialFieldManager.UpdateSpatialFieldPrimitive()`).

Result

Result represents a set of Spatial Field Primitives corresponding to a particular Spatial Field. Each Result can have a name, description, units, etc. (see `AnalysisResultSchema`). A number of Results can be displayed in a particular Revit View (see `SpatialFieldManager.RegisterResult()` and `SpatialFieldManagerGetRegisteredResults()`). All Results in the same View must have the same number of measurements (see `SpatialFieldManager.NumberOfMeasurements` property).

5.1.2.1 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 identify the primitive to remove it (`RemoveSpatialFieldPrimitive()`) or to modify the primitive (`UpdateSpatialFieldPrimitive()`).

Note that the `AddSpatialFieldPrimitive()` method creates an empty analysis results primitive. `UpdateSpatialFieldPrimitive()` must be called in order populate the analysis results data with points and values as shown in the [Creating analysis results data](#) section.

The `UpdateSpatialFieldPrimitive()` method requires the unique index of an `AnalysisResultSchema` that has been registered with the `SpatialFieldManager`. An `AnalysisResultSchema` holds information about an analysis results, such as a name, description and the names and multipliers of all units for result visualization. The following example demonstrates how to create a new `AnalysisResultSchema` and set its units.

Code Region: AnalysisResultsSchema

```
private void CreateSchema()
{
    IList<string> unitNames = new List<string>();
    unitNames.Add("Feet");
    unitNames.Add("Inches");
    IList<double> multipliers = new List<double>();
    multipliers.Add(1);
    multipliers.Add(12);
```

```

    AnalysisResultSchema resultSchema = new AnalysisResultSchema("Schema Name",
", "Description");

resultSchema.SetUnits(unitNames, multipliers);

}

```

Once the AnalysisResultschema is configured, it needs to be registered using the SpatialFieldManager.RegisterResult() method, which will return a unique index for the result. Use GetResultSchema() and SetResultSchema() using this unique index to get and change the result after it has been registered.

5.1.2.2 *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

```

public void CreateAnalysisResults(UIDocument uidoc)
{
    Document doc = uidoc.Document;

    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 = doc.GetElement(reference).GetGeometryObjectFromReference(reference) 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);

AnalysisResultSchema resultSchema = new AnalysisResultSchema("Schema
Name", "Description");

int schemaIndex = sfm.RegisterResult(resultSchema);

sfm.UpdateSpatialFieldPrimitive(idx, pnts, vals, schemaIndex);

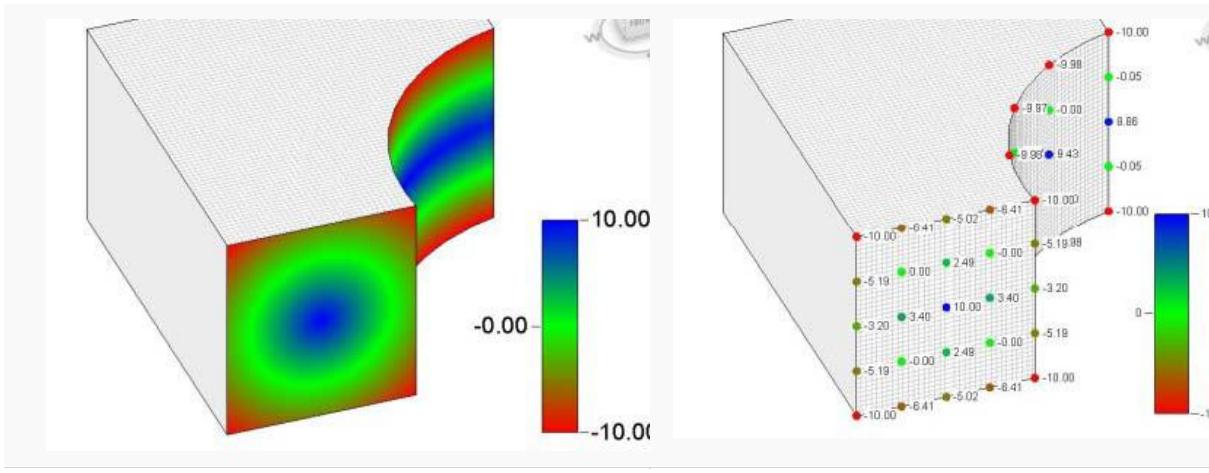
}

```

5.1.2.3 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, a markers with text style, a deformed shape style, diagram style or vector style. For any 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.



The following example creates a new colored surface 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

```
public void SetDisplayStyle(Document doc)
{
    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.TextTypeId = textType.Id;
}

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;

}

```

5.1.2.4 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.

5.1.3 Detailed Energy Analysis Model

The Autodesk.Revit.DB.Analysis namespace includes several classes to obtain and analyze the contents of a project's detailed energy analysis model.

The Export to gbXML and the Heating and Cooling Loads features produce an analytical thermal model from the physical model of a building. The analytical thermal model is composed of spaces, zones and planar surfaces that represent the actual volumetric elements of the building.

The classes related to the detailed energy analysis model are:

- EnergyAnalysisDetailModel
- EnergyAnalysisDetailModelOptions
- EnergyAnalysisOpening
- EnergyAnalysisSpace
- EnergyAnalysisSurface
- Polyloop

Energy analysis model creation

Use the static method `EnergyAnalysisDetailModel.Create()` to create and populate the energy analysis model. The `EnergyAnaysisDetailModel` is stored as an element in the Revit model, and

thus the `EnergyAnalysisDetailModel.Create()` method requires there to be an open transaction. The generated model is always returned in world coordinates, but the method `TransformModel()` transforms all surfaces in the model according to ground plane, shared coordinates and true north.

If an energy analysis model is already created, the static method `EnergyAnalysisDetailModel.GetMainEnergyAnalysisDetailModel()` returns the main `EnergyAnalysisDetailModel` contained in the given document (or null if none has been created). The energy analysis detail model can be displayed in associated views.

Set the appropriate options using the `EnergyAnalysisDetailModelOptions` class.

The options available when creating the energy analysis detail model include:

- The level of computation for energy analysis model - `NotComputed`, `FirstLevelBoundaries`, meaning analytical spaces and zones, `SecondLevelBoundaries`, meaning analytical surfaces, or `Final`, meaning constructions, schedules, and non-graphical data
- Whether the energy model is based on rooms/spaces or building elements
- Whether mullions should be exported as shading surfaces
- Whether shading surfaces will be included
- Whether to simplify curtain systems - When true, a single large window/opening will be exported for a curtain wall/system regardless of the number of panels in the system

The `EnergyAnalysisDetailModelOptions.EnergyModelType` property can be set to `SpatialElement` (where the energy model is based on rooms or spaces) or `BuildingElement` (where the energy model is based on analysis of building element volumes). However, note that the generated energy model is also affected by settings in `EnergyDataSettings`, including the `EnergyDataSettings.AnalysisType` property. If this property is set to `AnalysisMode.ConceptualMassesAndBuildingElements`, the `EnergyAnalysisDetailModel` will use the combination of conceptual masses and building elements.

The following example creates a new energy analysis detailed model from the physical model then displays the originating element for each surface of each space in the model.

Code Region: Energy Analysis Detail Model

```
public void GetThermalModelData(Document doc)
{
    // Collect space and surface data from the building's analytical thermal
    // model

    EnergyAnalysisDetailModelOptions options = new EnergyAnalysisDetailModelOptions();
```

```
    options.Tier = EnergyAnalysisDetailModelTier.Final; // include constructions, schedules, and non-graphical data in the computation of the energy analysis model

    options.EnergyModelType = EnergyModelType.SpatialElement; // Energy model based on rooms or spaces

    EnergyAnalysisDetailModel eadm = EnergyAnalysisDetailModel.Create(doc, options); // Create a new energy analysis detailed model from the physical model

    IList<EnergyAnalysisSpace> spaces = eadm.GetAnalyticalSpaces();

    StringBuilder builder = new StringBuilder();

    builder.AppendLine("Spaces: " + spaces.Count);

    foreach (EnergyAnalysisSpace space in spaces)

    {

        SpatialElement spatialElement = doc.GetElement(space.CADObjectUniqueId) as SpatialElement;

        ElementId spatialElementId = spatialElement == null ? ElementId.InvalidElementId : spatialElement.Id;

        builder.AppendLine("    >>> " + space.SpaceName + " related to " + spatialElementId);

        IList<EnergyAnalysisSurface> surfaces = space.GetAnalyticalSurfaces();

        builder.AppendLine("        has " + surfaces.Count + " surfaces.");

        foreach (EnergyAnalysisSurface surface in surfaces)

        {

            builder.AppendLine("            +++ Surface from " + surface.OrientingElementDescription);

        }

    }

    TaskDialog.Show("EAM", builder.ToString());

}
```

After creating the EnergyAnalysisDetailModel, the spaces, openings and surfaces associated with it can be retrieved with the GetAnalyticalOpenings(), GetAnalyticalSpaces(), GetAnalyticalShadingSurfaces() and GetAnalyticalSurfaces() methods.

It is recommended that applications call Document.Delete() on the EnergyAnalysisDetailModel elements that they create when finished accessing the data, but any energy models created after the main energy model will be deleted automatically before document saving or synchronization.

EnergyAnalysisSpace

From an EnergyAnalysisSpace you can retrieve the collection of EnergyAnalysisSurfaces which define an enclosed volume bounded by the center plane of walls and the top plane of roofs and floors. Alternatively, GetClosedShell() retrieves a collection of Polyloops, which are planar polygons, that define an enclosed volume measured by interior bounding surfaces. For two-dimensions, use GetBoundary() which returns a collection of Polyloops representing the 2D boundary of the space that defines an enclosed area measured by interior bounding surfaces.

The EnergyAnalysisSpace class also has a number of properties for accessing information about the analysis space, such as AnalyticalVolume, SpaceName and Area.

EnergyAnalysisSurface

From an EnergyAnalysisSpace you can retrieve the primary analysis space associated with the surface as well as the secondary adjacent analytical space. The GetAnalyticalOpenings() method will retrieve a collection of all analytical openings in the surface. The GetPolyloop() method obtains the planar polygon describing the surface geometry as described in gbXML.

The EnergyAnalysisSurface class has numerous properties to provide more information about the analytical surface, such as Height, Width, Corner (lower-left coordinate for the analytical rectangular geometry viewed from outside), and an originating element description.

Use these properties to identify the originating Revit element's id and name:

- EnergyAnalysisSurface.OriginatingElementId
- EnergyAnalysisSurface.OriginatingElementName

The method EnergyAnalysisSurface.GetConstruction() allows users to get the analytic construction this surface is associated with.

The surface type is available either as an EnergyAnalysisSurfaceType or as a gbXMLSurfaceType. The gbXML surface type attribute is determined by the source element and the number of space adjacencies. Possible types are:

Type	Source element and space adjacencies
Shade	No associated source element and no space adjacencies

Air	No associated source element and at least one space adjacency
ExteriorWall	Source element is a Wall or a Curtain Wall there is one space adjacency
InteriorWall	Source element is a Wall or a Curtain Wall and: there are two space adjacencies or the type Function parameter is set to "Interior" or "CoreShaft"
UndergroundWall	Source element is a Wall or a Curtain Wall and there is one space adjacency and if it is below grade
SlabOnGrade	Source element is a Floor and there is one space adjacency
RaisedFloor	Source element is a Floor and there is one space adjacency and it is above grade
UndergroundSlab	Source element is a Floor and there is one space adjacency and it is below grade
InteriorFloor	Source element is a Floor and: there are two space adjacencies or the type Function parameter is set to "Interior"
Roof	Source element is a Roof or a Ceiling and there is one space adjacency
UndergroundCeiling	Source element is a Roof or a Ceiling and there is one space adjacency and it is below grade
Ceiling	Source element is a Roof or a Ceiling and there are two space adjacencies

Thermal Properties for Analytical Surfaces

Several classes in the Autodesk.Revit.DB.Analysis namespace allow users to retrieve thermal properties and material layers for analytical constructions and window types associated with analytical surfaces in the energy model. All thermal properties on EnergyAnalysisConstruction and EnergyAnalysisMaterial are read-only.

`EnergyAnalysisZone` - Represents the analytical zone. EnergyAnalysisZone is associated with EnergyAnalysisSpace, and each EnergyAnalysisZone may be associated with one or more EnergyAnalysisSpace elements.

`EnergyAnalysisConstruction` - Represents an analytical construction as a composite of layered materials and is generally associated with EnergyAnalysisSurface.

`EnergyAnalysisWindowType` - Represents an analytical construction containing window type data and is associated with `EnergyAnalysisOpening`.

`EnergyAnalysisMaterial` - Represents the description of a material with thermal properties in a composite construction.

The energy analysis enumerated types `gbXMLSurfaceType` and `ConstructionType` have values for a given element that can be retrieved for using the `EnergyAnalysisSurface.Type` and `EnergyAnalysisOpening.Type` properties.

EnergyAnalysisOpening

From an `EnergyAnalysisOpening` you can retrieve the associated parent analytical surface element. The `GetPolyloop()` method returns the opening geometry as a planar polygon.

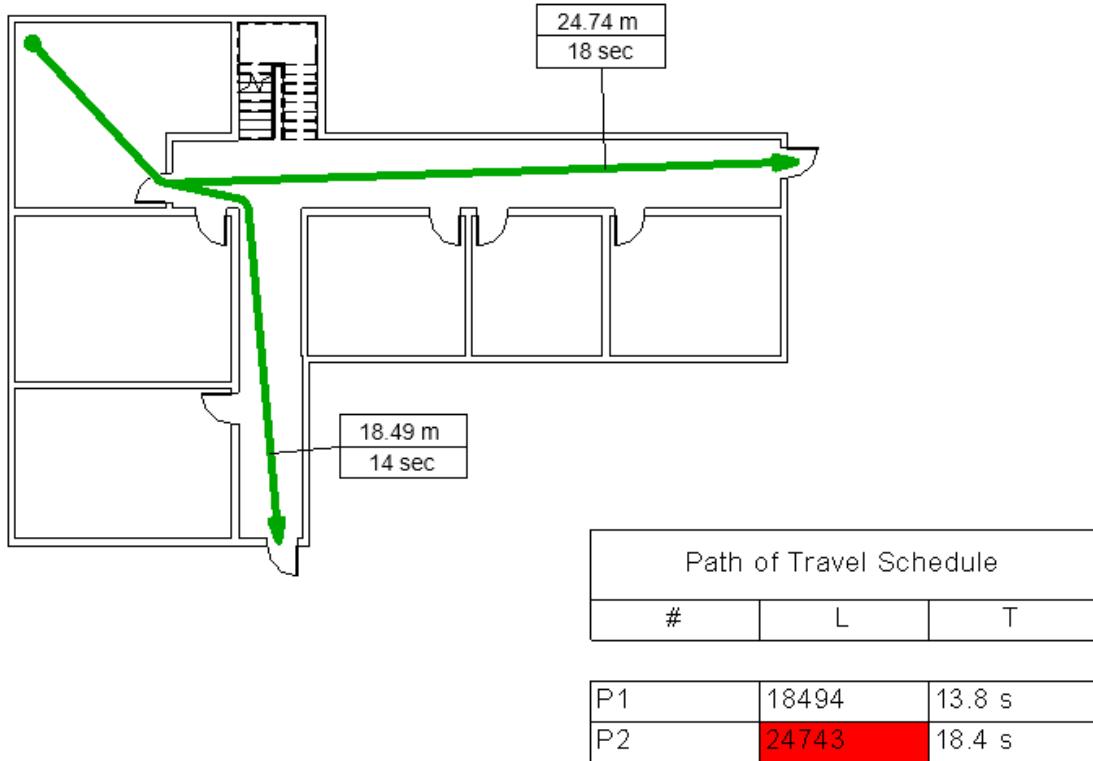
A number of properties are available to obtain information about the analytical opening, such as Height, Width, Corner and OpeningName. Similar as for analytical surfaces, the analytical opening type can be obtained as a simple `EnergyAnalysisOpeningType` enumeration or as a `gbXMLOpeningType` attribute. The type of the opening is based on the family category for the opening and in what element it is contained, as shown in the following table:

Type	Family Category or containing element
<code>OperableWindow</code>	Window
<code>NonSlidingDoor</code>	Door
<code>FixedSkylight</code>	Opening contained in a Roof
<code>FixedWindow</code>	Opening contained in a Curtain Wall Panel
<code>Air</code>	Opening of the category Openings

5.1.4 Path Of Travel

The path of travel element allows you to analyze travel distances and times between 2 selected points in your model. The path of travel is generated based on the model elements acting as obstacles along the path of travel. It avoids contact with model elements in the analysis zone and calculates the shortest distance between the start and end

points.



The class `Autodesk.Revit.DB.Analysis.PathOfTravel` represents this path element.

Members of this class include:

- `PathOfTravel.Create()` - creates a single `PathOfTravel` element given a start and end point.
- `PathOfTravel.CreateMultiple()` - creates multiple `PathOfTravel` elements given arrays of start and end points.
- `PathOfTravel.CreateMapped()` - creates multiple `PathOfTravel` elements by mapping each of a set of start points to each of a set of end points.
- `PathOfTravel.GetCurves()`
- `PathOfTravel.LineStyle`
- `PathOfTravel.PathStart`
- `PathOfTravel.PathMidpoint`
- `PathOfTravel.PathEnd`
- `PathOfTravel.Update()`
- `PathOfTravel.UpdateMultiple()`

The class `Autodesk.Revit.DB.Analysis.RouteAnalysisSettings` represents a settings element which contains project-wide settings for route calculations in plan views. Currently, these settings are only used by the `Autodesk.Revit.DB.Analysis.PathOfTravel` element, but in the future they can be used by any other functionalities which do route calculations.

Members of this class include:

- `RouteAnalysisSettings.EnableIgnoredCategoryIds()` - enable ignoring specified categories of elements during calculations
- `RouteAnalysisSettings.SetIgnoredCategoryIds()` - sets the categories to be ignored during calculations
- `RouteAnalysisSettings.AnalysisZoneTopOffset` - the analysis zone top (an offset above the level of the plan view)
- `RouteAnalysisSettings.AnalysisZoneBottomOffset` - the analysis zone bottom (an offset above the level of the plan view)
- `RouteAnalysisSettings.IgnoreImports` - If true, import instances are ignored by route calculation

Reveal Obstacles mode for Path of Travel

The Reveal Obstacles view mode highlights elements in the plan view when those elements will act as obstacles for the current Path of Travel calculation settings. These methods provide access to read or set if a view is displaying this mode:

- `PathOfTravel.IsInRevealObstaclesMode()`
- `PathOfTravel.SetRevealObstaclesMode()`

Path finding analysis for Path of Travel

`List<XYZ> PathOfTravel.FindStartsOfLongestPathsFromRooms(View view, List<XYZ> destinationPoints)` For a floor plan view, calculates paths from points inside rooms to the closests of the destinations. Returns the start points of the longest path(s). If multiple paths have the same longest length, returns multiple start points. The entire plan is divided in small tiles, and the distance to the closest destination point is calculated for each tile center point. Only tile center points that are located in rooms in the view are taken into account.

`List<XYZ> PathOfTravel.FindEndsOfShortestPaths(View view, List<XYZ> destinationPoints, List<XYZ> startPoints)` For a floor plan view, calculates the paths from each start point to its closest destination and return the path end points. The calculation is done in a floor plan with one or more destinationPoints and one or more startPoints. The shortest path is calculated from each start point to its corresponding closest destination.

`List<List<XYZ>> PathOfTravel.FindShortestPaths(View view, List<XYZ> destinationPoints, List<XYZ> startPoints)` For a floor plan view, calculates paths from each start point to its closest destinations. Returns the path, represented by an array of XYZ points. The calculation is done in a floor plan with one or more destinationPoints and one or more startPoints. The shortest path is calculated from each start point to its closest destination point.

Waypoints

These methods provide access to read and modify the waypoints associated to a particular `PathOfTravel` element. Waypoints force the path of travel calculation to ensure that the path includes each of the specified points, in the order specified, between the start and end points.

- `PathOfTravel.GetWaypoints()`

- `PathOfTravel.InsertWaypoint()`
- `PathOfTravel.SetWaypoint()`
- `PathOfTravel.RemoveWaypoint()`

5.2 Browser Organization

The `BrowserOrganization` class provides read-access to the settings for grouping, sorting, and filtering of items in the project browser.

`BrowserOrganization.AreFiltersSatisfied()` determines if the given element satisfies the filters defined by the browser organization.

Getting the current organization

- `GetCurrentBrowserOrganizationForViews()` gets the `BrowserOrganization` that applies to the Views section of the project browser.
- `GetCurrentBrowserOrganizationForSheets()` gets the `BrowserOrganization` that applies to the Sheets section of the project browser.
- `BrowserOrganization.GetCurrentBrowserOrganizationForSchedules()` gets the `BrowserOrganization` that applies to the Schedules section of the project browser

Sorting

- `SortingOrder` – The sorting order if sorting of items is applicable in the browser.
- `SortingParameterId` – The id of the parameter used to determine the sorting order of items in the browser.

FolderItemInfo

`BrowserOrganization.GetFolderItems(ElementId)` returns a collection of leaf `FolderItemInfo` objects each containing the given element Id.

`FolderItemInfo` contains the `ElementId` and `Name` info for each item in the organization settings of the project browser.

5.3 Commands

The Revit API provides access to existing Revit commands, either located on a tab, the application menu, or right-click menu. The main ways to work with Revit commands using the API is to either replace the existing command implementation or to post a command.

Overriding a Revit command

The `AddInCommandBinding` class can be used to override an existing command in Revit. It has three events related to replacing the existing command implementation.

- **BeforeExecuted**- This read-only event occurs before the associated command executes. An application can react to this event but cannot make changes to documents, or affect the invocation of the command.
- **CanExecute** - Occurs when the associated command initiates a check to determine whether the command can be executed on the command target.
- **Executed** - This event occurs when the associated command executes and is where any overriding implementation should be performed.

To create the commandbinding, call either `UIApplication.CreateAddInCommandBinding()` or `UIControlledApplication.CreateAddInCommandBinding()`. Both methods require a `RevitCommandId` id to identify the command handler you want to replace. The `RevitCommandId` has two static methods for obtaining a command's id:

- **LookupCommandId** - Retrieves the Revit command id with the given id string. To find the command id string, open a session of Revit, invoke the desired command, close Revit, then look in the journal from that session. The "Jrn.Command" entry that was recorded when it was selected will have the string needed for `LookupCommandId()` and will look something like "ID_EDIT_DESIGNOPTIONS".
- **LookupPostableCommandId** - Retrieves the Revit command id using the PostableCommand enumeration. This only works for commands which are postable (discussed in the following section).

The following example, taken from Revit 2014 SDK's `DisableCommand` sample, demonstrates how to create an `AddInCommandBinding` and override the implementation to disable the command with a message to the user.

Code Region: Overriding a command

```
/// <summary>
/// Implements the Revit add-in interface IExternalApplication
/// </summary>

public class MyApplication : IExternalApplication
{
    #region IExternalApplication Members

    /// <summary>
    /// Implements the OnStartup event
    /// </summary>
```

```
/// <param name="application"></param>
/// <returns></returns>

public Result OnStartup(UIControlledApplication application)
{
    // Lookup the desired command by name
    s_commandId = RevitCommandId.LookupCommandId(s_commandToDisable);

    // Confirm that the command can be overridden
    if (!s_commandId.CanHaveBinding)
    {
        ShowDialog("Error", "The target command " + s_commandToDisable +
                   " selected for disabling cannot be overridden");
        return ResultFailed;
    }

    // Create a binding to override the command.

    // Note that you could also implement .CanExecute to override the accessibility of the command.

    // Doing so would allow the command to be grayed out permanently or selectively, however,
    // no feedback would be available to the user about why the command is grayed out.

    try
    {
        AddInCommandBinding commandBinding = application.CreateAddInCommandBinding(s_commandId);
        commandBinding.Executed += DisableEvent;
    }
}
```

```
// Most likely, this is because someone else has bound this command already.

        catch (Exception)
        {

            ShowDialog("Error", "This add-in is unable to disable the target
command " + s_commandToDisable +
                       "; most likely another add-in has overridden this com
mand.");
        }

        return Result.Succeeded;
    }

/// <summary>
/// Implements the OnShutdown event
/// </summary>
/// <param name="application"></param>
/// <returns></returns>

public Result OnShutdown(UIControlledApplication application)
{
    // Remove the command binding on shutdown
    if (s_commandId.HasBinding)
        application.RemoveAddInCommandBinding(s_commandId);

    return Result.Succeeded;
}

#endregion
```

```
/// <summary>
/// A command execution method which disables any command it is applied to (with a user-visible message).
/// </summary>
/// <param name="sender">Event sender.</param>
/// <param name="args">Arguments.</param>
private void DisableEvent(object sender, ExecutedEventArgs args)
{
    ShowDialog("Disabled", "Use of this command has been disabled.");
}

/// <summary>
/// Show a task dialog with a message and title.
/// </summary>
/// <param name="title">The title.</param>
/// <param name="message">The message.</param>
private static void ShowDialog(string title, string message)
{
    // Show the user a message.
    TaskDialog td = new TaskDialog(title)
    {
        MainInstruction = message,
        TitleAutoPrefix = false
    };
    td.Show();
}
```

```

    /// <summary>
    /// The string name of the command to disable. To lookup a command id string, open a session of Revit,
    /// invoke the desired command, close Revit, then look to the journal from that session. The command
    /// id string will be toward the end of the journal, look for the "Jrn.Command" entry that was recorded
    /// when it was selected.
    /// </summary>
    static String s_commandToDisable = "ID_EDIT_DESIGNOPTIONS";

    /// <summary>
    /// The command id, stored statically to allow for removal of the command binding.
    /// </summary>
    static RevitCommandId s_commandId;

}

}

```

Posting a command

The method `UIApplication.PostCommand()` will post a command to the Revit message queue to be invoked when control returns from the current API application. Only certain commands can be posted this way. They include all of the commands in the `Autodesk.Revit.UI.PostableCommand` enumerated type as well as external commands created by any add-in.

Note: Even a postable command may not execute when using `PostCommand()`. One reason this may happen is if another command has already been posted. Only one command may be posted to Revit at a given time, so if a second command is posted, `PostCommand()` will throw an exception. Another reason a posted command may not execute is if the command to be executed is not accessible at the time. Whether it is accessible is determined only at the point where Revit returns from the API context, so a failure to execute for this reason will not be reported directly back to the application that posted the command. `UIApplication.CanPostCommand()` can be used to identify if the given command can be posted, meaning whether it is a member of `PostableCommand` or an external command. It does not identify if the command is currently accessible.

Both PostCommand() and CanPostCommand() require a RevitCommandId which can be obtained as described in the "Overriding a Revit command" section above.

5.4 Construction Modeling

The Revit API allows elements to be divided into sub-parts or collected into assemblies to support construction modeling workflows, much the same way as can be done with the Revit user interface. Both parts and assemblies can be independently scheduled, tagged, filtered, and exported. You can also divide a part into smaller parts. After creating an assembly type, you can place additional instances in the project and generate isolated assembly views.

The main classes related to Construction Modeling are:

- **AssemblyInstance** - This class combines multiple elements for tagging, filtering, scheduling and creating isolated assembly views.
- **AssemblyType** - Represents a type for construction assembly elements. Each new unique assembly created in the project automatically creates a corresponding AssemblyType. A new AssemblyInstance can be placed in the document from an existing AssemblyType.
- **PartUtils** - This utility class contains general part utility methods, including the ability to create parts, divide parts, and to get information about parts.
- **AssemblyViewUtils** - A utility class to create various types of assembly views.

5.4.1 Assemblies and Views

You can combine any number of model elements to create an assembly, which can then be edited, tagged, scheduled, and filtered.

Creating assemblies

The static Create() method of the AssemblyInstance class is used to create a new assembly instance in the project. The Create() method must be created inside a transaction and the transaction must be committed before performing any action on the newly created assembly instance. The assembly type is assigned after the transaction is complete. Each unique assembly has its own AssemblyType.

The following example creates a new assembly instance, changes the name of its AssemblyType and then creates some views for the assembly instance.

Code Region: Create Assembly and Views

```
AssemblyInstance CreateAssemblyAndViews(Autodesk.Revit.DB.Document doc, IColl  
ection<ElementId> elementIds)  
{
```

```
AssemblyInstance assemblyInstance = null;

using (Transaction transaction = new Transaction(doc))

{

    ElementId categoryId = doc.GetElement(elementIds.First()).Category.Id; // use category of one of the assembly elements

    if (AssemblyInstance.IsValidNamingCategory(doc, categoryId, elementIds))

    {

        transaction.Start("Create Assembly Instance");

        assemblyInstance = AssemblyInstance.Create(doc, elementIds, categoryId);

        transaction.Commit(); // commit the transaction that creates the assembly instance before modifying the instance's name


        if (transaction.GetStatus() == TransactionStatus.Committed)

        {

            transaction.Start("Set Assembly Name");

            assemblyInstance.AssemblyTypeName = "My Assembly Name";

            transaction.Commit();

        }

    }

    if (assemblyInstanceAllowsAssemblyViewCreation()) // create assembly views for this assembly instance

    {

        if (transaction.GetStatus() == TransactionStatus.Committed)

        {

            transaction.Start("View Creation");

            View3D view3d = AssemblyViewUtils.Create3DOrthographic(doc, assemblyInstance.Id);

        }

    }

}
```

```

        ViewSchedule partList = AssemblyViewUtils.CreatePartList
(doc, assemblyInstance.Id);

        transaction.Commit();

    }

}

}

return assemblyInstance;
}

```

Another way to create an AssemblyInstance is to use an existing AssemblyType. To create an AssemblyInstance using an AssemblyType, use the static method AssemblyInstance.PlaceInstance() and specify the ElementId of the AssemblyType to use and a location at which to place the assembly.

Assembly Views

Various assembly views can be created for an assembly instance using static methods of the AssemblyViewUtils class, including an orthographic 3D assembly view, a detail section assembly view, a material takeoff multicategory schedule assembly view, a part list multicategory schedule assembly view, a single-category schedule and a sheet assembly view. All of these except sheet views have overloaded creation methods which create the schedule or view from a template. In addition to a template Id, these overloads have a parameter to indicate if the template will be assigned or applied.

Note that assembly views must all be assigned to the same assembly instance of the assembly type. AssemblyInstance.AllowAssemblyViewCreation() returns true if that assembly instance can accept new assembly views (either because it already has views or because no assembly instance has views).

The following example creates a new single-category schedule for an assembly from a given template.

Code Region: Create assembly view from template

```

private ViewSchedule CreateScheduleForAssembly(Document doc, AssemblyInstance
assemblyInstance, ElementId viewTemplateId)

```

```
{  
    ViewSchedule schedule = null;  
  
    if (assemblyInstanceAllowsAssemblyViewCreation()) // create assembly views for this assembly instance  
    {  
        using (Transaction transaction = new Transaction(doc))  
        {  
            transactionStart("Create Schedule");  
  
            // use naming category for the schedule  
            if (ViewSchedule.IsValidCategoryForSchedule(assemblyInstanceNamingCategoryId))  
            {  
                schedule = AssemblyViewUtils.CreateSingleCategorySchedule(doc, assemblyInstanceId, assemblyInstanceNamingCategoryId, viewTemplateId, false);  
            }  
            transactionCommit();  
  
            if (schedule != null && transactionGetStatus() == TransactionStatus.Committed)  
            {  
                transactionStart("Edit Schedule");  
                scheduleName = "AssemblyViewSchedule";  
                transactionCommit();  
            }  
        }  
    }  
}
```

```
    return schedule;  
}
```

The document must be regenerated before using any of these newly created assembly views. You'll note in the example above that a transaction is committed after creating a new assembly view. The Commit() method automatically regenerates the document.

5.4.2 Parts

Parts can be generated from elements with layered structures, such as:

- Walls (excluding stacked walls and curtain walls)
- Floors (excluding shape-edited floors)
- Roofs (excluding those with ridge lines)
- Ceilings
- Structural slab foundations

In the Revit API, elements can be divided into parts using the PartUtils class. The static method PartUtils.CreateParts() is used to create parts from one or more elements. Note that this method instantiates a PartMaker and creates Parts that will be split in smaller parts if you chose to use DivideParts. The PartMaker uses its embedded rules to drive creation of the needed parts during regeneration.

The API also offers an interface to subdivide parts. PartUtils.DivideParts() accepts as input a collection of part ids, a collection of “intersecting element” ids (which can be layers and grids), and a collection of curves. The routine uses the intersecting elements and curves as boundaries from which to divide and generate new parts.

The GetAssociatedParts() method can be called to find some or all of the parts associated with an element, or use HasAssociatedParts() to determine if an element has parts.

You can delete parts through the API either by deleting the individual part elements, or by deleting the PartMaker associated to the parts (which will delete all parts generated by this PartMaker after the next regeneration).

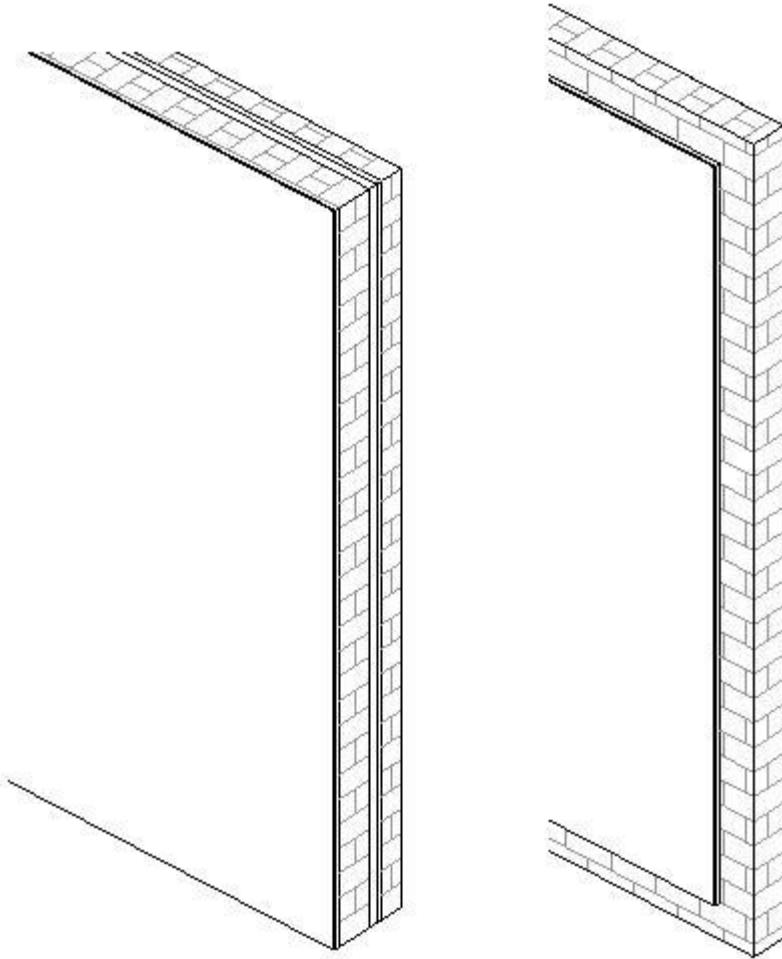
Parts can be manipulated in the Revit API much the same as they can in the Revit user interface. For example, the outer boundaries of parts may be offset with PartUtils.SetFaceOffset().

The following example offsets all the faces of a part that can be offset.

Code Region: Offset Faces of a Part

```
public void OffsetPartFaces(Part part)
{
    Document doc = part.Document;

    Autodesk.Revit.DB.GeometryElement geomElem = part.get_Geometry(new Options());
    foreach(GeometryObject geomObject in geomElem)
    {
        if (geomObject is Solid)
        {
            Solid solid = geomObject as Solid;
            FaceArray faceArray = solid.Faces;
            foreach (Face face in faceArray)
            {
                if (part.CanOffsetFace(face))
                {
                    part.SetFaceOffset(face, 1);
                }
            }
        }
    }
}
```



Before and After Offsetting faces of a selected Part

Returning the entities that create a divided Part

These methods identify and return the curves that were sketched to create the part division and, optionally, also outputs also the sketch plane for those curves.

- `PartUtils.GetSplittingCurves(Document, ElementId)`
- `PartUtils.GetSplittingCurves(Document, ElementId, out SketchPlane)`

`PartUtils.GetSplittingElements(Document, ElementId)` identifies and returns the elements (ReferencePlane, Level or Grid) that were used to create the division.

Parts and Direct Shapes

Parts can be created from `DirectShape` instances, either in the same host document or in a link. These methods indicate whether it is possible to create parts from an instance of one of those classes.

- `DirectShape.CanCreateParts()`
- `DirectShapeType.CanCreateParts()`

5.5 Context Menus

Context menus can be created from an add-in using the interface:

The constructors are:

- `Revit.UI.ContextMenu()` - Creates a new instance of Context Menu.
- `Revit.UI.CommandMenuItem(name , className, assemblyName)` - Creates a new instance of command menu item with name, external command class name and external application assembly name.
- `Revit.UI.SubMenuItem(name , ContextMenu)` - Creates a new instance of flyout menu with name and sub menu instance.
- `Revit.UI.Separator()` - Creates a new instance of separator menu item.

The methods are

- `ContextMenu.AddItem()` - Adds a specific type of MenuItem object to context menu.
- `CommandMenuItem.SetAvailabilityClassName()` - Sets the availabilityClassName of CommandMenuItem.
- `CommandMenuItem.SetToolTip()` - Sets the tooltip of a CommandMenuItem.
- `IContextMenuCreator.BuildContextMenu()` - Adds menu items to the passed in ContextMenu object.
- `UIControlledApplication.RegisterContextMenu()` - Registers a new context menu.

5.6 Dockable Dialog Panes

Since Revit 2013, applications have been able to use modeless dialogs by taking advantage of the [Idling Event](#) and the [External Events](#) class in the Revit API. Add-ins requiring modeless dialogs also have the option to use dockable modeless dialogs. Similar to standard modeless dialogs, dockable dialogs are registered Windows Presentation Foundation (WPF) dialog panes that participate in Revit's window docking system. A registered dockable pane can dock into the top, left, right, and bottom of the main Revit window, as well as be added as a tab to an existing system pane, such as the project browser. Additionally, dockable panes can float, behaving much like a standard modeless dialog.

IDockablePaneProvider

Registering a dockable pane requires an instance of the `IDockablePaneProvider` interface. The `SetupDockablePane()` method of this interface is called during initialization of the Revit user interface to gather information about add-in dockable pane windows. `SetupDockablePane()` has one parameter of type `DockablePaneProviderData`, which is a container for information about the new dockable pane.

Implementations of the `IDockablePaneProvider` interface should set the `FrameworkElement` and `InitialState` properties of `DockablePaneProviderData`. The `FrameworkElement` property is the Windows Presentation Framework object containing the pane's user interface.

Note: It is recommended that the dockable dialog in the add-in be the class that implements `IDockablePaneProvider` and that it be subclassed from `System.Windows.Controls.Page`.

The `InitialState` property is the initial position and settings of the docking pane, indicated by the `DockablePaneState` class. The pane's `DockPosition` can be `Top`, `Bottom`, `Left`, `Right`, `Floating` or `Tabbed`. If the position is `Tabbed`, the `DockablePaneState.TabBehind` property can be used to specify which pane the new pane will appear behind. If the position is `Floating`, the `DockablePaneState.FloatingRectangle` property contains the rectangle that determines the size and position of the pane.

DockablePane

To access a dockable pane during runtime, it needs to be registered by calling the `UIApplication.RegisterDockablePane()` method. This method requires a unique identifier for the new pane (`DockablePanelId`), a string specifying the caption for the pane, and an implementation of the `IDockablePaneProvider` interface.

Dockable panes can be accessed by calling `UIApplication.GetDockablePane()` and passing in the unique `DockablePanelId`. This method returns a `DockablePane`. `DockablePane.Show()` will display the pane in the Revit user interface at its last docked location, if not currently visible. `DockablePane.Hide()` will hide a visible dockable pane. However, it has no effect on built-in Revit dockable panes.

5.7 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.

5.7.1 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 unless the Updater is flagged as optional. By default, updaters are non-optional and optional updaters should be used only when necessary.

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

```
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.GetElement(addedElemId) as Wall;
        if (wall != null)
        {
            wall
```

```
        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";
}
```

{}

5.7.2 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.

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.

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)
- AreaReinforcement.Create()
- PathReinforcement.Create()

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.

Calls to the `UpdaterRegistry` class, such as `RegistryUpdater()` or `AddTrigger()`, from within the `Execute()` method of an updater are also forbidden. Calling any of the `UpdaterRegistry` methods will throw an exception. The one exception to this rule is the `UpdaterRegistry.UnregisterUpdater()` method, which may be called during execution of an updater as long as the updater to be unregistered is not the one currently being executed.

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.NewFamilyInstances2()`
- `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.

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.

5.7.3 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 and can be registered as application-wide or document-specific, meaning they will only be triggered by changes made to the specified document. In order to use the UpdaterRegistry functionality, the Revit add-in must be registered in a manifest file 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.

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.

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.

5.7.4 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 3_{rd} party updater. Such messages will use the value of the GetUpdaterName() method to refer to the updater.

Updater not installed

If a document is modified by a non-optional updater and later loaded when that updater is not installed, a task dialog similar to the following is displayed:

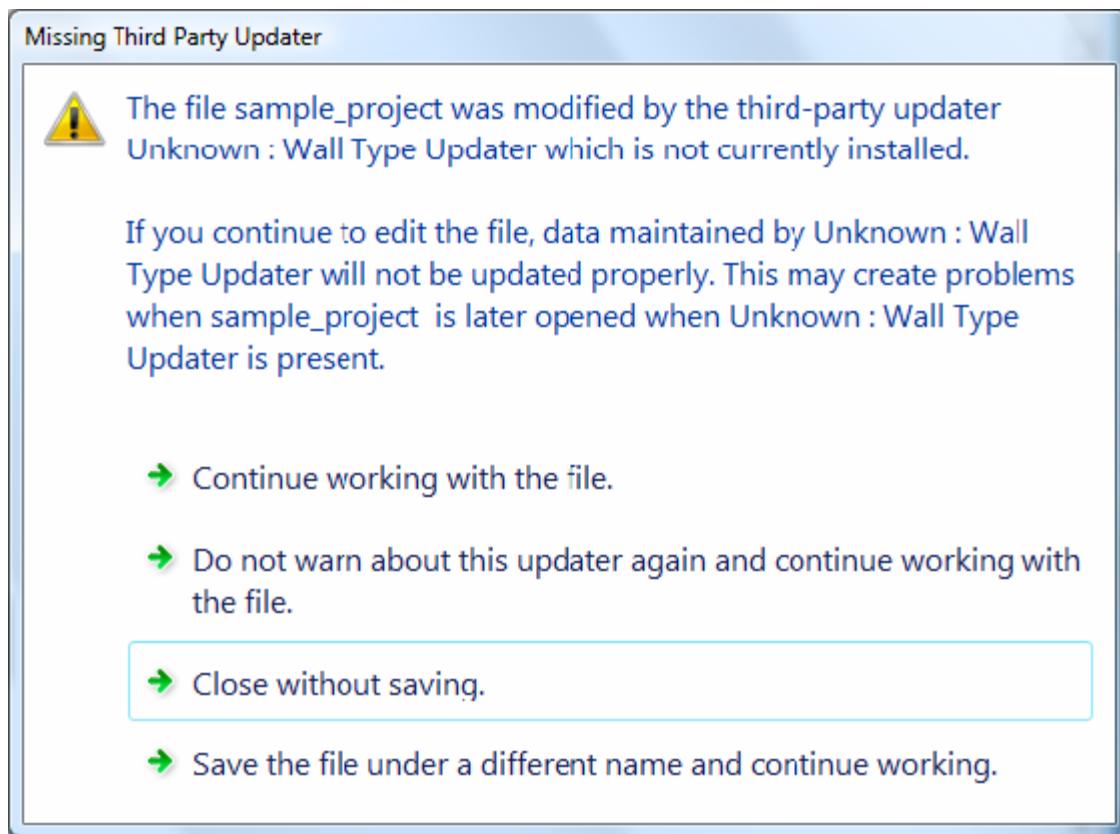


Figure 135: Missing Third Party Updater Warning

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:

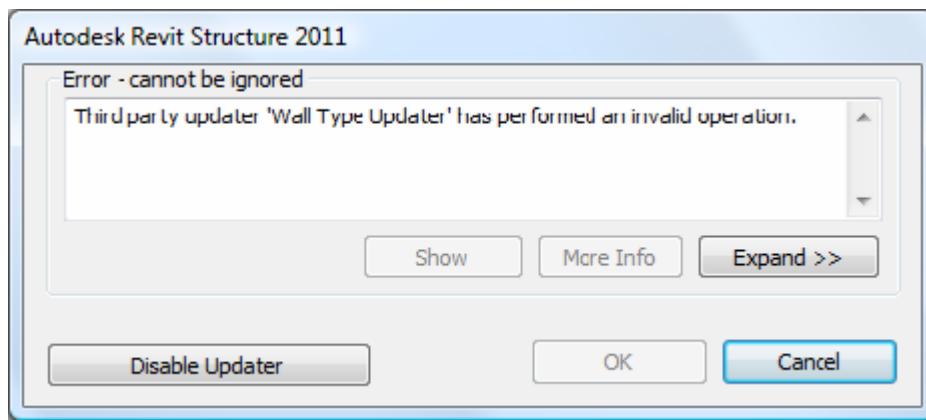


Figure 136: Updater performed an invalid operation

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.

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.

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.

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.

5.8 Storing Data in the Revit model

Use shared parameters or extensible storage to store data in the Revit model.

The Revit API provides two methods for storing data in the Revit model. The first is using shared parameters. The Revit API gives programmatic access to the same shared parameters feature that is available through the Revit UI. Shared parameters, if defined as visible, will be viewable to the user in an element's property window. Shared parameters can be assigned to many, but not all, categories of elements. For more information, see [Shared Parameters](#).

The other option is extensible storage, which allows you to create custom data structures and then assign instances of that data to elements in the model. This data is never visible to the user in the Revit UI, but may be accessible to other third party applications via the Revit API depending on the read/write access assigned to the schema when it is defined. Unlike Shared Parameters, extensible storage is not limited to certain categories of elements. Extensible storage data can be assigned to any object that derives from the base class Element in the Revit model.

5.8.1 Extensible Storage

Create your own class-like Schema data structures and attach instances of them to any Element in a Revit model.

Schema-based data is saved with the Revit model and allows for higher-level, metadata-enhanced, object-oriented data structures. Schema data can be configured to be readable and/or writable to all users, just a specific application vendor, or just a specific application from a vendor.

The following steps are necessary to store data with Elements in Revit:

1. Create and name a new schema
2. Set the read/write access for the schema
3. Define one or more fields of data for the schema
4. Create an entity based on the schema
5. Assign values to the fields for the entity
6. Associate the entity with a Revit element

Schemas and SchemaBuilder

The first step to creating extensible storage is to define the schema. A schema is similar to a class in an object-oriented programming language. Use the SchemaBuilder class constructor to create a new schema. SchemaBuilder is a helper class used to create schemas. Once a schema is finalized using SchemaBuilder, the Schema class is used to access properties of the schema. At that stage, the schema is no longer editable.

Although the SchemaBuilder constructor takes a GUID which is used to identify the schema, a schema name is also required. After creating the schema, call SchemaBuilder.SetSchemaName() to assign a user-friendly identifier for the schema. The schema name is useful to identify a schema in an error message.

The read and write access levels of entities associated with the schema can be set independently. The options are Public, Vendor, or Application. If either the read or write access level is set to Vendor, the VendorId of the third-party vendor that may access entities of the schema must be specified. If either access level is set to Application, the GUID of the application or add-in that may access entities of the schema must be supplied.

Note: Schemas are stored with the document and any Revit API add-in may read the available schemas in the document, as well as some data of the schema. However, access to the fields of a schema is restricted based on the read access defined in the schema and the actual data in the entities stored with specific elements is restricted based on the read and write access levels set in the schema when it is defined.

Fields and FieldBuilder

Once the schema has been created, fields may be defined. A field is similar to a property of a class. It contains a name, documentation, value type and unit type. Fields can be a simple type, an array, or a map. The following simple data types are allowed:

Type	Default Value
int	0
short	0
byte	0

double	0.0
float	0.0
bool	false
string	Empty string ("")
GUID	Guid.Empty {00000000-0000-0000-0000-000000000000}
ElementId	ElementId.InvalidElementId
Autodesk.Revit.DB.XYZ	(0.0,0.0,0.0)
Autodesk.Revit.DB.UV	(0.0,0.0)

Additionally, a field may be of type Autodesk.Revit.DB.ExtensibleStorage.Entity. In other words, an instance of another Schema, also known as a SubSchema or SubEntity. The default value for a field of this type is Entity with null schema, and guid of Guid.Empty.

When using string fields, note that Revit has a 16mb limit on string objects.

A simple field can be created using the SchemaBuilder.AddSimpleField() method to specify a name and type for the field. AddSimpleField() returns a FieldBuilder, which is a helper class for defining Fields. If the type of the field was specified as Entity, use FieldBuilder.SetSubSchemaGUID() to specify the GUID of the schema of the Entities that are to be stored in this field.

Use the SchemaBuilder.AddArrayField() method to create a field containing an array of values in the Schema, with a given name and type of contained values. Array fields can have all the same types as simple fields.

Use the SchemaBuilder.AddMapField() method to create a field containing an ordered key-value map in the Schema, with given name, type of key and type of contained values. Supported types for values are the same as for simple fields. Supported types for keys are limited to int, short, byte, string, bool, ElementId and GUID.

Once the schema is finalized using SchemaBuilder, fields can no longer be edited using FieldBuilder. At that stage, the Schema class provides methods to get a Field by name, or a list of all Fields defined in the Schema.

Entities

After all fields have been defined for the schema, SchemaBuilder.Finish() will return the finished Schema. A new Entity can be created using that schema. For each Field in the Schema, the

value can be stored using Entity.Set(), which takes a Field and a value (whose type is dependent on the field type). Once all applicable fields have been set for the entity, it can be assigned to an element using the Element.SetEntity() method.

To retrieve the data later, call Element.GetEntity() passing in the corresponding Schema. If no entity based on that schema was saved with the Element, an invalid Entity will be returned. To check that a valid Entity was returned, call the Entity.IsValid() method. Field values from the entity can be obtained using the Entity.Get() method.

To remove an extensible storage entity from an Element, call Element.DeleteEntity() passing in the Schema that was used to create it.

To determine Entities stored with an element, use the Element.GetEntitySchemaGuids() method, which returns the Schema guids of any Entities for the Element. The Schema guids can be used with the static method Schema.Lookup() to retrieve the corresponding Schemas.

The following is an example of defining an extensible storage Schema, creating an Entity, setting its values, assigning it to an Element, and retrieving the data.

Code Region 22-9: Extensible Storage

```
// Create a data structure, attach it to a wall, populate it with data, and r
etrieve the data back from the wall

void StoreDataInWall(Wall wall, XYZ dataToStore)

{
    Transaction createSchemaAndStoreData = new Transaction(wall.Document,
    "tCreateAndStore");

    createSchemaAndStoreData.Start();

    SchemaBuilder schemaBuilder =
        new SchemaBuilder(new Guid("720080CB-DA99-40DC-9415-E53F280AA
1F0"));

    schemaBuilder.SetReadAccessLevel(AccessLevel.Public); // allow anyone
    to read the object

    schemaBuilder.SetWriteAccessLevel(AccessLevel.Vendor); // restrict wr
iting to this vendor only

    schemaBuilder.SetVendorId("ADSK"); // required because of restricted
    write-access

    schemaBuilder.SetSchemaName("WireSpliceLocation");
```

```
// create a field to store an XYZ

FieldBuilder fieldBuilder =
    schemaBuilder.AddSimpleField("WireSpliceLocation", typeof(XYZ));
fieldBuilder.SetSpec(SpecTypeId.Length);
fieldBuilder.SetDocumentation("A stored location value representing a
wiring splice in a wall.");

Schema schema = schemaBuilder.Finish(); // register the Schema object

Entity entity = new Entity(schema); // create an entity (object) for
this schema (class)

// get the field from the schema

Field fieldSpliceLocation = schema.GetField("WireSpliceLocation");

// set the value for this entity

entity.Set<XYZ>(fieldSpliceLocation, dataToStore, UnitTypeId.Meters);
wall.SetEntity(entity); // store the entity in the element

// get the data back from the wall

Entity retrievedEntity = wall.GetEntity(schema);

XYZ retrievedData =
    retrievedEntity.Get<XYZ>(schema.GetField("WireSpliceLocation"),
    UnitTypeId.Meters);

createSchemaAndStoreData.Commit();
}
```

Advantages

Self Documenting and Self-Defining

Creating a schema by adding fields, units, sub-entities, and description strings is not only a means for storing data. It is also implicit documentation for other users and a way for others to create entities of the same schema later with an easy adoption path.

Takes Advantage of Locality

Because schema entities are stored on a per-element basis, there is no need to necessarily read all extensible storage data in a document (e.g. all data from all beam family instances) when an application might only need data for the currently selected beam. This allows the potential for more specifically targeted data access code and better data access performance overall.

5.9 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.

5.9.1 Database Events

The following table lists database events, their type and whether they are available at the application and/or document level:

Table 53: DB Event Types

Event	Type	Application	Document
<u>DocumentChanged event</u>	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	
DocumentSynchronizedWithCentral	post	X	
FailuresProcessing	single	X	
FileExporting	pre	X	
FileExported	post	X	

FileImporting	pre	X	
FileImported	post	X	
ProgressChanged	single	X	
ViewPrinting	pre	X	X
ViewPrinted	post	X	X

- 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
- ProgressChanged - notification when an operation in Revit has progress bar data

- 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

5.9.1.1 *DocumentChanged event*

The DocumentChanged event is triggered when the Revit document has changed. This event is raised whenever a Revit transaction is either committed, undone or redone. This is a read-only event, designed to allow external data to be kept in sync with the state of the Revit database. To update the Revit database in response to changes in elements, use the IUpdater framework.

The DocumentChangedEventArgs class is used by the DocumentChanged event. This class has several methods to get the element IDs of any newly added elements (GetAddElementIDs()), deleted elements (GetDeletedElementIDs()) or elements that have been modified (GetModifiedElementIDs()). The GetAddElementIDs() and GetModifiedElementIDs() methods have overloads that take an ElementFilter, which makes it easy to detect only changes of interest.

5.9.2 User Interface Events

The following table lists user interface events, their type and whether they are available at the application and/or document level:

Table 54: UI Event Types

Event	Type	UI Application	Controlled Application	UIDocument
ApplicationClosing	pre	X		
ApplicationInitialized	single		X	
DialogBoxShowing	single	X		
DisplayingOptionsDialog	single	X		
Idling	single	X		
ViewActivating	pre	X		
ViewActivated	post	X		

- ApplicationClosing - notification when the Revit application is about to be closed
- ApplicationInitialized - notification after the Revit application has been initialized, after all external applications have been started and the application is ready to work with documents

- DialogBoxShowing - notification when Revit is showing a dialog or message box
- DisplayingOptionsDialog - notification when Revit options dialog is displaying
- 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

5.9.3 Registering Events

Where and how to register 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 DocumentSavingEventArgs 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. Also note that registering to and unregistering from events must happen while executing on the main thread. An exception will be thrown if an external application attempts to register to (or unregister from) events from outside of a valid API context.

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 ControlledApplication.DocumentOpened

```
public class Application_DocumentOpened : IExternalApplication
{
    /// <ExampleMethod>
    /// <summary>
    /// Implement this method to subscribe to event.
    /// </summary>
    public Result OnStartup(UIControlledApplication application)
    {
```

```
try

{
    // Register event.

    application.ControlledApplication.DocumentOpened += new EventHandler
ler

        <Autodesk.Revit.DB.Events.DocumentOpenedEventArgs>(applicatio
n_DocumentOpened);

}

catch (Exception)
{
    return Result.Failed;
}

return Result.Succeeded;
}

public Result OnShutdown(UIControlledApplication application)
{
    // remove the event.

    application.ControlledApplication.DocumentOpened -= application_Docum
entOpened;

    return Result.Succeeded;
}

public void application_DocumentOpened(object sender, DocumentOpenedEvent
Args args)
{
    // get document from event args.
```

```
Document doc = args.Document;

// Following code snippet demonstrates support of DocumentOpened even-
t to modify the model.

// Because DocumentOpened supports model changes, it allows user to u-
pdate document data.

// Here, this sample assigns a specified value to ProjectInformation.
Address property.

// User can change other properties of document or create(delete) som-
ething as he likes.

// Please note that ProjectInformation property is empty for family d-
ocument.

// So please don't run this sample on family document.

using (Transaction transaction = new Transaction(doc, "Edit Address
"))

{
    if (transaction.Start() == TransactionStatus.Started)

    {
        doc.ProjectInformation.Address =
            "United States - Massachusetts - Waltham - 1560 Trapelo R-
oad";
        transaction.Commit();
    }
}
}
```

5.9.4 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: 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, DocumentSavingEventArgs 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();
            TaskDialog.Show("Error", "Status project parameter is not set. Save is aborted.");
        }
    }
}
```

{}

Note: Although most event arguments have the Cancel and Cancellable properties, the DocumentChanged and FailuresProcessing events have corresponding Cancel() and IsCancellable() methods.

5.10 Export

The Revit API allows for a Revit document, or a portion thereof, to be exported to various formats for use with other software. The Document class has an overloaded Export() method that will initiate an export of a document using the built-in exporter in Revit (when available). For more advanced needs, some types of exports can be customized with a Revit add-in, such as export to IFC and export to Navisworks. (Note, Navisworks export is only available as an add-in exporter).

The Document.Export() method overloads are outlined in the table below.

Table: Document.Export() Methods

Format	Export() parameters	Comments
gbXML	String, String, GBXMLExportOptions	Exports a gbXML file from a mass model document
gbXML	String, String, GBXMLExportOptions	Exports the document in Green-Building XML format. If EnergyDataSettings is set to use conceptual models, this function cannot be used: instead use the method above.
IFC	String, String, IFCExportOptions	Exports the document to the Industry Standard Classes (IFC) format.
NWC	String, String, NavisworksExportOptions	Exports a Revit project to the Navisworks .nwc format. Note that in order to use this function, you must have a compatible Navisworks exporter add-in registered with your session of Revit.
DWF	String, String, ViewSet, DWFExportOptions	Exports the current view or a selection of views in DWF format.
DWFX	String, String, ViewSet, DWFXExportOptions	Exports the current view or a selection of views in DWFX format.

FBX	<code>String, String, ViewSet, FBXExportOptions</code>	Exports the document in 3D-Studio Max (FBX) format.
DGN	<code>String, String, ICollection(ElementId), DGNElexportOptions</code>	Exports a selection of views in DGN format.
DWG	<code>String, String, ICollection(ElementId), DWGExportOptions</code>	Exports a selection of views in DWG format.
DXF	<code>String, String, ICollection(ElementId), DXFExportOptions</code>	Exports a selection of views in DXF format.
SAT	<code>String, String, ICollection(ElementId), SATExportOptions</code>	Exports the current view or a selection of views in SAT format.
PDF	<code>String, IList(ElementId), PDFExportOptions</code>	Exports a selection of views in PDF format.

Exporting to gbXML

There are two methods for exporting to the Green Building XML format. The one whose last parameter is `GBXMLExportOptions` is only available for projects containing one or more instances of Conceptual Mass families. The `GBXMLExportOptions` object to pass into this method can be constructed with just the ids of the mass zones to analyze in the exported gbXML, or with the mass zone ids and the ids of the masses to use as shading surfaces in the exported gbXML. When using masses, they must not have mass floors or mass zones so as not to end up with duplicate surface information in the gbXML output.

The `GBXMLExportOptions` object used for the other gbXML export option has no settings to specify. It uses all default settings. Note that this method does not generate the energy model. The main energy model must already be stored in the document before this export is invoked.

Exporting to IFC

Calling `Document.Export()` using the IFC option will either use the default Revit IFC export implementation or a custom [IFC Export](#), if one has been registered with the current session of Revit. In either case, the `IFCExportOptions` class is used to set export options such as whether to export IFC standard quantities currently supported by Revit or to allow division of multi-level walls and columns by levels.

Exporting to Navisworks

The Export method for Navisworks requires a compatible Navisworks exporter add-in registered with the current Revit session. If there is no compatible exporter registered, the method will throw an exception. Use the `OptionalFunctionalityUtils.IsNavisworksExporterAvailable()` method to determine if a Navisworks exporter is registered.

The `NavisworksExportOptions` object can be used to set numerous export settings for exporting to Navisworks, such as whether to divide the file into levels and whether or not to export room geometry. Additionally, the `NavisworksExportOptions.ExportScope` property specifies the export scope. The default is `Model`. Other options include `View` and `SelectedElements`. When set to `View`, the `NavisworksExportOptions.ViewId` property should be set accordingly. This property is only used when the export scope is set to `View`. When set to `SelectedElements`, the `NavisworksExportOptions.SetSelectedElementIds()` method should be called with the ids of the elements to be exported.

Exporting to DWF and DWFX

Both DWF and DWFX files can be exported using the corresponding `Document.Export()` overloads. Both methods have a `ViewSet` parameter that represents the views to be exported. All the views in the `ViewSet` must be printable in order for the export to succeed. This can be checked using the `View.CanBePrinted` property of each view. The last parameter is either `DWFExportOptions` or `DWFXExportOptions`. `DWFXExportOptions` is derived from `DWFExportOptions` and has all the same export settings. Options include whether or not to export the crop box, the image quality whether or not to export textures to 3D DWF files, and the paper format.

Code Region: Export DWF

```
public bool ExportViewToDWF(Document document, View view, string pathname)
{
    DWFExportOptions dwfOptions = new DWFExportOptions();

    // export with crop box and area and room geometry
    dwfOptions.CropBoxVisible = true;
    dwfOptions.ExportingAreas = true;
    dwfOptions.ExportTexture = false;

    ViewSet views = new ViewSet();
    views.Insert(view);

    return (document.Export(Path.GetDirectoryName(pathname),
        Path.GetFileNameWithoutExtension(pathname), views, dwfOptions));
}
```

{}

Exporting to 3D-Studio Max

FBX Export requires the presence of certain modules that are optional and may not be part of the installed Revit, so the `OptionalFunctionalityUtils . IsFBXExportAvailable()` method reports whether the FBX Export functionality is available. The `Export()` method for exporting to 3D-Studio Max has a `ViewSet` parameter representing the set of views to export. Only 3D views are allowed. The `FBXExportOptions` parameter can be used to specify whether to export without boundary edges, whether to use levels of detail, and whether the export process should stop when a view fails to export.

Exporting to CAD Formats

Exporting to DGN , DWG and DXF format files have similar export methods and options.

DGN , DWG and DXF Export all require the presence of certain modules that are optional and may not be part of the installed version of Revit, so the `OptionalFunctionalityUtils` class has corresponding methods to report whether each of these types of export functionality are available.

The `Export()` methods for exporting to DGN, DWG and DXF formats all have a parameter representing the views to be exported (as an `ICollection<ElementId>` of the `ElementIds` of the views). At least one valid view must be present and it must be printable for the export to succeed. This can be checked using the `View.CanBePrinted` property of each view.

The export options for each of these formats derives from `BaseExportOptions`, so there are many export settings in common, such as the color mode or whether or not to hide the scope box. `BaseExportOptions` also has a static method called `GetPredefinedSetupNames()` which will return any predefined setups for a document. The name of a predefined setup can then be passed into the static method `GetPredefinedOptions()` available from the corresponding options class: `DWGExportOptions`, `DGNEExportOptions` or `DXFExportOptions`. The export options for DWG and DXF files have even more options in common as they share the base class `ACADEExportOptions`.

The following example exports the active view (if it can be printed) to a DGN file using the first predefined setup name and the predefined options, without making any changes.

Code Region: Export DGN

```
public bool ExportDGN(Document document, View view)
{
    bool exported = false;
```

```

// Get predefined setups and export the first one

IList<string> setupNames = BaseExportOptions.GetPredefinedSetupNames(document);

if (setupNames.Count > 0)

{

    // Get predefined options for first predefined setup

    DGNExportOptions dgnOptions = DGNExportOptions.GetPredefinedOptions(document, setupNames[0]);


    // export the active view if it is printable

    if (document.ActiveView.CanBePrinted == true)

    {

        ICollection<ElementId> views = new List<ElementId>();

        views.Add(view.Id);

        exported = document.Export(Path.GetDirectoryName(document.PathName),

            Path.GetFileNameWithoutExtension(document.PathName), views, dgnOptions);

    }

}

return exported;
}

```

For these file types it is possible to specify or modify various mapping settings, such as layer mapping or text font mapping by creating the corresponding table and passing it into the appropriate method of the `BaseExportOptions` class. For layer mapping, you may alternatively pass in a string value to `BaseExportOptions.LayerMapping` of a predefined layer mapping style or the filename of a layer mapping file. For more information on creating or modifying export tables, see the [Export Tables](#) topic.

Exporting to SAT

The Export method for SAT has a parameter representing the views to be exported (as an ICollection of the ElementIds of the views). At least one valid view must be present and it must be printable for the export to succeed. This can be checked using the View.CanBePrinted property of each view. The SATExportOptions object has no settings to specify. It uses all default settings.

Exporting to Civil Engineering Design Applications

The last Export() method exports a 3D view of the document in the format of Civil Engineering design applications. One parameter of the method is a Viewplan that specifies the gross area plan. All the areas on the view plan will be exported and it must be 'Gross Building' area plan. To check whether its area scheme is Gross Building, use the AreaScheme.GrossBuildingArea property. The BuildingSiteExportOptions object allows custom values for settings such as gross area or total occupancy.

Exporting to PDF

Code Region: Print all sheets to PDF

```
private void PDFExport(Document doc)
{
    PDFExportOptions opt = new PDFExportOptions();
    opt.Combine = true;
    opt.FileName = "My House";
    opt.PaperFormat = ExportPaperFormat.ARCH_E;
    doc.Export(
        Environment.GetFolderPath(Environment.SpecialFolder.Desktop),
        new FilteredElementCollector(doc)
            .OfClass(typeof(ViewSheet))
            .ToElementIds().ToList(),
        opt);
}
```

5.10.1 Export Tables

The classes listed in the table below expose read and write access to the tables used for mapping on export to various formats such as DWG and DGN. Each class contains (key, info) pairs of mapping data.

Class	Description
ExportLayerTable	Represents a table supporting a mapping of various layer properties (Category, name, color name) to layer name in the target export format.
ExportLinetypeTable	Represents a table supporting a mapping of linetype names in the target export format.
ExportPatternTable	Represents a table supporting a mapping of FillPattern names and ids to FillPattern names in the target export format.
ExportFontTable	Represents a table supporting a mapping of font names in the target export format.
ExportLineweightTable	Represents a table supporting a mapping of linewidth names in the target export format.

Most of these tables are available through the BaseExportOptions class (the base class for options for DGN, DXF and DWG formats). The ExportLineweightTable is available from the DGNExportOptions class. Each table has a corresponding Get and Set method. To modify the mappings, get the corresponding table, make changes and then set it back to the options class.

The following example modifies the ExportLayerTable prior to exporting a DWG file.

Code Region: Export DWG with modified ExportLayerTable

```
public bool ExportDWGModifyLayerTable(Document document, View view)
{
    bool exported = false;

    IList<string> setupNames = BaseExportOptions.GetPredefinedSetupNames(document);

    if (setupNames.Count > 0)
    {
        // Get the export options for the first predefined setup
        DWGExportOptions dwgOptions = DWGExportOptions.GetPredefinedOptions(document, setupNames[0]);
    }
}
```

```
// Get the export layer table
ExportLayerTable layerTable = dwgOptions.GetExportLayerTable();

// Find the first mapping for the Ceilings category
string category = "Ceilings";

ExportLayerKey targetKey = layerTable.GetKeys().First<ExportLayerKey>
(layerKey => layerKey.CategoryName == category);

ExportLayerInfo targetInfo = layerTable[targetKey];

// change the color name and cut color number for this mapping
targetInfo.ColorName = "31";
targetInfo.CutColorNumber = 31;

// Set the change back to the map
layerTable[targetKey] = targetInfo;

// Set the modified table back to the options
dwgOptions.SetExportLayerTable(layerTable);

ICollection<ElementId> views = new List<ElementId>();
views.Add(view.Id);

exported = document.Export(Path.GetDirectoryName(document.PathName),
Path.GetFileNameWithoutExtension(document.PathName), views, dwgOptions);
}

}
```

```
    return exported;  
}
```

5.10.2 IFC Export

Custom IFC Export

The Revit API allows custom applications to override the default implementation for the IFC export process.

When an **IExporterIFC** object is registered with Revit, it will be used both when the user invokes an export to IFC from the UI as well as from the API method `Document.Export(String, String, IFCExportOptions)`. In both cases, if no custom IFC exporter is registered, the default Revit implementation for IFC export is used.

When invoking an IFC export from the API, `IFCExportOptions` can be used to set the same export options as are available to the user from the Export IFC dialog box.

The functions:

- `IFCImportOptions.GetExtraOptions()`
- `IFCImportOptions.SetExtraOptions()`

allow for passing in arbitrary options for custom IFC importers. Users can pass in a string to string map specifying extra data they wish to pass for IFC import.

IExporterIFC

The interface **IExporterIFC** has only one method to implement, `ExportIFC()`. This method is invoked by Revit to perform an export to IFC. An **ExporterIFC** object is passed to this method as one of its parameters. **ExporterIFC** is the main class provided by Revit to allow implementation of an IFC export. It contains information on the options selected by the user for the export operation, as well as members used to access specific types of data needed to implement the export properly.

The Autodesk.Revit.DB.IFC namespace contains numerous IFC related API classes that can be utilized by a custom implementation of the IFC export process. For a complete sample of a custom IFC export application, see the Open Source example at <http://sourceforge.net/projects/ifcexporter/>.

5.10.3 Custom export

Use a custom export process to export views from a Revit document.

The Revit API provides a set of classes that make it possible to export views via a custom export context. These classes provide access to the rendering output pipeline through which Revit sends the graphical representation of a model to an output device. In the case of a custom export, the "device" is represented by a context object that could be any kind of a device. A file would be the most common case.

An implementation of a custom exporter provides a context and invokes rendering of a model, upon which Revit starts processing the model and sends graphic data out via methods of the context. The data describes the model exactly as it would have appeared in Revit when the model is rendered. The data includes all geometry and material properties.

CustomExporter class

The CustomExporter class allows exporting views via a custom export context. The Export() method of this class triggers the standard rendering process in Revit, but instead of displaying the result on screen or printer, the output is channeled through the given custom context that handles processing the geometric as well as non-geometric information.

CustomExporter support for 2D views

CustomExporter can export 2D plan, section and elevation views.

The method `CustomExporter.Export(IList<ElementId>)` can accept either 3D or 2D views, with the limitation that views in the collection must be either all 3D or all 2D.

For both Export() calls, the exporter context must correspond to the views' type; use IModelExportContext or IPhotoRenderContext for 3D views and IExportContext2D for 2D views.

Several properties for the CustomExporter exist to support 2D objects:

- `CustomExporter.Export2DGeometricObjectsIncludingPatternLines` - Indicates whether pattern lines of geometric objects should be exported in a 2D context. Defaults to false.
- `CustomExporter.Export2DIncludingAnnotationObjects` - Indicates whether annotation objects should be exported in a 2D context. Defaults to false.
- `CustomExporter.Export2DForceDisplayStyle` - Forces a display style for the export. If the style is `DisplayStyle.Undefined`, then export uses `DisplayStyle.Wireframe` for wireframe views and `DisplayStyle.HLR` for other views. Defaults to `DisplayStyle.Undefined`.

Use the interface `IExportContext2D` for exporting 2D views. It has the following methods in addition to the method inherited from `IExportContext`:

- `IExportContext2D.OnElementBegin2D()`
- `IExportContext2D.OnElementEnd2D()`
- `IExportContext2D.OnFaceEdge2D()`
- `IExportContext2D.OnFaceSilhouette2D()`

To access data for various 2D exported objects, use the classes:

- `ElementNode`

- FaceEdgeNode
- FaceSilhouetteNode

Some notes on 2D export in `DisplayStyle.Wireframe`:

1. Geometric object methods (`OnCurve`, `OnFaceEdge2D`, `OnFaceSilhouette2D`) are called regardless of the object being eventually output, i.e. even if it is occluded by another element.

And in `DisplayStyle.HLR`:

1. Tessellated geometry methods (`OnLineSegment` and `OnPolylineSegments`) are called regardless of the return value of the respective geometric object methods (`OnCurve`, `OnFaceEdge2D`, `OnFaceSilhouette2D`).
2. None of these methods are called between the respective pairs of calls `OnInstanceBegin/OnInstanceEnd` or `OnLinkBegin/OnLinkEnd`. They are called between `OnElementBegin2D/OnElementEnd2D` and `OnViewBegin/OnViewEnd`.

For an example of the use of the API for custom export of 2D views, see the SDK sample `CustomExporter/Custom2DExporter`.

CustomExporter events

Subscribe to these events to be notified when Revit is just about to export, or has just exported, one or more views of the document via an export context by `CustomExporter`:

- `Autodesk.Revit.ApplicationServices.Application.ViewsExportingByContext`
- `Autodesk.Revit.ApplicationServices.Application.ViewsExportedByContext`

`Autodesk.Revit.DB.Events.ViewsExportingByContextEventArgs` provides information when Revit is just about to export one or more views of the document via an export context by `CustomExporter`. It has the method `ViewsExportingByContextEventArgs.GetViewIds()` to get the ids of views about to be exported by `CustomExporter`.

`Autodesk.Revit.DB.Events.ViewsExportedByContextEventArgs` provides information when Revit has just exported one or more views of the document via an export context by `CustomExporter`. It has the method `ViewsExportedByContextEventArgs.GetViewIds()` - Gets the ids of views that have been exported by `CustomExporter`.

IExportContext

An instance of the `IExportContext` class is passed in as a parameter of the `CustomExporter` constructor. The methods of this interface are then called as the entities of the model are exported.

Although it is possible to create classes derived from the `IExportContext` class, it is preferred to use one of its derived classes: `IPhotoRenderContext` or `IModelExportContext`. When using an `IPhotoRenderContext` to perform a custom export, Revit will traverse the model and output the model's geometry as if processing the `Render` command invoked via the UI. Only elements that

have actual geometry and are suitable to appear in a rendered view will be processed and output.

The `IModelExportContext` should be used for processing elements in the view in the same manner that Revit processes them in 3D views. This context supports additional elements including model curves and text as shown in the 3D views.

RenderNode classes

`RenderNode` is the base class for all output nodes in a model-exporting process. A node can be either geometric (such as an element or light) or non-geometric (such as material). Some types of nodes are container nodes, which include other render nodes.

CameraInfo

The `CameraInfo` class describes information about projection mapping of a 3D view to a rendered image. An instance of this class can be obtained via a property of `ViewNode`. If it is null, an orthographic view should be assumed.

5.11 External Events

The Revit API provides an External Events framework to accommodate the use of modeless dialogs. It is tailored for asynchronous processing and operates similarly to the `Idling` event with default frequency.

To implement a modeless dialog using the External Events framework, follow these steps:

1. Implement an external event handler by deriving from the `IExternalEventHandler` interface
2. Create an `ExternalEvent` using the static `ExternalEvent.Create()` method
3. When an event occurs in the modeless dialog where a Revit action needs to be taken, call `ExternalEvent.Raise()`
4. Revit will call the implementation of the `IExternalEventHandler.Execute()` method when there is an available Idling time cycle.

IExternalEventHandler

This is the interface to be implemented for an external event. An instance of a class implementing this interface is registered with Revit, and every time the corresponding external event is raised, the `Execute` method of this interface is invoked.

The `IExternalEventHandler` has only two methods to implement, the `Execute()` method and `GetName()` which should return the name of the event. Below is a basic implementation which will display a `TaskDialog` when the event is raised.

Code Region: Implementing `IExternalEventHandler`

```
public class ExternalEventExample : IExternalEventHandler

{
    public void Execute(UIApplication app)
    {
        TaskDialog.Show("External Event", "Click Close to close.");
    }

    public string GetName()
    {
        return "External Event Example";
    }
}
```

ExternalEvent

The `ExternalEvent` class is used to create an `ExternalEvent`. An instance of this class will be returned to an external event's owner upon the event's creation. The event's owner will use this instance to signal that the event should be called by Revit. Revit will periodically check if any of the events have been signaled (raised), and will execute all events that were raised by calling the `Execute` method on the events' respective handlers.

The following example shows the implementation of an `IExternalApplication` that has a method `ShowForm()` that is called from an `ExternalCommand` (shown at the end of the code region). The `ShowForm()` method creates a new instance of the external events handler from the example above, creates a new `ExternalEvent` and then displays the modeless dialog box which will later use the passed in `ExternalEvent` object to raise events.

Code Region: Create the ExternalEvent

```
public class ExternalEventExampleApp : IExternalApplication
{
    // class instance
```

```
public static ExternalEventExampleApp thisApp = null;

// ModelessForm instance

private ExternalEventExampleDialog m_MyForm;

public Result OnShutdown(UIControlledApplication application)

{

    if (m_MyForm != null && m_MyForm.Visible)

    {

        m_MyForm.Close();

    }

    return Result.Succeeded;

}

public Result OnStartup(UIControlledApplication application)

{

    m_MyForm = null; // no dialog needed yet; the command will bring it

    thisApp = this; // static access to this application instance

    return Result.Succeeded;

}

// The external command invokes this on the end-user's request

public void ShowForm(UIApplication uiapp)

{

    // If we do not have a dialog yet, create and show it
```

```
if (m_MyForm == null || m_MyForm.IsDisposed)
{
    // A new handler to handle request posting by the dialog
    ExternalEventExample handler = new ExternalEventExample();

    // External Event for the dialog to use (to post requests)
    ExternalEvent exEvent = ExternalEvent.Create(handler);

    // We give the objects to the new dialog;
    // The dialog becomes the owner responsible for disposing them, eventually.

    m_MyForm = new ExternalEventExampleDialog(exEvent, handler);
    m_MyForm.Show();
}

}

}

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

public class Command : IExternalCommand
{
    public virtual Result Execute(ExternalCommandData commandData, ref string message, ElementSet elements)
    {
        try
        {
            ExternalEventExampleApp.thisApp.ShowForm(commandData.Application);
        }
    }
}
```

```
        return Result.Succeeded;

    }

    catch (Exception ex)

    {

        message = ex.Message;

        return Result.Failed;

    }

}

}
```

Raise Event

Once the modeless dialog is displayed, the user may interact with it. Actions in the dialog may need to trigger some action in Revit. When this happens, the `ExternalEvent.Raise()` method is called. The following example is the code for a simple modeless dialog with two buttons: one to raise our event and one to close the dialog.

Code Region: Raise the Event

```
public partial class ExternalEventExampleDialog : Form

{

    private ExternalEvent m_ExEvent;

    private ExternalEventExample m_Handler;

    public ExternalEventExampleDialog(ExternalEvent exEvent, ExternalEventExample handler)

    {

        InitializeComponent();

        m_ExEvent = exEvent;

        m_Handler = handler;

    }

}
```

```
}

protected override void OnFormClosed(FormClosedEventArgs e)
{
    // we own both the event and the handler
    // we should dispose it before we are closed

    m_ExEvent.Dispose();

    m_ExEvent = null;
    m_Handler = null;

    // do not forget to call the base class
    base.OnFormClosed(e);
}

private void closeButton_Click(object sender, EventArgs e)
{
    Close();
}

private void showMessageButton_Click(object sender, EventArgs e)
{
    m_ExEvent.Raise();
}

}
```

When the ExternalEvent.Raise() method is called, Revit will wait for an available Idling timecycle and then call the IExternalEventHandler.Execute() method. In this simple example, it will display a TaskDialog with the text "Click Close to close." as shown in the first code region above.

For a more complex example of using the External Events framework, see the sample code in the SDK under the ModelessDialog\ModelessForm_ExternalEvent folder. It uses a modeless dialog with numerous buttons and the IExternalEventHandler implementation has a public property to track which button was pressed so it can switch on that value in the Execute() method.

5.12 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.

5.12.1 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.

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

```
public void PostWallFailure()
{
    WallWarnUpdater wallUpdater = new WallWarnUpdater(new AddInId(new Guid("F
0F045A5-06E8-4C89-837D-8A8F85484953")));
    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("FB4F5AF3-4
2BB-4371-B559-FB1648D5B4D1"));

// register the new warning using FailureDefinition

FailureDefinition failDef = FailureDefinition.CreateFailureDefinition(war
nId, FailureSeverity.Warning, "Wall is too big (>100'). Performance problems
may result.");

FailureDefinitionId failId = new FailureDefinitionId(new Guid("691E5825-9
3DC-4f5c-9290-8072A4B631BC"));

FailureDefinition failDefError = FailureDefinition.CreateFailureDefinitio
n(failId, FailureSeverity.Error, "Wall is WAY too big (>200'). Performance pr
oblems may result.");

// save ids for later reference

wallUpdater.WarnId = warnId;

wallUpdater.FailureId = failId;

}
```

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.

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 FailureMessage 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.

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.

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 upda-
    ter
    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.GetElement(id) as Wall;

        Autodesk.Revit.DB.Parameter p = wall.LookupParameter("Unconnected
Height");

        if (p != null)
        {
            if (p.AsDouble() > 200)

            {
                FailureMessage failMessage = new FailureMessage(FailureI
d);

                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";
    }
}

```

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.

5.12.2 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.

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.

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.

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 GetFailureMessages() method.

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.

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;

}
}

```

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);
    }
}
}

```

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.Manual)]  
  
public class MyFailuresUI : IExternalApplication  
{  
  
    static AddInId m_appId = new AddInId(new Guid("9F179363-B349-4541-823  
F-A2DDB2B86AF3"));  
  
    public Autodesk.Revit.UI.Result OnStartup(UIControlledApplication uiC  
ontrolledApplication)  
  
    {  
  
        IFailuresProcessor myFailUI = new FailUI();  
  
        Autodesk.Revit.ApplicationServices.Application.RegisterFailur  
esProcessor(myFailUI);  
  
        return Result.Succeeded;  
    }  
  
    public Autodesk.Revit.UI.Result OnShutdown(UIControlledApplication ap  
plication)  
  
    {  
  
        return Result.Succeeded;  
    }  
  
    public class FailUI : IFailuresProcessor  
{  
  
        public void Dismiss(Document document)  
        {  
  
            // This method is being called in case of exception o  
r document destruction to  
  
            // dismiss any possible pending failure UI that may h  
ave left on the screen  
        }  
    }  
}
```

```
}

public FailureProcessingResult ProcessFailures(FailuresAccess
or failuresAccessor)

{
    IList<FailureResolutionType> resolutionTypeList =
        new List<FailureResolutionType>();

    IList<FailureMessageAccessor> failList = new List<Fai
lureMessageAccessor>();

    // 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 atte
mpted to try to prevent

        // entering infinite loop

        resolutionTypeList =
            failuresAccessor.GetAttemptedResoluti
onTypes(failure);

        if (resolutionTypeList.Count >= 3)

        {

            TaskDialog.Show("Error", "Cannot reso
lve failures - transaction will be rolled back.");

            return FailureProcessingResult.Procee
dWithRollBack;
        }
    }
}
```

```
        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;
}
}
```

5.13 Linked Files

The Revit API can determine which elements in Revit are references to external files ("linked files") and can make some modifications to how Revit loads external files.

An Element which contains an ExternalFileReference is an element which refers to some file outside of the base .rvt file. Examples include Revit links, CAD links, the element which stores the location of the keynote file, and rendering decals. Element.IsExternalFileReference() returns whether or not an element represents an external file. And Element.GetExternalFileReference() returns the ExternalFileReference for a given Element which contains information pertaining to the external file referenced by the element.

The following classes are associated with linked files in the Revit API:

- **ExternalFileReference** - A non-Element class which contains path and type information for a single external file which a Revit project references.
- **ExternalFileUtils** - A utility class which allows the user to find all external file references, get the external file reference from an element, or tell whether an element is an external file reference.
- **RevitLinkType** - An element representing a Revit file linked into a Revit project.
- **ModelPath** - A non-Element class which contains path information for a file (not necessarily a .rvt file.) Paths can be to a location on a local or network drive, or to a Revit Server location.
- **ModelPathUtils** - A utility class which provides methods for converting between strings and ModelPaths.
- **TransmissionData** - A class which stores information about all of the external file references in a document. The TransmissionData for a Revit project can be read without opening the document.

`Autodesk.Revit.DB.RevitLinkGraphicsSettings` represents settings to override display of Revit link in a view.

It has methods:

- `Revit.DB.View.GetLinkOverrides(ElementId)` - Allows users to return settings representing graphic overrides for the input element Id in the view. Accepts ElementId of a RevitLinkType or RevitLinkInstance
- `Revit.DB.View.SetLinkOverrides(ElementId, RevitLinkGraphicsSettings)` - Allows users to set graphic overrides of a RevitLinkType or RevitLinkInstance in the view.
- `Revit.DB.View.RemoveLinkOverrides(ElementId)` - Allows users to delete graphical link overrides in the current view. Accepts ElementId of a RevitLinkType or RevitLinkInstance

It has properties:

- `Revit.DB.RevitLinkGraphicsSettings.LinkedViewId` - The id of the linked view associated with RevitLinkGraphicsSettings or the invalid element Id if no view is selected.
- `Revit.DB.RevitLinkGraphicsSettings.LinkVisibilityType` - The visibility type of RevitLinkGraphicsSettings.

ModelPath

ModelPaths are paths to another file. They can refer to Revit models, or to any of Revit's external file references such as DWG links. Paths can be relative or absolute, but they must include an extension indicating the file type. Relative paths are generally relative to the currently

opened document. If the current document is workshared, paths will be treated as relative to the central model. To create a ModelPath, use one of the derived classes FilePath or ServerPath.

The class ModelPathUtils contains utility functions for converting ModelPaths to and from user-visible path strings, as well as to determine if a string is a valid server path.

Shared Coordinates let the position of one linked file be known to other linked models.

- Document.AcquireCoordinates acquires the project coordinates from a specified link instance. This works for both Revit links (RevitLinkInstance) and DWG links (ImportInstance).
- Document.PublishCoordinates - Publishes shared coordinates to a specified ProjectLocation. This method works only on Revit links. These read-only properties provide information on the geographic coordinate system of a SiteLocation. The geographic coordinate system is imported from a DWG file from AutoCAD or Civil 3D. If the SiteLocation has geographic coordinate system information, the latitude and longitude of the SiteLocation will be updated automatically when the model's Survey Point is moved.
- SiteLocation.GeoCoordinateSystemId - Gets a string corresponding to geographic coordinate system ID, such as "AMG-50" or "Beijing1954/a.GK3d-40" for the SiteLocation. The value will be the empty string if there is no coordinate system specified for the SiteLocation.
- SiteLocation.GeoCoordinateSystemDefinition - Gets an XML string describing the geographic coordinate system. The value will be the empty string if there is no coordinate system specified for the SiteLocation.

5.13.1 Revit Links

Revit documents can have links to various external files, including other Revit documents. These types of links in the Revit API are represented by the RevitLinkType and RevitLinkInstance classes. The RevitLinkType class represents another Revit Document ("link") brought into the current one ("host"), while the RevitLinkInstance class represents an instance of a RevitLinkType.

Creating New Links

To create a new Revit link, use the static RevitLinkType.Create() method which will create a new Revit link type and load the linked document and the static RevitLinkInstance.Create() method to place an instance of the link in the model. The RevitLinkType.Create() method requires a document (which will be the host), a ModelPath to the file to be linked, and a RevitLinkOptions object. The RevitLinkOptions class represents options for creating and loading a Revit link. Options include whether or not Revit will store a relative or absolute path to the linked file and the workset configuration. The WorksetConfiguration class is used to specify which, if any, worksets will be opened when creating the link. Note that the relative or absolute path determines how Revit will store the path, but the ModelPath passed into the Create() method needs a complete path to find the linked document initially.

The following example demonstrates the use of RevitLinkType.Create(). The variable `pathName` is the full path to the file on disk to be linked.

Code Region: Create new Revit Link

```
public ElementId CreateRevitLink(Document doc, string pathName)
{
    FilePath path = new FilePath(pathName);

    RevitLinkOptions options = new RevitLinkOptions(false);
    // Create new revit link storing absolute path to file
    LinkLoadResult result = RevitLinkType.Create(doc, path, options);

    return (result.ElementId);
}
```

Once the RevitLinkType is created, instances can be added to the document. In the following example, two instances of a RevitLinkType are added, offset by 100'. Until a RevitLinkInstance is created, the Revit link will show up in the Manage Links window, but the elements of the linked file will not be visible in any views.

Code Region: Create new Revit Link Instance

```
public void CreateLinkInstances(Document doc, ElementId linkTypeId)
{
    // Create revit link instance at origin
    RevitLinkInstance.Create(doc, linkTypeId);

    RevitLinkInstance instance2 = RevitLinkInstance.Create(doc, linkTypeId);
    // Offset second instance by 100 feet
    Location location = instance2.Location;
    location.Move(new XYZ(0, -100, 0));
}
```

The examples above work with files on the local disk. Below is a more complex example involving a link to a model on Revit server.

Code Region: Create new Revit Link to a model located on Revit server

```
public static void CreateLinkToServerModel(UIApplication uiApp)
{
    UIDocument uiDoc = uiApp.ActiveUIDocument;
    Document doc = uiDoc.Document;

    // Try to get the server path for the particular model on the server
    Application application = uiApp.Application;
    String hostId = application.GetRevitServerNetworkHosts().First();
    String rootFolder = "|";
    ModelPath serverPath = FindModelPathOnServer(application, hostId, rootFolder, "Wall pin model for updaters.rvt");

    using (Transaction t = new Transaction(doc, "Create link"))
    {
        t.Start();
        RevitLinkOptions options = new RevitLinkOptions(false);

        LinkLoadResult result = RevitLinkType.Create(doc, serverPath, options);
        RevitLinkInstance.Create(doc, result.ElementId);
        t.Commit();
    }
}
```

```
private static ModelPath FindModelPathOnServer(Application app, string hostId, string folderName, string fileName)

{
    // Connect to host to find list of available models (the "/contents" flag)
    XmlDictionaryReader reader = GetServerResponse(app, hostId, folderName + "/contents");

    bool found = false;

    // Look for the target model name in top level folder
    List<String> folders = new List<String>();

    while (reader.Read())
    {
        // Save a list of subfolders, if found
        if (reader.NodeType == XmlNodeType.Element && reader.Name == "Folders")
        {
            while (reader.Read())
            {
                if (reader.NodeType == XmlNodeType.EndElement && reader.Name == "Folders")
                    break;

                if (reader.NodeType == XmlNodeType.Element && reader.Name == "Name")
                {
                    reader.Read();
                    folders.Add(reader.Value);
                }
            }
        }
    }
}
```

```
        }

    }

}

// Check for a matching model at this folder level

if (reader.NodeType == XmlNodeType.Element && reader.Name == "Models")
{
    found = FindModelInServerResponse(reader, fileName);

    if (found)
        break;
}

reader.Close();

// Build the model path to match the found model on the server

if (found)
{
    // Server URLs use "|" for folder separation, Revit API uses "/"

    String folderNameFragment = folderName.Replace('|', '/');

    // Add trailing "/" if not present

    if (!folderNameFragment.EndsWith("/"))
        folderNameFragment += "/";

    // Build server path
}
```

```
        ModelPath modelPath = new ServerPath(hostId, folderNameFragment + fileName);

        return modelPath;
    }

    else
    {
        // Try subfolders

        foreach (String folder in folders)
        {

            ModelPath modelPath = FindModelPathOnServer(app, hostId, folder,
fileName);

            if (modelPath != null)

                return modelPath;
        }
    }

    return null;
}

// This string is different for each RevitServer version

private static string revitServerVersion = "/RevitServerAdminRESTService2014/
AdminRESTService.svc/";

private static XmlDictionaryReader GetServerResponse(Application app, string
hostId, string info)

{
    // Create request

    WebRequest request = WebRequest.Create("http://" + hostId + revitServerVe
rsion + info);
```

```
request.Method = "GET";

// Add the information the request needs

request.Headers.Add("User-Name", app.Username);
request.Headers.Add("User-Machine-Name", app.Username);
request.Headers.Add("Operation-GUID", Guid.NewGuid().ToString());

// Read the response

XmlDictionaryReaderQuotas quotas =
    new XmlDictionaryReaderQuotas();
XmlDictionaryReader jsonReader =
    JsonReaderWriterFactory.CreateJsonReader(request.GetResponse().GetResponseStream(), quotas);

return jsonReader;
}

private static bool FindModelInServerResponse(XmlDictionaryReader reader, string fileName)
{
    // Read through entries in this section
    while (reader.Read())
    {
        if (reader.NodeType == XmlNodeType.EndElement && reader.Name == "Models")
            break;
    }
}
```

```
if (reader.NodeType == XmlNodeType.Element && reader.Name == "Name")  
{  
    reader.Read();  
  
    String modelName = reader.Value;  
  
    if (modelName.Equals(fileName))  
    {  
        // Match found, stop looping and return  
  
        return true;  
    }  
}  
  
}  
  
return false;  
}
```

In the example below, the WorksetConfiguration is obtained, modified so that only one specified workset is opened and set back to the RevitLinkOptions prior to creating the new link.

Code Region: Create link with one workset open

```
public bool CreateRevitLinkWithOneWorksetOpen(Document doc, string pathName,  
string worksetName)  
{  
    FilePath path = new FilePath(pathName);  
  
    RevitLinkOptions options = new RevitLinkOptions(true);  
  
    // Get info on all the user worksets in the project prior to opening
```

```
    IList<WorksetPreview> worksets = WorksharingUtils.GetUserWorksetInfo(path);

    IList<WorksetId> worksetIds = new List<WorksetId>();

    // Find worksetName

    foreach (WorksetPreview worksetPrev in worksets)
    {

        if (worksetPrev.Name.CompareTo(worksetName) == 0)

        {

            worksetIds.Add(worksetPrev.Id);

            break;

        }

    }

    // close all worksets but the one specified

    WorksetConfiguration worksetConfig = new WorksetConfiguration(WorksetConfigurationOption.CloseAllWorksets);

    if (worksetIds.Count > 0)

    {

        worksetConfig.Open(worksetIds);

    }

    options.SetWorksetConfiguration(worksetConfig);

}

LinkLoadResult result = RevitLinkType.Create(doc, path, options);

RevitLinkType type = doc.GetElement(result.ElementId) as RevitLinkType;

return (result.LoadResult == LinkLoadResultType.LinkLoaded);

}
```

Whether creating or loading a link, a `LinkLoadResult` is returned. This class has a property to determine if the link was loaded. It also has an `ElementId` property that is the id of the created or loaded linked model.

`RevitLinkInstance.Create(ImportPlacement placement)` creates a new instance of a linked Revit project (`RevitLinkType`). Instances will be placed origin-to-origin or by shared coordinates according to the input placement type.

Loading and Unloading Links

`RevitLinkType` has several methods related to loading links. `Load()`, `LoadFrom()` and `Unload()` allow a link to be loaded or unloaded, or to be loaded from a new location. These methods regenerate the document. The document's Undo history will be cleared by these methods. All transaction phases (e.g. transactions, transaction groups and sub-transactions) that were explicitly started must be finished prior to calling one of these methods.

The static method `RevitLinkType.IsLoaded()` will return whether or not the link is loaded.

Getting Link Information

Each `RevitLinkType` in a document can have one or more associated `RevitLinkInstances`. The `RevitLinkInstance.GetLinkDocument()` method returns a `Document` associated with the Revit link. This document cannot be modified, meaning that operations that require a transaction or modify the document's status in memory (such as `Save` and `Close`) cannot be performed.

The associated `RevitLinkType` for a `RevitLinkInstance` can be retrieved from the document using the `ElementId` obtained from the `RevitLinkInstance.GetTypeId()` method. The `RevitLinkType` for a linked file has several methods and properties related to nested links. A document that is linked in to another document may itself have links. The `IsNested` property returns true if the `RevitLinkType` is a nested link (i.e. it has some other link as a parent), or false if it is a top-level link. The method `GetParentId()` will get the id of the immediate parent of this link, while `GetRootId()` will return the id of the top-level link which this link is ultimately linked under. Both methods will return `invalidElementId` if this link is a top-level link. The method `GetChildIds()` will return the element ids of all links which are linked directly into this one.

For example, if C is linked into document B and B in turn is linked into document A, calling `GetParentId()` for the C link will return the id of document B and calling `GetRootId()` for the C link will return the id of document A. Calling `GetChildIds()` for document A will only return the id of B's link since C is not a direct link under A.

`RevitLinkType` also has a `PathType` property which indicates if the path to the external file reference is relative to the host file's location (or to the central model's location if the host is workshared), an absolute path to a location on disk or the network, or if the path is to a Revit Server location.

The `AttachmentType` property of `RevitLinkType` indicates if the link is an attachment or an overlay. "Attachment" links are considered to be part of their parent link and will be brought along if their parent is linked into another document. "Overlay" links are only visible when their parent is opened directly.

The following example gets all `RevitLinkInstances` in the document and displays some information on them.

Code Region: Getting Link information

```
public void GetAllRevitLinkInstances(Document doc)
{
    FilteredElementCollector collector = new FilteredElementCollector(doc);
    collector.OfClass(typeof(RvitLinkInstance));

    StringBuilder linkedDocs = new StringBuilder();
    foreach (Element elem in collector)
    {
        RevitLinkInstance instance = elem as RevitLinkInstance;
        Document linkDoc = instance.GetLinkDocument();

        linkedDocs.AppendLine("FileName: " + Path.GetFileName(linkDoc.PathName));
        RevitLinkType type = doc.GetElement(instance.GetTypeId()) as RevitLinkType;
        linkedDocs.AppendLine("Is Nested: " + type.IsNestedLink.ToString());
    }

    TaskDialog.Show("Revit Links in Document", linkedDocs.ToString());
}
```

Link Geometry

Shared coordinates

The RevitLinkType methods `SavePositions()` and `HasSaveablePositions()` support saving shared coordinates changes back to the linked document. Use `HasSaveablePositions()` to determine if the link has shared positioning changes which can be saved. Call `SavePositions()` to save shared coordinates changes back to the linked document. `SavePositions()` requires an instance of the `ISaveSharedCoordinatesCallback` interface to resolve situations when Revit encounters modified links. The interface's `GetSaveModifiedLinksOption()` method determines whether Revit should save the link, not save the link, or discard shared positioning entirely.

While SavePositions() does not clear the document's undo history, it cannot be undone since it saves the link's shared coordinates changes to disk.

`ResetSharedCoordinates()` resets the shared coordinates for the host model. It provides the same functionality as the UI Command "Reset Shared Coordinates". After resetting coordinates, the following changes will take place:

- GIS coordinate system will be erased
- Shared coordinates?relationships with other linked models will be eliminated.
- The Survey Point will be moved back to the startup location, where it coincides with the Internal Origin.
- The angle between Project North and True North will be reset to 0.

Note: There will be no changes to linked models.

Conversion of geometric references

The Reference class has members related to linked files that allow conversion between Reference objects which reference only the contents of the link and Reference objects which reference the host. This allows an application, for example, to look at the geometry in the link, find the needed face, and convert the reference to that face into a reference in the host suitable for use to place a face-based instance. Also, these Reference members make it possible to obtain a reference in the host (e.g. from a dimension or family) and convert it to a reference in the link suitable for use in Element.GetGeometryObjectFromReference().

The Reference. LinkedElementId property represents the id of the top-level element in the linked document that is referred to by this reference, or InvalidElementId for references that do not refer to an element in a linked RVT file. The Reference . CreateLinkReference() method uses a RevitLinkInstance to create a Reference from a Reference in a Revit link. And the Reference. CreateReferenceInLink() method creates a Reference in a Revit Link from a Reference in the host file

Picking elements in links

The Selection methods PickObject() and PickObjects() allow the selection of objects in Revit links. To allow the user to select elements in linked files, use the ObjectType.LinkedElement enumeration value for the first parameter of the PickObject() or PickObjects(). Note that this option allows for selection of elements in links only, not in the host document.

In the example below, an ISelectionFilter is used to allow only walls to be selected in linked files.

Code Region: Selecting Elements in Linked File

```
public void SelectElementsInLinkedDoc(Autodesk.Revit.DB.Document document)
{
    UIDocument uidoc = new UIDocument(document);
```

```
Selection choices = uidoc.Selection;

// Pick one wall from Revit link

WallInLinkSelectionFilter wallFilter = new WallInLinkSelectionFilter();

Reference elementRef = choices.PickObject(ObjectType.LinkedElement, wallFilter, "Select a wall in a linked document");

if (elementRef != null)

{

    TaskDialog.Show("Revit", "Element from link document selected.");

}

}

// This filter allows selection of only a certain element type in a link instance.

class WallInLinkSelectionFilter : ISelectionFilter

{

    private RevitLinkInstance m_currentInstance = null;

    public bool AllowElement(Element e)

    {

        // Accept any link instance, and save the handle for use in AllowReference()

        m_currentInstance = e as RevitLinkInstance;

        return (m_currentInstance != null);

    }

    public bool AllowReference(Reference refer, XYZ point)

    {

        if (m_currentInstance == null)
```

```

        return false;

    // Get the handle to the element in the link

    Document linkedDoc = m_currentInstance.GetLinkDocument();

    Element elem = linkedDoc.GetElement(refer.LinkedElementId);

    // Accept the selection if the element exists and is of the correct type

    return elem != null && elem is Wall;
}

}

```

Link Graphics

Link Visibility/Graphic Override API

This functionality allows getting and setting the details of the 'Custom' option for the Visibility/Graphic Overrides of Revit Links.

`RevitLinkGraphicsSettings.IsViewRangeSupported(View)` – Checks if the view supports view range settings for RevitLinkGraphicsSettings graphic overrides.

Phase

- `RevitLinkGraphicsSettings.GetPhaseId()`
- `RevitLinkGraphicsSettings.GetPhaseType()`
- `RevitLinkGraphicsSettings.GetPhaseFilterId()`
- `RevitLinkGraphicsSettings.GetPhaseFilterType()`
- `RevitLinkGraphicsSettings.SetPhase(LinkVisibility, ElementId)` – Sets the phase and phase type of RevitLinkGraphicsSettings. Accepts LinkVisibility and ElementId of the phase from the linked document or ElementId.InvalidElementId.
- `RevitLinkGraphicsSettings.SetPhaseFilter(LinkVisibility, ElementId)` – Sets the phase filter and phase filter type of RevitLinkGraphicsSettings. Accepts LinkVisibility and ElementId of the phase filter from the linked document or ElementId.InvalidElementId.

Detail Level

- `RevitLinkGraphicsSettings.GetViewDetailLevel()`

- `RevitLinkGraphicsSettings.GetViewDetailLevelType()`
- `RevitLinkGraphicsSettings.SetViewDetailLevel(LinkVisibility, ViewDetailLevel)` – Sets the detail level and detail level type of RevitLinkGraphicsSettings. Accepts LinkVisibility and ViewDetailLevel types.

Discipline

- `RevitLinkGraphicsSettings.GetDiscipline()`
- `RevitLinkGraphicsSettings.GetDisciplineType()`
- `RevitLinkGraphicsSettings.SetDiscipline(LinkVisibility, ViewDiscipline)` – Sets the discipline and discipline type of RevitLinkGraphicsSettings. Accepts LinkVisibility and ViewDiscipline types.

5.13.2 Managing External Files

ExternalFileUtils

As its name implies, this utility class provides information about external file references. The `ExternalFileUtils.GetAllExternalFileReferences()` method returns a collection of `ElementIds` of all elements that are external file references in the document. (Note that it will not return the ids of nested Revit links; it only returns top-level references.) This utility class has two other methods, `IsExternalFileReference()` and `GetExternalFileReference()` which perform the same function as the similarly named methods of the `Element` class, but can be used when you have an `ElementId` rather than first obtaining the `Element`.

`TransmissionData` stores information on both the previous state and requested state of an external file reference. This means that it stores the load state and path of the reference from the most recent time this `TransmissionData`'s document was opened. It also stores load state and path information for what Revit should do the next time the document is opened.

As such, `TransmissionData` can be used to perform operations on external file references without having to open the entire associated Revit document. The methods `ReadTransmissionData` and `WriteTransmissionData` can be used to obtain information about external references, or to change that information. For example, calling `WriteTransmissionData` with a `TransmissionData` object which has had all references set to `LinkedFileStatus.Unloaded` would cause no references to be loaded upon next opening the document.

`TransmissionData` cannot add or remove references to external files. If `AddExternalFileReference` is called using an `ElementId` which does not correspond to an element which is an external file reference, the information will be ignored on file load.

The following example reads the `TransmissionData` for a file at the given location and sets all Revit links to be unloaded the next time the document is opened.

Code Region: Unload Revit Links

```
void UnloadRevitLinks(ModelPath location)

    /// This method will set all Revit links to be unloaded the next time the document at the given location is opened.

    /// The TransmissionData for a given document only contains top-level Revit links, not nested links.

    /// However, nested links will be unloaded if their parent links are unloaded, so this function only needs to look at the document's immediate links.

{

    // access transmission data in the given Revit file

    TransmissionData transData = TransmissionData.ReadTransmissionData(location);

    if (transData != null)

    {

        // collect all (immediate) external references in the model

        ICollection<ElementId> externalReferences = transData.GetAllExternalFileReferenceIds();

        // find every reference that is a link

        foreach (ElementId refId in externalReferences)

        {

            ExternalFileReference extRef = transData.GetLastSavedReferenceData(refId);

            if (extRef.ExternalFileReferenceType == ExternalFileReferenceType.RevitLink)

            {

                // we do not want to change neither the path nor the path-type

                // we only want the links to be unloaded (shouldLoad = false)

                transData.SetDesiredReferenceData(refId, extRef.GetPath(), extRef.PathType, false);

            }

        }

    }

}
```

```

    }

    // make sure the IsTransmitted property is set
    transData.IsTransmitted = true;

    // modified transmission data must be saved back to the model
    TransmissionData.WriteTransmissionData(location, transData);

}

else
{
    Autodesk.Revit.UI.TaskDialog.Show("Unload Links", "The document does
not have any transmission data");
}

}
}

```

Construct ModelPath for location on Revit Server

To read the TransmissionData object, you need to call the static method TransmissionData.ReadTransmissionData. It requires a ModelPath object.

There are two ways to construct a ModelPath object that refers to a central file. The first way involves using ModelPathUtils and the base ModelPath class. The steps are as follows:

1. Compose the user-visible path string of the central file: RSN:// + “relative path”.

Note: The folder separator used in the “relative path” is a forward slash(/). The correct separator is a forward slash.

2. Create a ModelPath object via the ModelPathUtils.ConvertUserVisiblePathToModelPath() method. Pass in the string composed in the previous step.
3. Read the transmission data via the TransmissionData::ReadTransmissionData() method. Pass in the ModelPath obtained in the previous step.

The following example demonstrates this method assuming a central file testmodel.rvt is stored in the root folder of Revit Server, SHACNG035WQRP.

Code Region: Constructing path to central file using ModelPath

```
[TransactionAttribute(Autodesk.Revit.Attributes.TransactionMode.Manual)]  
  
public class RevitCommandLink : IExternalCommand  
  
{  
  
    public Result Execute(ExternalCommandData commandData,  
  
        ref string messages, ElementSet elements)  
  
{  
  
    UIApplication app = commandData.Application;  
  
    Document doc = app.ActiveUIDocument.Document;  
  
    Transaction trans = new Transaction(doc, "ExComm");  
  
    trans.Start();  
  
    string visiblePath = "RSN://testmodel.rvt";  
  
    ModelPath serverPath = ModelPathUtils.ConvertUserVisiblePathToModelPath(visiblePath);  
  
    TransmissionData transData = TransmissionData.ReadTransmissionData(serverPath);  
  
    string mymessage = null;  
  
    if (transData != null)  
  
    {  
  
        //access the data in the transData here.  
  
    }  
  
    else  
  
    {  
  
        Autodesk.Revit.UI.TaskDialog.Show("Unload Links",  
  
            "The document does not have any transmission data");  
  
    }  
  
    trans.Commit();  
  
    return Result.Succeeded;
```

```

    }
}
```

The second way to construct the ModelPath object that refers to a central file is to use the child class ServerPath. This way can be used if the program knows the local server name, however, it is not recommended as the server name may be changed by the Revit user from the Revit UI. The steps are as follows:

1. Create a ServerPath object using ServerPath constructor.

```

public ServerPath GetServerPath()
{
    return new ServerPath("ServerNameOrServerIp", "relative path without the initial
forward slash");
}
```

Note: The first parameter is the server name, not the "RSN://" string. The second parameter does not include the initial forward slash. See the following sample code. The folder separator is a forward slash(/) too.

1. Read the TransmissionData object via the TransmissionData.ReadTransmissionData() method. Pass in the ServerPath obtained in the previous step

The following code demonstrates this method.

Code Region: Constructing path to central file using ServerPath

```

[TransactionAttribute(Autodesk.Revit.Attributes.TransactionMode.Manual)]
public class RevitCommand : IExternalCommand
{
    public Result Execute(ExternalCommandData commandData,
        ref string messages, ElementSet elements)
    {
        UIApplication app = commandData.Application;
```

```

Document doc = app.ActiveUIDocument.Document;

Transaction trans = new Transaction(doc, "ExComm");

trans.Start();

ServerPath serverPath = new ServerPath("SHACNG035WQRP", "testmodel.rvt");

TransmissionData transData = TransmissionData.ReadTransmissionData(server
Path);

string mymessage = null;

if (transData != null)

{

    //access the data in the transData here.

}

else

{

    Autodesk.Revit.UI.TaskDialog.Show("Unload Links",

        "The document does not have any transmission data");

}

trans.Commit();

return Result.Succeeded;

}

}

```

5.14 Performance Adviser

The performance adviser feature of the Revit API is designed to analyze a document and flag for the user any elements and/or settings that may cause performance degradation. The Performance Adviser command executes a set of rules and displays their result in a standard review warnings dialog.

The API for performance adviser consists of 2 classes:

- **PerformanceAdviser** - an application-wide object that has a dual role as a registry of rules to run in order to detect potential performance problems and an engine to execute them

- **IPerformanceAdviserRule** - an interface that allows you to define new rules for the Performance Adviser

Performance Adviser

PerformanceAdviser is used to add or delete rules to be checked, enable and disable rules, get information about rules in the list, and to execute some or all rules in the list. Applications that create new rules are expected to use AddRule() to register the new rule during application startup and DeleteRule() to deregister it during application shutdown. ExecuteAllRules() will execute all rules in the list on a given document, while ExecuteRules() can be used to execute selected rules in a document. Both methods will return a list of failure messages explaining performance problems detected in the document.

The following example demonstrates looping through all performance adviser rules and executing all the rules for a document.

Code Region: Performance Adviser

```
//Get the name of each registered PerformanceRule and then execute all of them.

public void RunAllRules(Document document)
{
    foreach (PerformanceAdviserRuleId id in PerformanceAdviser.GetPerformanceAdviser().GetAllRuleIds())
    {
        string ruleName = PerformanceAdviser.GetPerformanceAdviser().GetRuleName(id);
    }

    PerformanceAdviser.GetPerformanceAdviser().ExecuteAllRules(document);
}
```

IPerformanceAdviserRule

Create an instance of the IPerformanceAdviserRule interface to create new rules for the Performance Adviser. Rules can be specific to elements or can be document-wide rules. The following methods need to be implemented:

- GetName() - a short string naming the rule
- GetDescription() - a one to two sentence description of the rule

- `InitCheck()` -method invoked by performance advisor once in the beginning of the check. If rule checks the document as a whole rather than specific elements, the check should be performed in this method.
- `FinalizeCheck()` - method invoked by performance advisor once in the end of the check. Any problematic results found during rule execution can be reported during this message using `FailureMessage(s)`
- `WillCheckElements()` - indicates if rule needs to be executed on individual elements
- `GetElementFilter()` - retrieves a filter to restrict elements to be checked
- `ExecuteElementCheck()` - method invoked by performance advisor for each element to be checked

The following excerpt from the `PerformanceAdviserControl` sample in the Revit API SDK Samples folder demonstrates the implementation of a custom rule used to identify any doors in the document that are face-flipped. (See the sample project for the complete class implementation.)

Code Region: Implementing `IPerformanceAdviserRule`

```
public class FlippedDoorCheck : Autodesk.Revit.DB.IPerformanceAdviserRule
{
    private string m_name;
    private string m_description;
    private FailureDefinitionId m_doorWarningId;
    private FailureDefinition m_doorWarning;
    private List<ElementId> m_FlippedDoors;

    #region Constructor

    /// <summary>
    /// Set up rule name, description, and error handling
    /// </summary>
    public FlippedDoorCheck()
    {
        m_name = "Flipped Door Check";
    }
}
```

```
        m_description = "An API-based rule to search for and return any doors  
that are face-flipped";  
  
        m_doorWarningId = new Autodesk.Revit.DB.FailureDefinitionId(new Guid  
("25570B8FD4AD42baBD78469ED60FB9A3"));  
  
        m_doorWarning = Autodesk.Revit.DB.FailureDefinition.CreateFailureDefi  
nition(m_doorWarningId, Autodesk.Revit.DB.FailureSeverity.Warning, "Some door  
s in this project are face-flipped.");  
  
    }  
  
#endregion  
  
  
#region IPerformanceAdviserRule implementation  
  
/// <summary>  
  
/// Does some preliminary work before executing tests on elements. In th  
is case,  
  
/// we instantiate a list of FamilyInstances representing all doors that  
are flipped.  
  
/// </summary>  
  
/// <param name="document">The document being checked</param>  
  
public void InitCheck(Autodesk.Revit.DB.Document document)  
  
{  
  
    if (m_FlippedDoors == null)  
  
        m_FlippedDoors = new List<Autodesk.Revit.DB.ElementId>();  
  
    else  
  
        m_FlippedDoors.Clear();  
  
    return;  
  
}  
  
  
/// <summary>  
  
/// This method does most of the work of the IPerformanceAdviserRule impl  
ementation.
```

```
    /// It is called by PerformanceAdviser.

    /// It examines the element passed to it (which was previously filtered b
y the filter

    /// returned by GetElementFilter() (see below)). After checking to make
sure that the

    /// element is an instance, it checks the FacingFlipped property of the e
lement.

    ///

    /// If it is flipped, it adds the instance to a list to be used later.

    /// </summary>

    /// <param name="document">The active document</param>

    /// <param name="element">The current element being checked</param>

    public void ExecuteElementCheck(Autodesk.Revit.DB.Document document, Auto
desk.Revit.DB.Element element)

    {

        if ((element is Autodesk.Revit.DB.FamilyInstance))

        {

            Autodesk.Revit.DB.FamilyInstance doorCurrent = element as Autode
sk.Revit.DB.FamilyInstance;

            if (doorCurrent.FacingFlipped)

                m_FlippedDoors.Add(doorCurrent.Id);

        }

    }

    /// <summary>

    /// This method is called by PerformanceAdviser after all elements in doc
ument

    /// matching the ElementFilter from GetElementFilter() are checked by Exe
cuteElementCheck().
```

```
///  
  
    /// This method checks to see if there are any elements (door instances,  
    in this case) in the  
  
    /// m_FlippedDoor instance member. If there are, it iterates through tha  
t list and displays  
  
    /// the instance name and door tag of each item.  
  
    /// </summary>  
  
    /// <param name="document">The active document</param>  
  
    public void FinalizeCheck(Autodesk.Revit.DB.Document document)  
  
{  
  
    if (m_FlippedDoors.Count == 0)  
  
        System.Diagnostics.Debug.WriteLine("No doors were flipped. Test  
passed.");  
  
    else  
  
    {  
  
        //Pass the element IDs of the flipped doors to the revit failure  
reporting APIs.  
  
        Autodesk.Revit.DB.FailureMessage fm = new Autodesk.Revit.DB.Failu  
reMessage(m_doorWarningId);  
  
        fm.SetFailingElements(m_FlippedDoors);  
  
        Autodesk.Revit.DB.Transaction failureReportingTransaction = new A  
utodesk.Revit.DB.Transaction(document, "Failure reporting transaction");  
  
        failureReportingTransaction.Start();  
  
        document.PostFailure(fm);  
  
        failureReportingTransaction.Commit();  
  
        m_FlippedDoors.Clear();  
  
    }  
  
}
```

```
/// <summary>
/// Gets the description of the rule
/// </summary>
/// <returns>The rule description</returns>
public string GetDescription()
{
    return m_description;
}

/// <summary>
/// This method supplies an element filter to reduce the number of elements that PerformanceAdviser
/// will pass to GetElementCheck(). In this case, we are filtering for door elements.
/// </summary>
/// <param name="document">The document being checked</param>
/// <returns>A door element filter</returns>
public Autodesk.Revit.DB.ElementFilter GetElementFilter(Autodesk.Revit.DB.Document document)
{
    return new Autodesk.Revit.DB.ElementCategoryFilter(Autodesk.Revit.DB.BuiltInCategory.OST_Doors);
}

/// <summary>
/// Gets the name of the rule
/// </summary>
/// <returns>The rule name</returns>
```

```
public string GetName()
{
    return m_name;
}

/// <summary>
/// Returns true if this rule will iterate through elements and check the
m, false otherwise
/// </summary>
/// <returns>True</returns>

public bool WillCheckElements()
{
    return true;
}

#endregion
}
```

5.15 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.

5.15.1 Place

SiteLocation

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. When setting either the Latitude or Longitude, note that:

1. Revit will attempt to match the coordinates to a city it knows about, and if a match is found, will set the name accordingly.
2. Revit will attempt to automatically adjust the time zone value to match the new Longitude or Latitude value set using `SunAndShadowSettings.CalculateTimeZone()`. For some boundary cases, the time zone calculated may not be correct and the `TimeZone` property can be set directly if necessary.

Properties include information about the longitude, latitude, place name, and time zone. Its methods are:

- `ConvertFromProjectTime` - Converts project time to UTC time
- `ConvertToProjectTime` - Converts local time or UTC time to project time
- `IsCompatibleWith` - Checks whether the geographic coordinate system of this site is compatible with the given site
- `SiteLocation.SetGeoCoordinateSystem(string coordSystem)` - sets the Geo coordinate system for the current document

InternalOrigin

The `InternalOrigin` class represents the origin of Revit's internal coordinate system. Each Revit project contains one `InternalOrigin`. It has the following members:

- `InternalOrigin.Get(Document doc)` - Returns the internal origin of the project
- `InternalOrigin.Position` - Read-only property which returns the XYZ value of the internal coordinates. The position will always be (0,0,0)
- `InternalOrigin.SharedPosition` - Read-only property which returns the shared position of the internal origin based on the active `ProjectLocation` of its project

BasePoint

The `BasePoint` class represents the Project Base Point and Survey Point. Each Revit project contains one project base point and one survey point. The project base point represents the origin of the project coordinate system. The survey point represents the origin of the shared coordinate system.

- `BasePoint.Position` - Gets the XYZ value corresponding to the base point's position in Revit's internal coordinates
- `BasePoint.SharedPosition` - Gets the XYZ value corresponding to the base point's position in the transformed (shared) coordinates
- `BasePoint.GetProjectBasePoint()` - Gets the project base point
- `BasePoint.GetSurveyPoint()` - Gets the survey point

- `BasePoint.Clipped` gets and sets the clipped state of the survey point `BasePoint` based on the active `ProjectLocation` of its `Document`. For the project base point, it will always return false and attempting to set it will throw an exception.

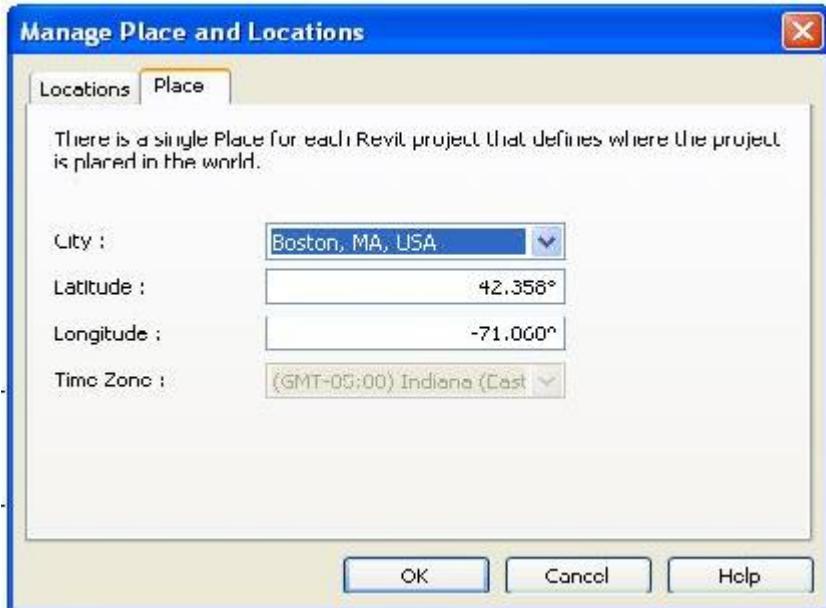


Figure 123: Project Place

5.15.2 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.

5.15.3 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.

5.15.4 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.

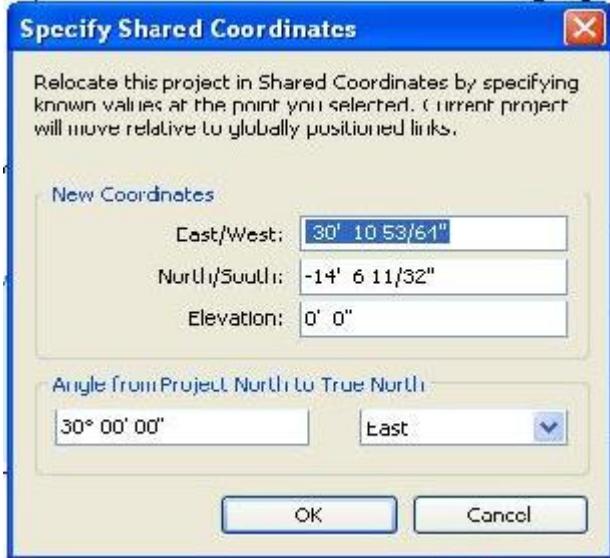


Figure 125: Point coordinates

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.

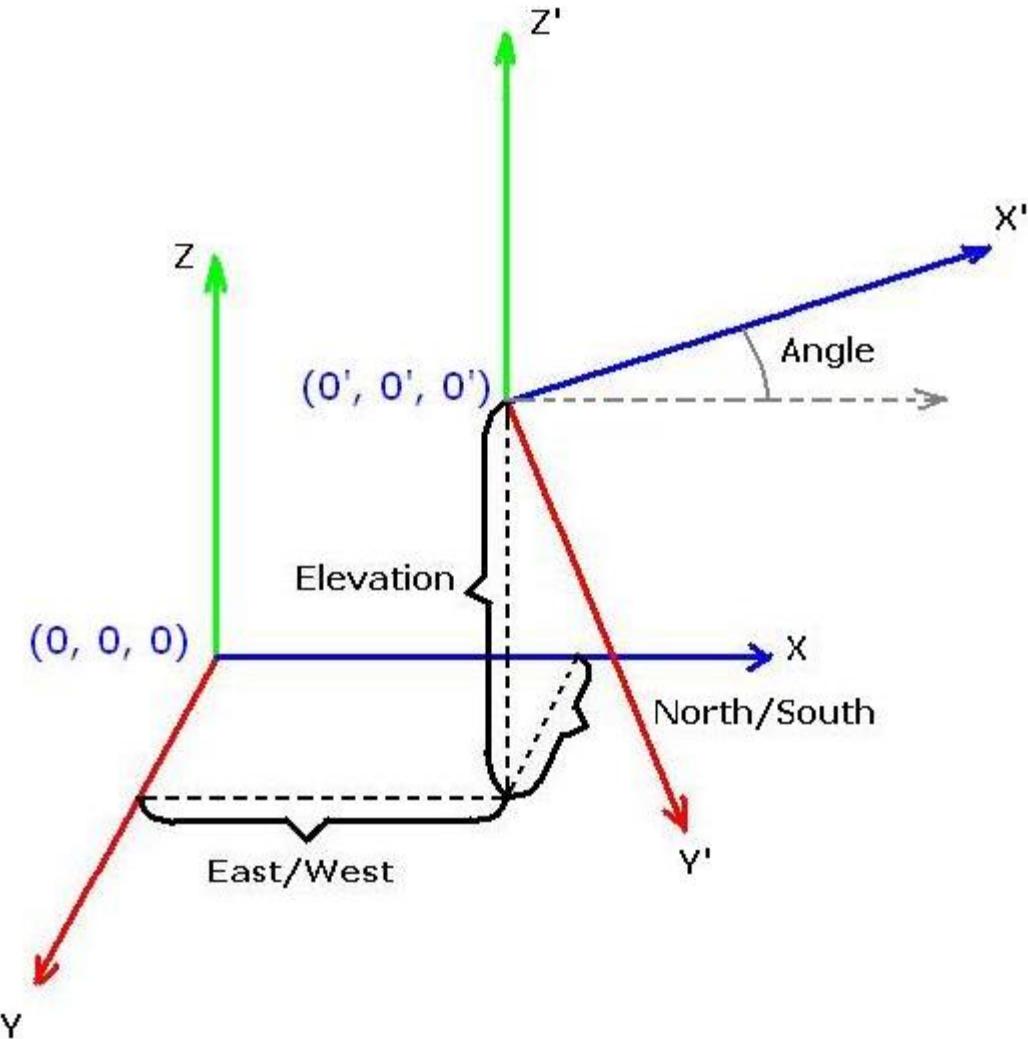


Figure 126: Geographical offset and rotation sketch map

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.GetProjectPosition(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 facto

r

SiteLocation site = projectLocation.GetSiteLocation();

string degreeSymbol = ((char)176).ToString();

prompt += "\n\t" + "Site location:";

prompt += "\n\t\t" + "Latitude: " + site.Latitude / angleRatio + degreeSy

mbol;
```

```
    prompt += "\n\t\t" + "Longitude: " + site.Longitude / angleRatio + degree
Symbol;

    prompt += "\n\t\t" + "TimeZone: " + site.TimeZone;

    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 pane.

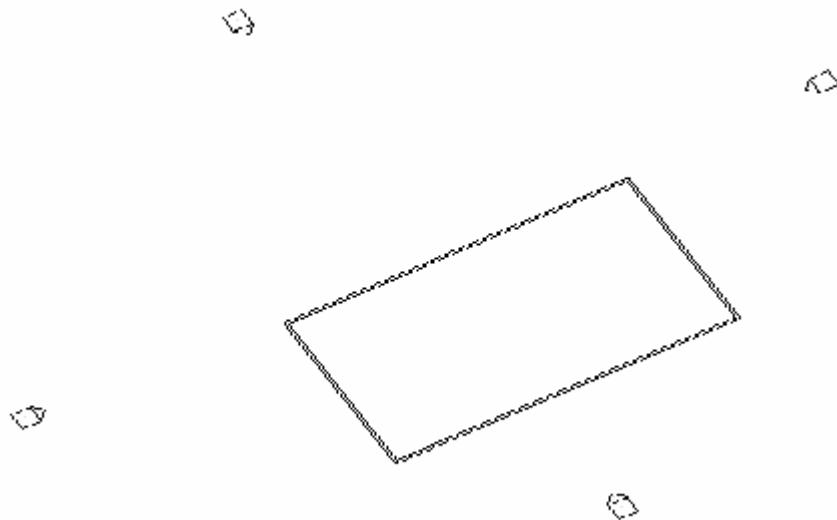


Figure 128: Project is rotated 30 degrees from Project North to True North

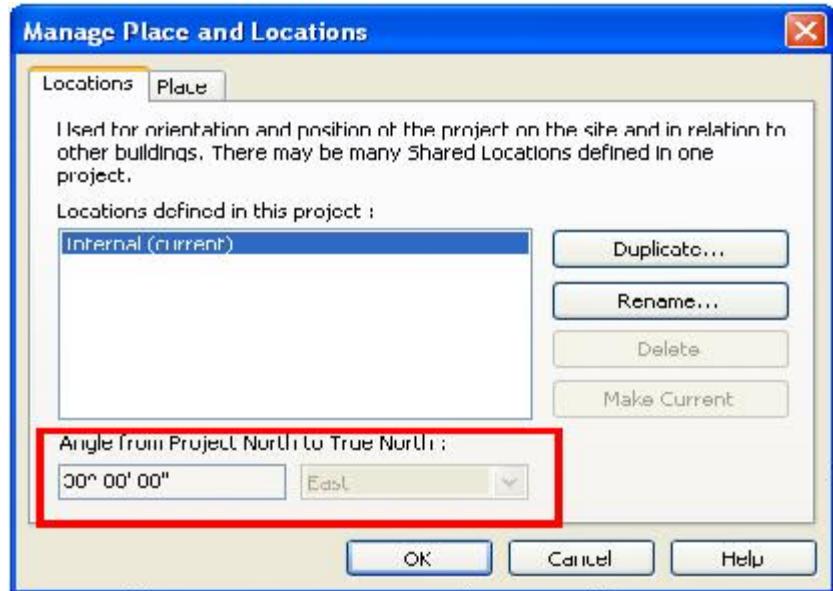


Figure 129: Project location information

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<Autodesk.Revit.DB.ElementId> elemSet = document.D  
elete(projectLocation.Id);  
  
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.

5.16 Point Clouds

The Revit API provides 2 ways to work with point clouds. The first way allows you to create new point cloud instances, read and filter points, select sub-sets of the overall points, and select points to be highlighted or isolated. The second way allows you to use your own point cloud engine and process unsupported file formats (i.e. other than .pcg, .rcp or .rcs), providing points to Revit for the user to see.

- Client API
 - Create new Point Cloud instances
 - Read & Filter Points
 - Point Set Selection
 - Control Point Cloud highlighting
 - Engine API
 - Register Point Cloud file extension
 - Provide points to Revit for rendering

5.16.1 Point Cloud Client

The point cloud client API supports read and modification of point cloud instances within Revit.

The points supplied by the point cloud instances come from the point cloud engine, which is either a built-in engine within Revit, or a [third party engine loaded as an application](#). A client point cloud API application doesn't need to be concerned with the details of how the engine stores and serves points to Revit. Instead, the client API can be used to create point clouds, manipulate their properties, and read the points found matching a given filter.

The main classes related to point clouds are:

- **PointCloudType** - type of point cloud loaded into a Revit document. Each PointCloudType maps to a single file or identifier (depending upon the type of Point Cloud Engine which governs it).
- **PointCloudInstance** - an instance of a point cloud in a location in the Revit project.
- **PointCloudFilter** - a filter determining the volume of interest when extracting points.
- **PointCollection** - a collection of points obtained from an instance and a filter.
- **PointIterator** - an iterator for the points in a PointCollection.
- **CloudPoint** - an individual point cloud point, representing an X, Y, Z location in the coordinates of the cloud, and a color.
- **PointCloudOverrides** - and its related settings classes specify graphic overrides that are stored by a view to be applied to a PointCloudInstance element, or a scan within the element.

Point cloud file paths

Two important path locations dealing with point clouds are available as read-only data:

1. PointCloudType.GetPath() - The path of the link source from which the points are loaded
2. Application.PointCloudsRootPath - The root path for point cloud files which is used by Revit to calculate relative paths to point cloud files

Creating a Point Cloud

To create a new point cloud in a Revit document, create a PointCloudType and then use it to create a PointCloudInstance. The static PointCloudType.Create() method requires the engine identifier, as it was registered with Revit by a third party, or the file extension of the point cloud file, if it is a supported file type. It also requires a file name or the identification string for a non-file based engine. In the following sample, a pcg file is used to create a point cloud in a Revit document.

Code Region: Create a point cloud from an rcs file

```
private PointCloudInstance CreatePointCloud(Document doc)
```

```
{
```

```

PointCloudType type = PointCloudType.Create(doc, "rcs", "c:\\32_cafeteri
a.rcs");

return (PointCloudInstance.Create(doc, type.Id, Transform.Identity));

}

```

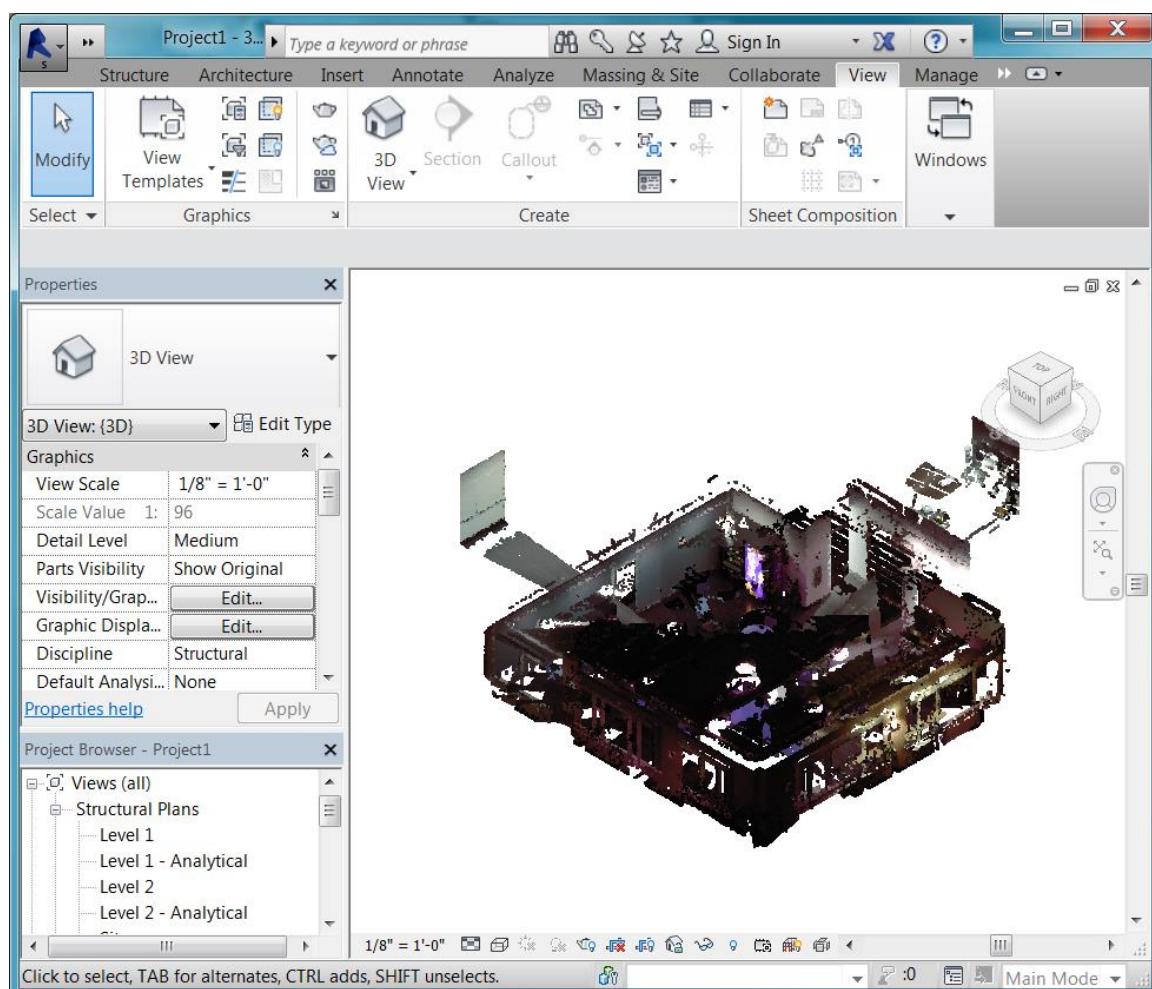


Figure: Point Cloud from 32_cafeteria.rcs

Accessing Points in a Point Cloud

There are two ways to access the points in a point cloud:

1. Iterate the resulting points directly from the PointCollection return using the `IEnumerable<CloudPoint>` interface
2. Get a pointer to the point storage of the collection and access the points directly in memory in an unsafe interface

Either way, the first step to access a collection of points from the PointCloudInstance is to use the method

- PointCloudInstance.GetPoints(PointCloudFilter filter, double averageDistance, int numPoints)

Note that as a result of search algorithms used by Revit and the point cloud engine, the exact requested number of points may not be returned.

Although the second option involves dealing with pointers directly, there may be performance improvements when traversing large buffers of points. However, this option is only possible from C# and C++/CLI.

The following two examples show how to iterate part of a point cloud using one of these two methods.

Code Region: Reading point cloud points by iteration

```
private void GetPointCloudDataByIteration(PointCloudInstance pcInstance, PointCloudFilter pointCloudFilter)

{
    // read points by iteration

    double averageDistance = 0.001;

    PointCollection points = pcInstance.GetPoints(pointCloudFilter, averageDistance, 1000); // Get points. Number of points is determined by the needs of the client

    foreach (CloudPoint point in points)
    {
        // Process each point

        System.Drawing.Color color = System.Drawing.ColorTranslator.FromWin32(point.Color);

        String pointDescription = String.Format("{0}, {1}, {2}, {3}", point.X, point.Y, point.Z, color.ToString());
    }
}
```

Code Region: Reading point cloud points by pointer

```

public unsafe void GetPointCloudDataByPointer(PointCloudInstance pcInstance,
PointCloudFilter pointCloudFilter)

{
    double averageDistance = 0.001;

    PointCollection points = pcInstance.GetPoints(pointCloudFilter, averageDistance, 1000);

    CloudPoint* pointBuffer = (CloudPoint*)points.GetPointBufferPointer().ToPointer();

    int totalCount = points.Count;

    for (int numberofPoints = 0; numberofPoints < totalCount; numberofPoints++)
    {
        CloudPoint point = *(pointBuffer + numberofPoints);

        // Process each point

        System.Drawing.Color color = System.Drawing.ColorTranslator.FromWin32(
point.Color);

        String pointDescription = String.Format("{0}, {1}, {2}, {3}", point.X, point.Y, point.Z, color.ToString());

    }
}

```

Filters

Filters are used both to limit the volume which is searched when reading points, and also to govern the display of point clouds. A PointCloudFilter can be created based upon a collection of planar boundaries. The filter will check whether a point is located on the “positive” side of each input plane, as indicated by the positive direction of the plane normal. Therefore, such filter implicitly defines a volume, which is the intersection of the positive half-spaces corresponding to all the planes. This volume does not have to be closed, but it will always be convex.

The display of point clouds can be controlled by assigning a filter to:

- PointCloudInstance.SetSelectionFilter()

Display of the filtered points will be based on the value of the property:

- PointCloudInstance.FilterAction

If it is set to None, the selection filter is ignored. If it is set to Highlight, points that pass the filter are highlighted. If it is set to Isolate, only points that pass the filter will be visible.

The following example will highlight a subset of the points in a point cloud based on its bounding box.

Code Region: Reading point cloud points by pointer

```
public PointCloudFilter ReadPointCloud(PointCloudInstance pointCloudInstance)
{
    // Filter will match 1/8 of the overall point cloud
    // Use the bounding box (filter coordinates are in the coordinates of the model)

    BoundingBoxXYZ boundingBox = pointCloudInstance.get_BoundingBox(null);
    List<Plane> planes = new List<Plane>();
    XYZ midpoint = (boundingBox.Min + boundingBox.Max) / 2.0;

    // X boundaries
    planes.Add(Plane.CreateByNormalAndOrigin(XYZ.BasisX, boundingBox.Min));
    planes.Add(Plane.CreateByNormalAndOrigin(-XYZ.BasisX, midpoint));

    // Y boundaries
    planes.Add(Plane.CreateByNormalAndOrigin(XYZ.BasisY, boundingBox.Min));
    planes.Add(Plane.CreateByNormalAndOrigin(-XYZ.BasisY, midpoint));

    // Z boundaries
    planes.Add(Plane.CreateByNormalAndOrigin(XYZ.BasisZ, boundingBox.Min));
    planes.Add(Plane.CreateByNormalAndOrigin(-XYZ.BasisZ, midpoint));

    // Create filter
```

```
PointCloudFilter filter = PointCloudFilterFactory.CreateMultiPlaneFilter(planes);

pointCloudInstance.FilterAction = SelectionFilterAction.Highlight;

return filter;

}
```

This is the result when the sample above is run on a small pipe point cloud:

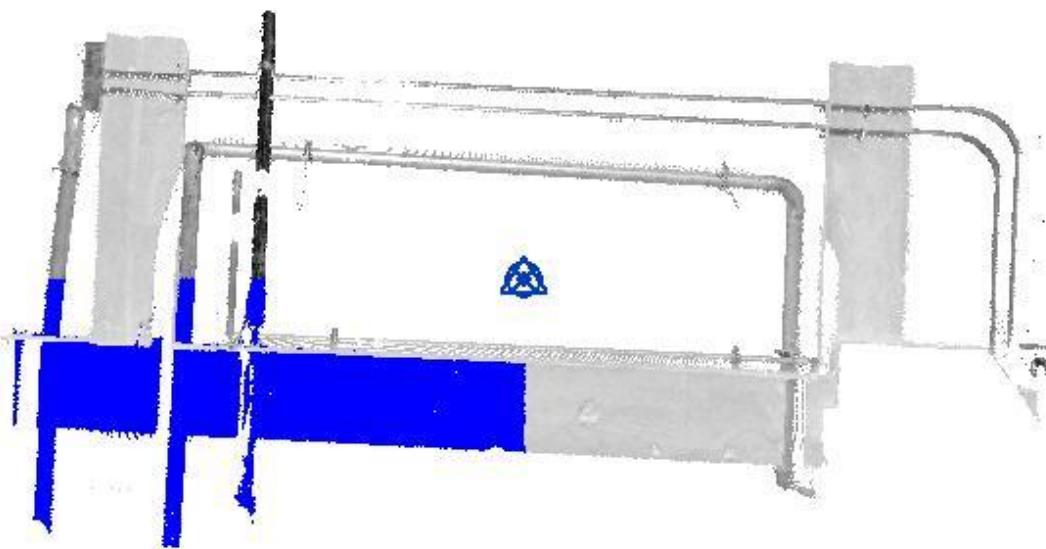


Figure: Point cloud with selection filter

The Selection.PickBox() method which invokes a general purpose two-click editor that lets the user to specify a rectangular area on the screen can be used in conjunction with a PointCloudFilter by using the resulting PickedBox to generate the planar boundaries of the filter.

Scans

An .rcp file can contain multiple scans. The method PointCloudInstance.GetScans() returns a list of scan names which can be used to set visibility and fixed color overrides independently for each scan in the PointCloudInstance. PointCloudInstance.ContainsScan() indicates whether the given scan name is contained in the point cloud instance while PointCloudInstance.GetScanOrigin() will return the origin of the given scan in model coordinates.

Regions

Scan regions are specific to Autodesk ReCap™. If a point cloud was created in ReCap, it may have regions. `PointCloudInstance.GetRegions()` returns a list of region names which can be used to set visibility and fixed color overrides independently for each region in the `PointCloudInstance`.

`PointCloudType.GetReCapProject()` provides a direct entry point to get access to an object from the ReCap SDK (`ReCapWrapper.RCProject`) from Revit. This object represents the point cloud from the RC file path stored in `PointCloudType`. The ReCap assembly `ReCapWrapper.dll` will need to be included into code using this method. The coordinate system in `RCProject` is defined by the Point Cloud. Please refer to ReCap SDK documentation for `RCProject.getCoordinateSystem()`. If you need points converted to the modeling coordinate system in Revit, you can obtain the transformation matrix from `PointCloudInstance.GetTransform()`.

Overrides

Point cloud override settings assigned to a given view can be modified using the Revit API. These settings correspond to the settings on the Point Clouds tab of the Visibility/Graphics Overrides task pane in the Revit UI. Overrides can be applied to an entire point cloud instance, or to specific scans within that instance. Options for the overrides include setting visibility for scans in the point cloud instance, setting it to a fixed color, or to color gradients based on elevation, normals, or intensity. The property `PointCloudInstance.SupportsOverrides` identifies point clouds which support override settings (clouds which are based on .rcp or .rcs files).

The following classes are involved in setting the overrides for point clouds:

- **PointCloudOverrides** - Used to get or set the `PointCloudOverrideSettings` for a `PointCloudInstance`, one of its scans, or for a particular region within the `PointCloudInstance`.
- **PointCloudOverrideSettings** - Used to get or set the visibility, color mode, and `PointCloudColorSettings`.
- **PointCloudColorSettings** - Used to assign specific colors for certain color modes to a `PointCloudInstance` element, or one of its scans. Does not apply if the `PointCloudColorMode` is `NoOverride` or `Normals`.

5.16.2 Point Cloud Engine

A custom point cloud engine can be implemented to supply cloud points to Revit.

A point cloud engine can be file-based or non-file-based. A file-based implementation requires that each point cloud be mapped to a single file on disk. Revit will allow users to create new point cloud instances in a document directly by selecting point cloud files whose extension matches the engine identifier. These files are treated as external links in Revit and may be reloaded and remapped when necessary from the Manage Links dialog.

A non-file-based engine implementation may obtain point clouds from anywhere (e.g. from a database, from a server, or from one part of a larger aggregate file). Because there is no file that the user may select, Revit's user interface will not allow a user to create a point cloud of this type. Instead, the engine provider supplies a custom command using `PointCloudType.Create()` and `PointCloudInstance.Create()` to create and place point clouds of this type. The Manage

Links dialog will show the point clouds of this type, but since there is no file associated with the point cloud, the user cannot manage, reload or remap point clouds of this type.

Regardless of the type of implementation, a custom engine implementation consists of the following:

- An implementation of `IPointCloudEngine` registered with Revit via the `PointCloudEngineRegistry`.
- An implementation of `IPointCloudAccess` which will respond to inquiries from Revit regarding the properties of a single point cloud.
- An implementation of `IPointSetIterator` which will return sets of points to Revit when requested.

In order to supply the points of the point cloud to Revit, there are two `ReadPoints()` methods which must be implemented:

- `IPointCloudAccess.ReadPoints()` - this provides a single set of points in a one-time call, either from Revit or the API. Revit uses this during some display activities including selection pre-highlighting. It is also possible for API clients to call this method directly via `PointCloudInstance.GetPoints()`.
- `IPointSetIterator.ReadPoints()` - this provides a subset of points as a part of a larger iteration of points in the cloud. Revit uses this method during normal display of the point cloud; quantities of points will be requested repeatedly until it obtains enough points or until something in the display changes. The engine implementation must keep track of which points have been returned to Revit during any given point set iteration.

See the `PointCloudEngine` folder under the `Samples` directory included with the Revit API SDK for a complete example of registering and implementing both file-based and non-file-based point cloud engines.

5.17 Transport Layer Security

When your add-on requires Internet communications and a specific security protocol is involved, if the protocol is TLS 1.0, 1.1 or 1.2 do not write any setting in the add-on. Instead take advantage of the native TLS support in Revit and its target .NET Framework. The complexities due to variety of Windows versions as well as .NET Framework versions have been handled in Revit.

If the add-on needs to specify a security protocol or its version, do not hard-code it exclusively, e.g. by directly assigning the `protocol/version` to the application-wide `property ServicePointManager.SecurityProtocol`. This will override Revit's native TLS configuration, which is critical for Revit to communicate with various Autodesk cloud services.

A **problematic** usage is `System.Net.ServicePointManager.SecurityProtocol = System.Net.SecurityProtocolType.Tls12;`

Instead, specify the protocol/version inclusively, e.g. by using bitwise OR (logical OR) on the application-wide property `ServicePointManager.SecurityProtocol`.

A **correct** usage is `System.Net.ServicePointManager.SecurityProtocol |= System.Net.SecurityProtocolType.Tls12;`

5.18 Window Handle

Two properties allow access to the handle of the Revit main window:

- Autodesk.Revit.UI.UIApplication.MainWindowHandle
- Autodesk.Revit.UI.UIControlledApplication.MainWindowHandle

This handle should be used when displaying modal dialogs and message windows to insure that they are properly parented. Use these properties instead of `System.Diagnostics.Process.GetCurrentProcess().MainWindowHandle`, which is not a reliable method for retrieving the main window handle.

5.19 Worksharing

Worksharing is a design method that allows multiple team members to work on the same project model at the same time. When worksharing is enabled, a Revit document can be subdivided into worksets, which are collections of elements in the project.

5.19.1 Worksharing Overview

When creating add-ins for Revit, it is important to understand how documents function in a workshared environment. Whether the file is local, central or managed by Revit server affects how changes to the model will impact other users, or whether a model is potentially out of date or has worksets locked by another user.

Workflow

This is the worksharing workflow from a high level perspective. When the user opens a central model, they get a local copy of the model. When they edit elements, the elements are checked out from the central model so that no one else can edit them. Local changes are only committed to the central model when a Synchronize with Central is performed. Once committed, other users can get the changes by performing a Reload Latest.

Worksets

Elements are placed in worksets. An entire workset can be checked out so that the user has exclusive editing rights to all the elements in the workset. If new elements are added, they are placed in the active workset in the local model.

Specific worksets can be opened with the model. Only opened worksets are visible, but all elements are available in the model. A workshared model may also be open "detached", in which there is no possibility of updating the central model. In this case, no workset management is required.

Worksharing types

There are three types of worksharing:

- **File-based** - The central model is accessible on disk over the network
- **Server-based** - Revit server manages the central model and possibly locally available accelerators
- **Cloud-based** - Uses the Revit Cloud Worksharing service to author Revit models in the cloud concurrently with other team members

5.19.2 Worksets

Worksets are a way to divide a set of elements in the Revit document into subsets for worksharing. There may be one or many worksets in a document.

Accessing worksets in the document

The document contains a WorksetTable which is a table containing references to all the worksets contained in that document. There is one WorksetTable for each document. There will be at least one default workset in the table, even if worksharing has not been enabled in the document. The Document.IsWorkshared property can be used to determine if worksharing has been enabled in the document. The WorksetTable class can be used to get the active workset (as shown in the example below) and to set the active workset, by calling SetActiveWorksetId()

Code Region: Get Active Workset

```
public Workset GetActiveWorkset(Document doc)
{
    // Get the workset table from the document
    WorksetTable worksetTable = doc.GetWorksetTable();

    // Get the Id of the active workset
    WorksetId activeId = worksetTable.GetActiveWorksetId();

    // Find the workset with that Id
    Workset workset = worksetTable.GetWorkset(activeId);

    return workset;
}
```

{

Filtering worksets

Since the Workset class is not derived from Element, use FilteredWorksetCollector to search, filter and iterate through a set of worksets. Conditions can be assigned to filter the worksets that are returned. If no condition is applied, this filter will access all of the worksets in the document. The WorksetKind enumerator is useful for filtering worksets as shown in the next example. The WorksetKind identifies the subdivision of worksets:

- **User** - user managed worksets for 3D instance elements
- **Family** - where family symbols & families are kept
- **Standard** - where project standards live including system family types
- **Other** - internally used worksets which should not typically be considered by applications
- **View** - contain views and view-specific elements

Code Region: Filtering Worksets

```
public void GetWorksetsInfo(Document doc)
{
    String message = String.Empty;
    // Enumerating worksets in a document and getting basic information for each
    FilteredWorksetCollector collector = new FilteredWorksetCollector(doc);

    // find all user worksets
    collector.OfKind(WorksetKind.UserWorkset);
    IList<Workset> worksets = collector.ToWorksets();

    // get information for each workset
    int count = 3; // show info for 3 worksets only
    foreach (Workset workset in worksets)
    {
```

```

message += "Workset : " + workset.Name;
message += "\nUnique Id : " + workset.UniqueId;
message += "\nOwner : " + workset.Owner;
message += "\nKind : " + workset.Kind;
message += "\nIs default : " + workset.IsDefaultWorkset;
message += "\nIs editable : " + workset.IsEditable;
message += "\nIs open : " + workset.IsOpen;
message += "\nIs visible by default : " + workset.IsVisibleByDefault;

TaskDialog.Show("GetWorksetsInfo", message);

if (0 == --count)
    break;
}

}

```

Workset properties

The Workset class represents a workset in a Revit document. As is shown in the filtering worksets example above, the Workset class provides many properties to get information about a given workset, such as the owner and whether or not the workset is editable. These properties are read-only. To change the name of an existing workset, use the static method `WorksetTable.RenameWorkset()`.

Creating worksets

The static `Workset.Create()` method can be used to create a new workset in a given document with a specified name. Worksets can only be created in a document that has worksharing enabled and the name must be unique. The static method `WorksetTable.IsWorksetNameUnique()` will confirm if a given name is unique in the document. The following example demonstrates how to create a new workset.

Code Region: Create a new workset

```
public Workset CreateWorkset(Document document)

{
    Workset newWorkset = null;

    // Worksets can only be created in a document with worksharing enabled

    if (document.IsWorkshared)
    {

        string worksetName = "New Workset";

        // Workset name must not be in use by another workset

        if (WorksetTable.IsWorksetNameUnique(document, worksetName))

        {

            using (Transaction worksetTransaction = new Transaction(document,
"Set preview view id"))

            {

                worksetTransaction.Start();

                newWorkset = Workset.Create(document, worksetName);

                worksetTransaction.Commit();

            }

        }

    }

    return newWorkset;
}
```

5.19.3 Elements in Worksets

Each element in the document must belong to one and only one workset. Each element has a WorksetId which identifies the unique workset to which it belongs. Additionally, given a

WorksetId, it is possible to get all of the elements in the document belonging to that Workset using the ElementWorksetFilter as shown below.

Code Region: ElementWorksetFilter

```
public void WorksetElements(Document doc, Workset workset)
{
    // filter all elements that belong to the given workset
    FilteredElementCollector elementCollector = new FilteredElementCollector(doc);

    ElementWorksetFilter elementWorksetFilter = new ElementWorksetFilter(workset.Id, false);

    ICollection<Element> worksetElemsounds = elementCollector.WherePasses(elementWorksetFilter).ToElements();

    // how many elements were found?
    int elementsCount = worksetElemsounds.Count;
    String message = "Element count : " + elementsCount;

    // Get name and/or Id of the elements that pass the given filter and show
    // a few of them
    int count = 5; // show info for 5 elements only
    foreach (Element ele in worksetElemsounds)
    {
        if (null != ele)
        {
            message += "\nElementId : " + ele.Id;
            message += ", Element Name : " + ele.Name;

            if (0 == --count)
        }
    }
}
```

```
        break;

    }

}

Autodesk.Revit.UI.TaskDialog.Show("ElementsOfWorkset", message);

}
```

New elements are automatically placed in the active workset in the user's local copy of the model. Since the WorksetId for an element is a read only property, use the parameter ELEM_PARTITION_PARAM. The following example demonstrates the creation of an element that is changed to belong to a different workset.

Code Region: Changing a new element's workset

```
public void ChangeWorkset(Document doc)
{
    String targetWorksetName = "Target workset";

    //Find target workset
    FilteredWorksetCollector worksetCollector = new FilteredWorksetCollector(doc);
    worksetCollector.OfKind(WorksetKind.UserWorkset);

    Workset workset = worksetCollector.FirstOrDefault<Workset>(ws => ws.Name
== targetWorksetName);

    // Workset not found, abort
    if (workset == null)
    {
        TaskDialog dialog = new TaskDialog("Error");
```

```
        dialog.MainInstruction = String.Format("There is no workset named {0}  
in the document. Aborting this operation.", targetWorksetName);  
  
        dialog.MainIcon = TaskDialogIcon.TaskDialogIconWarning;  
  
        dialog.Show();  
  
        return;  
  
    }  
  
  
    // Find "Level 1" for the new wall  
  
    FilteredElementCollector collector = new FilteredElementCollector(doc);  
  
    collector.OfClass(typeof(Level));  
  
    Level level = collector.Cast<Level>().First<Level>(lvl => lvl.Name == "Le  
vel 1");  
  
  
    using (Transaction t = new Transaction(doc, "Add elements by API"))  
    {  
  
        t.Start();  
  
  
        // Create the wall  
  
        Wall wall = Wall.Create(doc, Line.CreateBound(new XYZ(25, 0, 0), new  
XYZ(25, 15, 0)), level.Id, false);  
  
  
        // Get the parameter that stores the workset id  
  
        Parameter p = wall.GetParameter(ParameterTypeId.ElemPartitionParam);  
  
  
        // This parameter storage type is Integer  
  
        p.Set(workset.Id.IntegerValue);  
  
  
        t.Commit();
```

```
    }  
}
```

Worksharing information such as the current owner and checkout status of an element can be obtained using the `WorksharingUtils` class. It is a static class that contains utility functions related to worksharing.

Code Region: `WorksharingUtils`

```
public void GetElementWorksharingInfo(Document doc, ElementId elemId)  
{  
    String message = String.Empty;  
    message += "Element Id: " + elemId;  
  
    Element elem = doc.GetElement(elemId);  
    if(null == elem)  
    {  
        message += "Element does not exist";  
        return;  
    }  
  
    // The workset the element belongs to  
    WorksetId worksetId = elem.WorksetId;  
    message += ("\nWorkset Id : " + worksetId.ToString());  
  
    // Model Updates Status of the element  
    ModelUpdatesStatus updateStatus = WorksharingUtils.GetModelUpdatesStatus  
(doc, elemId);
```

```
message += ("\nUpdate status : " + updateStatus.ToString());  
  
// Checkout Status of the element  
  
CheckoutStatus checkoutStatus = WorksharingUtils.GetCheckoutStatus(doc, elemId);  
  
message += ("\nCheckout status : " + checkoutStatus.ToString());  
  
// Getting WorksharingTooltipInfo of a given element Id  
  
WorksharingTooltipInfo tooltipInfo = WorksharingUtils.GetWorksharingTooltipInfo(doc, elemId);  
  
message += ("\nCreator : " + tooltipInfo.Creator);  
  
message += ("\nCurrent Owner : " + tooltipInfo.Owner);  
  
message += ("\nLast Changed by : " + tooltipInfo.LastChangedBy);  
  
Autodesk.Revit.UI.TaskDialog.Show("GetElementWorksharingInfo", message);  
}
```

5.19.4 Editing Elements in Worksets

Overview

Users working in teams can encounter usability issues with Revit API add-ins beyond what a single user might experience. In particular, how an add-in is designed can prevent or create editing conflicts. For example, if an add-in attempts to edit thousands of elements, all of those elements will need to be checked out to the local user and will be unavailable to other users until a synchronize with central is performed. Or some of the elements may be checked out to other users and unavailable to be edited. This is important to keep in mind when making changes to a workshared model from the API.

The basic model editing workflow goes like this:

Action	Example	Why this is important
The user changes some elements in the model	User drags a wall	These changes are "user changes". The user must borrow these elements to make the change.
Revit regenerates additional data in the model as needed	Joined walls move, floor updates, roof updates, room updates, room tags check if they're still in their rooms	These changes are "system changes". Even though they are changed, they are still available to other users.

Most API changes are "user changes" and are treated the same as if the local user made the changes manually. This is the case whether called from an External Command, a macro, or an event. The one exception is that changes made from updaters are treated as system changes. .

Element Ownership

One way to address worksharing issues that may arise from attempting to edit an element in a workshared document is to set up FailureHandlingOptions for the transaction used to edit the element. This allows for catching and suppressing editing errors automatically and rollback the changes as shown below:

Code Region: Suppressing worksharing errors

```
public static void TryToEditGeometricElement(Element elem, bool useFailureHandlingOpts)
{
    Document doc = elem.Document;

    using (Transaction t = MakeTransaction(doc, "Move element", useFailureHandlingOpts))
    {
        t.Start();
        ElementTransformUtils.MoveElement(doc, elem.Id, new XYZ(1, 1, 0));
        t.Commit();
    }
}
```

```
}

private static Transaction MakeTransaction(Document doc, string name, bool useFailureHandlingOpts)
{
    Transaction t = new Transaction(doc, name);

    if (useFailureHandlingOpts)
    {
        FailureHandlingOptions opts = t.GetFailureHandlingOptions();
        opts.SetClearAfterRollback(true);
        opts.SetFailuresPreprocessor(new WorksharingErrorsPreprocessor());
        t.SetFailureHandlingOptions(opts);
    }

    return t;
}

private class WorksharingErrorsPreprocessor : IFailuresPreprocessor
{
    FailureProcessingResult IFailuresPreprocessor.PreprocessFailures(Failures
    Accessor failuresAccessor)
    {
        return FailureProcessingResult.Continue;
    }
}
```

The WorksharingUtils class can be used to modify element and workset ownership. The CheckoutElements() method obtains ownership for the current user of as many specified elements as possible, while the CheckoutWorksets() method does the same for worksets.

These methods are useful for attempting to checkout elements prior to performing edits. Editing is limited to elements and worksets the user owns until Reload Latest or Synchronize with Central is conducted after the model is opened. The RelinquishOwnership() method relinquishes elements and worksets owned by the current user based on the specified RelinquishOptions.

For best performance, checkout all elements or worksets and relinquish items in one big call, rather than many small calls. However, when working on a cloud model and checking out a large number of elements in one request `CheckoutElementsRequestTooLargeException` may be thrown. Checking out corresponding worksets is recommended in this case.

Note: When checking out an element, Revit may check out additional elements that are needed to make the requested element editable. For example, if an element is in a group, Revit will checkout the entire group. The following example tries to checkout the given element prior to editing and issues a message to the user if there is an issue.

Code Region: Checkout elements

```
public static bool AttemptToCheckoutInAdvance(Element element)
{
    Document doc = element.Document;
    String categoryName = element.Category.Name;

    // Checkout attempt
    ICollection<ElementId> checkedOutIds = WorksharingUtils.CheckoutElements
    (doc, new ElementId[] { element.Id });

    // Confirm checkout
    bool checkedOutSuccessfully = checkedOutIds.Contains(element.Id);

    if (!checkedOutSuccessfully)
    {
        TaskDialog.Show("Element is not checked out", "Cannot edit the " + ca
        tegoryName + " element - " +
```

```
        "it was not checked out successfully and may be checked out to another.");  
  
    return false;  
}  
  
  
// If element is updated in central or deleted in central, it is not editable  
  
ModelUpdatesStatus updatesStatus = WorksharingUtils.GetModelUpdatesStatus(doc, element.Id);  
  
if (updatesStatus == ModelUpdatesStatus.DeletedInCentral || updatesStatus == ModelUpdatesStatus.UpdatedInCentral)  
  
{  
  
    TaskDialog.Show("Element is not up to date", "Cannot edit the " + categoryName + " element - " +  
  
                    "it is not up to date with central, but it is checked out.");  
  
    return false;  
}  
  
  
return true;  
}
```

The next example demonstrates checking out all the view worksets.

Code Region: Checkout worksets

```
void CheckoutAllViewWorksets(Document doc)  
{  
  
    FilteredWorksetCollector collector = new FilteredWorksetCollector(doc);  
  
    collector.IncludeViewWorksets(true);  
  
    collector.OfType<View>().  
}
```

```
// find all view worksets

collector.OfKind(WorksetKind.ViewWorkset);

ICollection<WorksetId> viewworksets = collector.ToWorksetIds();

ICollection<WorksetId> checkoutworksets = WorksharingUtils.CheckoutWorksets(doc, viewworksets);

TaskDialog.Show("Checked out worksets", "Number of worksets checked out:
" + checkoutworksets.Count);

}
```

5.19.5 Opening a Workshared Document

The Application.OpenDocumentFile(ModelPath, OpenOptions) method can be used to set options related to opening a workshared document. In addition to options to detach from the central document or to allow a local file to be opened ReadOnly by a user other than its owner, options may also be set related to worksets. When a workshared document is opened, all system worksets are automatically opened, however user-created worksets can be specified to be opened or closed. Elements in an open workset can be expanded and displayed. However, elements in a closed workset are not displayed to avoid expanding them. By only opening worksets necessary in the current session, Revit's memory footprint is reduced, which can help with performance.

In the example below, a document is opened with two worksets specified to be opened. Note that the WorksharingUtils.GetUserWorksetInfo() method can be used to access workset information from a closed Revit document.

Code Region: Open Workshared Document

```
Document OpenDocumentWithWorksets(Application app, ModelPath projectPath)
{
    Document doc = null;

    try
    {
        // Get info on all the user worksets in the project prior to opening
```

```
IList<WorksetPreview> worksets = WorksharingUtils.GetUserWorksetInfo
(projectPath);

IList<WorksetId> worksetIds = new List<WorksetId>();

// Find two predetermined worksets

foreach (WorksetPreview worksetPrev in worksets)

{

    if (worksetPrev.Name.CompareTo("Workset1") == 0 ||

        worksetPrev.Name.CompareTo("Workset2") == 0)

    {

        worksetIds.Add(worksetPrev.Id);

    }

}

OpenOptions openOptions = new OpenOptions();

// Setup config to close all worksets by default

WorksetConfiguration openConfig = new WorksetConfiguration(WorksetCon
figurationOption.CloseAllWorksets);

// Set list of worksets for opening

openConfig.Open(worksetIds);

openOptions.SetOpenWorksetsConfiguration(openConfig);

doc = app.OpenDocumentFile(projectPath, openOptions);

}

catch (Exception e)

{

    TaskDialog.Show("Open File Failed", e.Message);

}
```

```
    return doc;  
}
```

Another option is to open the document while just opening the last viewed worksets.

Code Region: Open last viewed worksets

```
public static Document OpenLastViewed(UIApplication uiApplication)  
{  
    Application application = uiApplication.Application;  
  
    // Setup options  
    OpenOptions options1 = new OpenOptions();  
  
    // Default config opens all. Close all first, then open last viewed to g  
et the correct settings.  
    WorksetConfiguration worksetConfig = new WorksetConfiguration(WorksetConf  
igurationOption.OpenLastViewed);  
    options1.SetOpenWorksetsConfiguration(worksetConfig);  
  
    // Open the document  
    Document openedDoc = application.OpenDocumentFile(GetWSAPIModelPath("Work  
aredFileSample.rvt"), options1);  
  
    return openedDoc;  
}  
  
private static ModelPath GetWSAPIModelPath(string fileName)
```

```
{  
    // Utility to get a local path for a target model file  
  
    FileInfo filePath = new FileInfo(Path.Combine(@"C:\Documents\Revit Projects", fileName));  
  
    ModelPath mp = ModelPathUtils.ConvertUserVisiblePathToModelPath(filePath.FullName);  
  
  
    return mp;  
}
```

The following two examples demonstrate how to create a new local first from disk or from a Revit server, and then open it. Note that the sample below uses the GetWSAPIModelPath() method used in the previous example.

Code Region: Open new local from disk

```
public static Document OpenNewLocalFromDisk(UIApplication uiApplication)  
{  
    // Create new local from a disk location  
  
    ModelPath newLocalPath = GetWSAPIModelPath("LocalWorksharing.rvt");  
  
    return (OpenNewLocalFromModelPath(uiApplication.Application, GetWSAPIModelPath("NewLocalWorksharing.rvt"), newLocalPath));  
}  
  
  
private static Document OpenNewLocalFromModelPath(Application app, ModelPath centralPath, ModelPath localPath)  
{  
    // Create the new local at the given path  
  
    WorksharingUtils.CreateNewLocal(centralPath, localPath);  
}
```

```
// Select specific worksets to open

// First get a list of worksets from the unopened document

IList<WorksetPreview> worksets = WorksharingUtils.GetUserWorksetInfo(localPath);

List<WorksetId> worksetsToOpen = new List<WorksetId>();

foreach (WorksetPreview preview in worksets)

{

    // Match worksets to open with criteria

    if (preview.Name.StartsWith("0"))

        worksetsToOpen.Add(preview.Id);

}

// Setup option to open the target worksets

// First close all, then set specific ones to open

WorksetConfiguration worksetConfig = new WorksetConfiguration(WorksetConfigurationOption.CloseAllWorksets);

worksetConfig.Open(worksetsToOpen);

// Open the new local

OpenOptions options1 = new OpenOptions();

options1.SetOpenWorksetsConfiguration(worksetConfig);

Document openedDoc = app.OpenDocumentFile(localPath, options1);

return openedDoc;

}
```

The following example uses the `OpenNewLocalFromModelPath()` method demonstrated as part of the previous example.

Code Region: Open new local from Revit Server

```
/// <summary>
/// Get the server path for a particular model and open a new local copy
/// </summary>

public static Document OpenNewLocalFromServer(UIApplication uiApp)
{
    // Create new local from a server location
    Application app = uiApp.Application;

    // Get the host id/IP of the server
    String hostId = app.GetRevitServerNetworkHosts().First();

    // try to get the server path for the particular model on the server
    String rootFolder = "|";
    ModelPath serverPath = FindWSAPIModelPathOnServer(app, hostId, rootFolder, "WorksharingOnServer.rvt");

    ModelPath newLocalPath = GetWSAPIModelPath("WorksharingLocalFromServer.rvt");
    return (OpenNewLocalFromModelPath(uiApp.Application, serverPath, newLocalPath));
}

/// <summary>
/// Uses the Revit Server REST API to recursively search the folders of the Revit Server for a particular model.
/// </summary>
```

```
private static ModelPath FindWSAPIModelPathOnServer(Application app, string hostId, string folderName, string fileName)

{
    // Connect to host to find list of available models (the "/contents" flag)
    XmlDictionaryReader reader = GetResponse(app, hostId, folderName + "/contents");

    bool found = false;

    // Look for the target model name in top level folder
    List<String> folders = new List<String>();
    while (reader.Read())
    {
        // Save a list of subfolders, if found
        if (reader.NodeType == XmlNodeType.Element && reader.Name == "Folders")
        {
            while (reader.Read())
            {
                if (reader.NodeType == XmlNodeType.EndElement && reader.Name == "Folders")
                    break;
            }

            if (reader.NodeType == XmlNodeType.Element && reader.Name == "Name")
            {
                reader.Read();
                folders.Add(reader.Value);
            }
        }
    }
}
```

```
        }

    }

    // Check for a matching model at this folder level

    if (reader.NodeType == XmlNodeType.Element && reader.Name == "Models")
    {

        found = FindModelInServerResponseJson(reader, fileName);

        if (found)
            break;

    }

}

reader.Close();

// Build the model path to match the found model on the server

if (found)
{
    // Server URLs use "|" for folder separation, Revit API uses "/"

    String folderNameFragment = folderName.Replace('|', '/');

    // Add trailing "/" if not present

    if (!folderNameFragment.EndsWith("/"))
        folderNameFragment += "/";

    // Build server path

    ModelPath modelPath = new ServerPath(hostId, folderNameFragment + fil
eName);
}
```

```
        return modelPath;

    }

    else

    {

        // Try subfolders

        foreach (String folder in folders)

        {

            ModelPath modelPath = FindWSAPIModelPathOnServer(app, hostId, fol der, fileName);

            if (modelPath != null)

                return modelPath;

        }

    }

    return null;

}

// This string is different for each RevitServer version

private static string s_revitServerVersion = "/RevitServerAdminRESTService201 4/AdminRESTService.svc/";



/// <summary>

/// Connect to server to get list of available models and return server respo nse

/// </summary>

private static XmlDictionaryReader GetResponse(Application app, string hostI d, string info)

{
```

```
// Create request

WebRequest request = WebRequest.Create("http://" + hostId + s_revitServer
Version + info);

request.Method = "GET";

// Add the information the request needs

request.Headers.Add("User-Name", app.Username);

request.Headers.Add("User-Machine-Name", app.Username);

request.Headers.Add("Operation-GUID", Guid.NewGuid().ToString());

// Read the response

XmlDictionaryReaderQuotas quotas =
    new XmlDictionaryReaderQuotas();

XmlDictionaryReader jsonReader =
    JsonReaderWriterFactory.CreateJsonReader(request.GetResponse().GetRes
ponseStream(), quotas);

return jsonReader;

}

/// <summary>
/// Read through server response to find particular model
/// </summary>

private static bool FindModelInServerResponseJson(XmlDictionaryReader reader,
    string fileName)

{
    // Read through entries in this section
```

```

        while (reader.Read())
    {
        if (reader.NodeType == XmlNodeType.EndElement && reader.Name == "Models")
            break;

        if (reader.NodeType == XmlNodeType.Element && reader.Name == "Name")
        {
            reader.Read();
            String modelName = reader.Value;
            if (modelName.Equals(fileName))
            {
                // Match found, stop looping and return
                return true;
            }
        }
    }

    return false;
}

```

Open detached from central

If an add-in will be working on a workshared file but does not need to make permanent changes, it can open the model detached from the central file.

Code Region: Open detached

```

private static Document OpenDetached(Application application, ModelPath modelPath)
{

```

```
OpenOptions options1 = new OpenOptions();

    options1.DetachFromCentralOption = DetachFromCentralOption.DetachAndDiscardWorksets;

    Document openedDoc = application.OpenDocumentFile(modelPath, options1);

    return openedDoc;
}
```

If an application only needs read-only access to a server file, the example below demonstrates how to copy the server model locally and open it detached. Note this code sample re-uses methods demonstrated in previous examples.

Code Region: Copy and open detached

```
public static Document CopyAndOpenDetached(UIApplication uiApp)
{
    // Copy a server model locally and open detached
    Application application = uiApp.Application;
    String hostId = application.GetRevitServerNetworkHosts().First();

    // Try to get the server path for the particular model on the server
    String rootFolder = "|";
    ModelPath serverPath = FindWSAPIModelPathOnServer(application, hostId, rootFolder, "ServerModel.rvt");

    // For debugging
    String sourcePath = ModelPathUtils.ConvertModelPathToUserVisiblePath(serverPath);
```

```

    // Setup the target location for the copy

    ModelPath localPath = GetWSAPIModelPath("CopiedModel.rvt");

    // Copy, allowing overwrite

    application.CopyModel(serverPath, ModelPathUtils.ConvertModelPathToUserVisiblePath(localPath), true);

    // Open the copy as detached

    Document openedDoc = OpenDetached(application, localPath);

    return openedDoc;
}

```

5.19.6 Visibility and Display

Visibility

A workset's visibility can be set for a particular view using `View.SetWorksetVisibility()`. The `WorksetVisibility` options are `Visible` (it will be visible if the workset is open), `Hidden`, and `UseGlobalSetting` (indicating not to override the setting for the view). The corresponding `View.GetWorksetVisibility()` method retrieves the current visibility settings for a workset in that view. However, this method does not consider whether the workset is currently open. To determine if a workset is visible in a View, including taking into account whether the workset is open or closed, use `View.IsWorksetVisible()`.

The class `WorksetDefaultVisibilitySettings` manages default visibility of worksets in a document. It is not available for family documents. If worksharing is disabled in a document, all elements are moved into a single workset; that workset, and any worksets (re)created if worksharing is re-enabled, is visible by default regardless of any current settings.

The following example hides a workset in a given view and hides it by default in other views.

Code Region: Hide a Workset

```
public void HideWorkset(Document doc, View view, WorksetId worksetId)
```

```

{

    // get the current visibility

    WorksetVisibility visibility = view.GetWorksetVisibility(worksetId);

    // and set it to 'Hidden' if it is not hidden yet

    if (visibility != WorksetVisibility.Hidden)

    {

        view.SetWorksetVisibility(worksetId, WorksetVisibility.Hidden);

    }

    // Get the workset's default visibility

    WorksetDefaultVisibilitySettings defaultVisibility = WorksetDefaultVisibilitySettings.GetWorksetDefaultVisibilitySettings(doc);

    // and making sure it is set to 'false'

    if (true == defaultVisibility.IsWorksetVisible(worksetId))

    {

        defaultVisibility.SetWorksetVisibility(worksetId, false);

    }

}

```

Display Modes

In addition to getting and setting information about the workset visibility, the View class also provides methods to access information on the worksharing display mode and settings. The WorksharingDisplayMode enumeration indicates which mode a view is in, if any:

Member Name	Description
Off	No active worksharing display mode.

CheckoutStatus The view is displaying the checkout status of elements.

Owners The view is displaying the specific owners of elements.

ModelUpdates The view is displaying model updates.

Worksets The view is displaying which workset each element is assigned to.

Code Region: Activate different worksharing display modes

```
public Result Execute(ExternalCommandData commandData, ref string message, ElementSet elements)
{
    View activeView = commandData.View;
    Document doc = activeView.Document;

    // Prepare settings
    Color red = new Color(0xFF, 0x00, 0x00);

    WorksharingDisplayGraphicSettings settingsToApply = new WorksharingDisplayGraphicSettings(true, red);

    // Toggle mode based on the current mode
    using (Transaction t = new Transaction(doc, "Toggle display mode"))
    {
        t.Start();

        WorksharingDisplaySettings settings = WorksharingDisplaySettings.GetOrCreateWorksharingDisplaySettings(doc);

        switch (activeView.GetWorksharingDisplayStyle())
    }
```

```
{  
    case WorksharingDisplayMode.Off:  
        activeView.SetWorksharingDisplayMode(WorksharingDisplayMode.CheckoutStatus);  
        settings.SetGraphicOverrides(CheckoutStatus.OwnedByOtherUser,  
            settingsToApply);  
        break;  
    case WorksharingDisplayMode.CheckoutStatus:  
        activeView.SetWorksharingDisplayMode(WorksharingDisplayMode.ModelUpdates);  
        settings.SetGraphicOverrides(ModelUpdatesStatus.UpdatedInCentral,  
            settingsToApply);  
        break;  
    case WorksharingDisplayMode.ModelUpdates:  
        activeView.SetWorksharingDisplayMode(WorksharingDisplayMode.Owners);  
        settings.SetGraphicOverrides("Target user", settingsToApply);  
        break;  
    case WorksharingDisplayMode.Owners:  
        activeView.SetWorksharingDisplayMode(WorksharingDisplayMode.Worksets);  
        settings.SetGraphicOverrides(doc.GetWorksetTable().GetActiveWorksetId(),  
            settingsToApply);  
        break;  
    case WorksharingDisplayMode.Worksets:  
        activeView.SetWorksharingDisplayMode(WorksharingDisplayMode.Off);  
        break;  
}
```

```

        t.Commit();

    }

    return Result.Succeeded;
}

```

Graphics Settings

The WorksharingDisplaySettings class controls how elements will appear when they are displayed in any of the worksharing display modes. The colors stored in these settings are a common setting and are shared by all users in the model. Whether a given color is applied is specific to the current user and will not be shared by other users. Note that these settings are available even in models that are not workshared. This is to allow pre-configuring the display settings before enabling worksets so that they can be stored in template files.

Code Region: Worksharing Display Graphic Settings

```

public WorksharingDisplayGraphicSettings GetWorksharingDisplaySettings(Document doc, String userName, WorksetId worksetId, bool ownedbyCurrentUser)
{
    WorksharingDisplayGraphicSettings graphicSettings;

    // get or create a WorksharingDisplaySettings current active document
    WorksharingDisplaySettings displaySettings = WorksharingDisplaySettings.GetOrCreateWorksharingDisplaySettings(doc);

    // get graphic settings for a user, if specified
    if (!String.IsNullOrEmpty(userName))
        graphicSettings = displaySettings.GetGraphicOverrides(userName);

    // get graphicSettings for a workset, if specified
}

```

```
    else if (worksetId != WorksetId.InvalidWorksetId)
        graphicSettings = displaySettings.GetGraphicOverrides(worksetId);

        // get graphic settings for the OwnedByCurrentUser status

        else if (ownedbyCurrentUser)
            graphicSettings = displaySettings.GetGraphicOverrides(CheckoutStatus.OwnedByCurrentUser);

        // otherwise get graphic settings for the CurrentWithCentral status

        else
            graphicSettings = displaySettings.GetGraphicOverrides(ModelUpdatesStatus.CurrentWithCentral);

    return graphicSettings;
}
```

The overloaded method `WorksharingDisplaySettings.SetGraphicOverrides()` sets the graphic overrides assigned to elements based on the given criteria.

Code Region: Graphic Overrides

```
public void SetWorksharingDisplaySettings(Document doc, WorksetId worksetId,
String userName)

{
    String message = String.Empty;

    // get or create a WorksharingDisplaySettings current active document

    WorksharingDisplaySettings displaySettings = WorksharingDisplaySettings.GetOrCreateWorksharingDisplaySettings(doc);
```

```
// set a new graphicSettings for CheckoutStatus - NotOwned  
  
WorksharingDisplayGraphicSettings graphicSettings = new WorksharingDisplayGraphicSettings(true, new Color(255, 0, 0));  
  
displaySettings.SetGraphicOverrides(CheckoutStatus.NotOwned, graphicSettings);  
  
  
// set a new graphicSettings for ModelUpdatesStatus - CurrentWithCentral  
  
graphicSettings = new WorksharingDisplayGraphicSettings(true, new Color(128, 128, 0));  
  
displaySettings.SetGraphicOverrides(ModelUpdatesStatus.CurrentWithCentral, graphicSettings);  
  
  
// set a new graphicSettings by a given userName  
  
graphicSettings = new WorksharingDisplayGraphicSettings(true, new Color(0, 255, 0));  
  
displaySettings.SetGraphicOverrides(userName, graphicSettings);  
  
  
// set a new graphicSettings by a given workset Id  
  
graphicSettings = new WorksharingDisplayGraphicSettings(true, new Color(0, 0, 255));  
  
displaySettings.SetGraphicOverrides(worksetId, graphicSettings);  
}
```

The `WorksharingDisplaySettings` class can also be used to control which users are listed in the displayed users for the document. The `RemoveUsers()` method removes users from the list of displayed users and permanently discards any customization of the graphics. Only users who do not own any elements in the document can be removed. The `RestoreUsers()` method adds removed users back to the list of displayed users and permits customization of the graphics for those users. Note that any restored users will be shown with default graphic overrides and any customizations that existed prior to removing the user will not be restored.

Code Region: Removing Users

```
public void RemoveAndRestoreUsers(Document doc)

{
    // get or create a WorksharingDisplaySettings current active document
    WorksharingDisplaySettings displaySettings = WorksharingDisplaySettings.GetOrCreateWorksharingDisplaySettings(doc);

    // get all users with GraphicOverrides
    ICollection<string> users = displaySettings.GetAllUsersWithGraphicOverrides();

    // remove the users from the display settings (they will not have graphic
    // overrides anymore)
    ICollection<string> outUserList;
    displaySettings.RemoveUsers(doc, users, out outUserList);

    // show the current list of removed users
    ICollection<string> removedUsers = displaySettings.GetRemovedUsers();

    String message = "Current list of removed users: ";
    if (removedUsers.Count > 0 )
    {
        foreach (String user in removedUsers)
        {
            message += "\n" + user;
        }
    }
    else
```

```
{  
    message = "[Empty]";  
}  
  
TaskDialog.Show("Users Removed", message);  
  
// restore the previously removed users  
int number = displaySettings.RestoreUsers(outUserList);  
  
// again, show the current list of removed users  
// it should not contain the users that were restored  
removedUsers = displaySettings.GetRemovedUsers();  
  
message = "Current list of removed users: ";  
if (removedUsers.Count > 0)  
{  
    foreach (String user in removedUsers)  
    {  
        message += "\n" + user;  
    }  
}  
else  
{  
    message = "[Empty]";  
}
```

```

    TaskDialog.Show("Removed Users Restored", message);
}

```

5.19.7 Workshared File Management

There are several Document methods for use with a workshared project file.

Enable Worksharing

If a document is not already workshared, which can be determined from the Document.IsWorkshared property, worksharing can be enabled via the Revit API using the Document.EnableWorksharing() method. The document's Undo history will be cleared by this command, therefore this command and others executed before it cannot be undone. Additionally, all transaction phases (e.g. transactions, transaction groups and sub-transactions) that were explicitly started must be finished prior to calling EnableWorksharing().

Worksharing in the Cloud

- Document.EnableCloudWorksharing() converts a cloud model into a workshared cloud model.
- Document.CanEnableCloudWorksharing() can be used to check whether this operation is valid on a given model.

Reload Latest

The method Document.ReloadLatest() retrieves changes from the central model (due to one or more synchronizations with central) and merges them into the current session.

The following examples uses ReloadLatest() to update the current session after confirming with the user that it is OK to do so.

Code Region: Reload from Central

```

public static void ReloadLatestWithMessage(Document doc)
{
    // Tell user what we're doing
    TaskDialog td = new TaskDialog("Alert");
}

```

```

        td.MainInstruction = "Application 'Automatic element creator' needs to re
load changes from central in order to proceed./";

        td.MainContent = "This will update your local with all changes currently
in the central model. This operation " +
                          "may take some time depending on the number of change
s available on the central./";

        td.CommonButtons = TaskDialogCommonButtons.Ok | TaskDialogCommonButtons.C
ancel;

TaskDialogResult result = td.Show();

if (result == TaskDialogResult.Ok)
{
    // There are no currently customizable user options for ReloadLatest.
    doc.ReloadLatest(new ReloadLatestOptions());

    TaskDialog.Show("Proceeding...", "Reload operation completed, proceed
ing with updates.");
}

else
{
    TaskDialog.Show("Canceled.", "Reload operation canceled, so changes w
ill not be made. Return to this command later when ready to reload.");
}
}

```

Synchronizing with Central Model

The method Document.SynchronizeWithCentral() reloads any changes from the central model so that the current session is up to date and then saves local changes back to central. A save to central is performed even if no changes were made.

When using SynchronizeWithCentral(), options can be specified for accessing the central model as well as synchronizing with it. The main option for accessing the central is to determine how the call should behave if the central model is locked. Since the synchronization requires a temporary lock on the central model, it cannot be performed if the model is already locked. The

default behavior is to wait and repeatedly try to lock the central model in order to proceed with the synchronization. This behavior can be overridden using the `TransactWithCentralOptions` parameter of the `SynchronizeWithCentral()` method.

The `SynchronizeWithCentralOptions` parameter of the method is used to set options for the actual synchronization, such as whether elements or worksets owned by the current user should be relinquished during synchronization.

In the following example, an attempt is made to synchronize with a central model. If the central model is locked, it will immediately give up.

Code Region: Synchronize with Central

```
public void SyncWithoutRelinquishing(Document doc)
{
    // Set options for accessing central model
    TransactWithCentralOptions transOpts = new TransactWithCentralOptions();

    SynchLockCallback transCallBack = new SynchLockCallback();

    // Override default behavior of waiting to try again if the central model
    // is locked
    transOpts.SetLockCallback(transCallBack);

    // Set options for synchronizing with central
    SynchronizeWithCentralOptions syncOpts = new SynchronizeWithCentralOptions();

    // Sync without relinquishing any checked out elements or worksets
    RelinquishOptions relinquishOpts = new RelinquishOptions(false);

    syncOpts.SetRelinquishOptions(relinquishOpts);

    // Do not automatically save local model after sync
    syncOpts.SaveLocalAfter = false;

    syncOpts.Comment = "Changes to Workset1";

    try
```

```
{  
    doc.SynchronizeWithCentral(transOpts, syncOpts);  
}  
  
catch (Exception e)  
{  
    TaskDialog.Show("Synchronize Failed", e.Message);  
}  
}  
  
}  
  
  
class SynchLockCallback : ICentralLockedCallback  
{  
    // If unable to lock central, give up rather than waiting  
    public bool ShouldWaitForLockAvailability()  
    {  
        return false;  
    }  
}
```

In the next example, the user is given a message prior to synching, and is given options on whether to relinquish all elements when synchronizing, or keep worksets checked out.

Code Region: Synchronize with Central With Message

```
public static void SynchWithCentralWithMessage(Document doc)  
{  
    // Checkout workset (for use with "keep checked out worksets" option later)  
}
```

```
FilteredWorksetCollector fwc = new FilteredWorksetCollector(doc);

fwc.OfKind(WorksetKind.UserWorkset);

Workset workset1 = fwc.First<Workset>(ws => ws.Name == "Workset1");

WorksharingUtils.CheckoutWorksets(doc, new WorksetId[] { workset1.Id });

// Make a change

using (Transaction t = new Transaction(doc, "Add Level"))

{

    t.Start();

    Level.Create(doc, 100);

    t.Commit();

}

// Tell user what we're doing

TaskDialog td = new TaskDialog("Alert");

td.MainInstruction = "Application 'Automatic element creator' has made changes and is prepared to synchronize with central.';

td.MainContent = "This will update central with all changes currently made in the project by the application or by the user. This operation " +

                    "may take some time depending on the number of changes made by the app and by the user.";

td.AddCommandLink(TaskDialogCommandLinkId.CommandLink1, "Do not synchronize at this time.");

td.AddCommandLink(TaskDialogCommandLinkId.CommandLink2, "Synchronize and relinquish all elements.");

td.AddCommandLink(TaskDialogCommandLinkId.CommandLink3, "Synchronize but keep checked out worksets.");
```

```
        td.DefaultButton = TaskDialogResult.CommandLink1;

        TaskDialogResult result = td.Show();

        switch (result)
        {
            case TaskDialogResult.CommandLink1:
            default:
                {
                    // Do not synch. Nothing to do.

                    break;
                }
            case TaskDialogResult.CommandLink2:
            case TaskDialogResult.CommandLink3:
                {
                    // Prepare to synch

                    // TransactWithCentralOptions has to do with the behavior related to locked or busy central models.

                    // We'll use the default behavior.

                    TransactWithCentralOptions twcOpts = new TransactWithCentralOptions();
                }
            case TaskDialogResult.CommandLink4:
                {
                    // Setup synch-with-central options (add a comment about our change)

                    SynchronizeWithCentralOptions swcOpts = new SynchronizeWithCentralOptions();

                    swcOpts.Comment = "Synchronized by 'Automatic element creator' with user acceptance.";
                }
        }
    }
}
```

```

        if (result == TaskDialogResult.CommandLink3)

    {

        // Setup relinquish options to keep user worksets checked
        out

        RelinquishOptions rOptions = new RelinquishOptions(true);

        rOptions.UserWorksets = false;

        swcOpts.SetRelinquishOptions(rOptions);

    }

    doc.SynchronizeWithCentral(twcOpts, swcOpts);

    break;

}

}

}

```

Create New Local Model

The WorksharingUtils.CreateNewLocal() method copies a central model to a new local file. This method does not open the new file. For an example of creating a new local model and opening it, see [Opening a Workshared Document](#)

6 Appendices

6.1 Glossary

6.1.1 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.

6.1.2 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.

6.1.3 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.

6.1.4 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.

6.1.5 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.

6.1.6 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.

6.1.7 Namespace

A namespace is an organizational unit used to group similar and/or functionally related classes together.

6.1.8 Overloading

Method overloading is when different methods (functions) of the same name are invoked with different types and/or numbers of parameters passed.

6.1.9 Properties

Properties are data members of a class accessible to the class user. They are also known as public members or variables.

Some properties are read only (support Get() method) and some are modifiable (support Set() method).

Example: "X" is a read-only property on the XYZ class. "Name" is a modifiable property on the Zone class.

6.1.10 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.

6.1.11 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.

6.1.12 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.

6.1.13 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.

6.1.14 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.

6.1.15 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.

6.2 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.

6.2.1 Create a New Project

The first step in writing a VB.NET program with Visual Studio 2022 is to choose a project type and create a new project.

1. From the File menu, select New ➤ Project....
2. In the Installed Templates frame, under Other Languages, click Visual Basic.
3. In the right-hand frame, select Class Library. This example assumes that your project location is: C:\Samples.
4. In the Name field, type HelloWorld as the project name.
5. Ensure that the .NET 8.0 is selected at the top of the dialog.
6. Click OK.

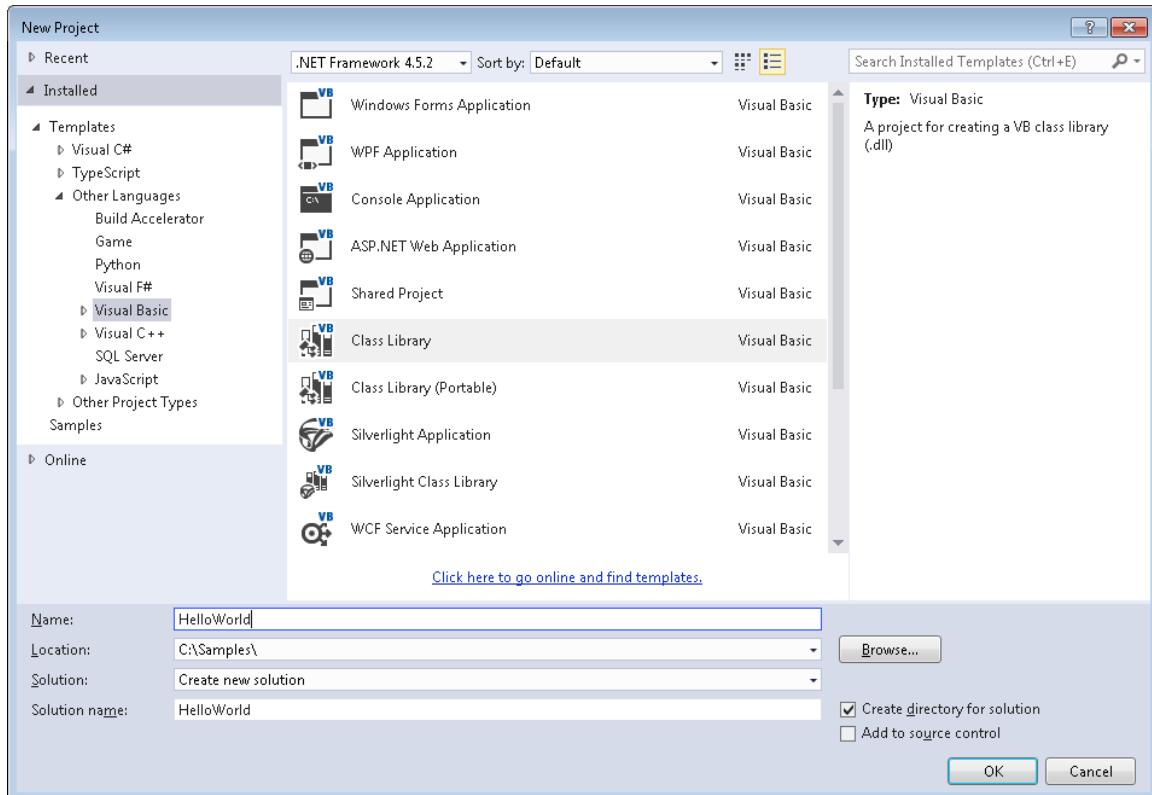


Figure 172: New Project dialog box

6.2.2 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 on References under the project name in the Solution Explorer to display a context menu.
2. From the context menu, select Add Reference to open the Reference Manager dialog box.

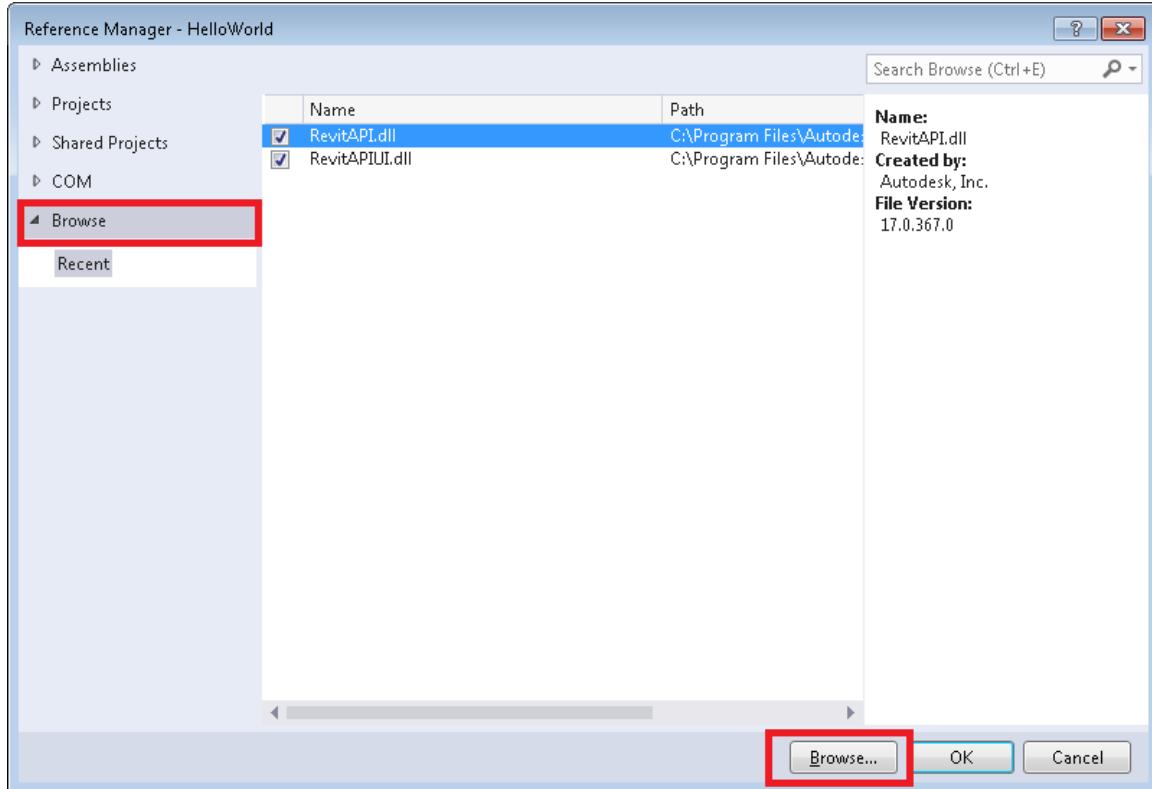


Figure: Reference Manager

3. In the Reference Manager dialog box, click the Browse tab, then click the Browse button.
4. 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 2018.
5. Click OK to add the reference.
6. Repeat steps above to add a reference to RevitAPIUI.dll, which is in the same folder as Revit API.dll.

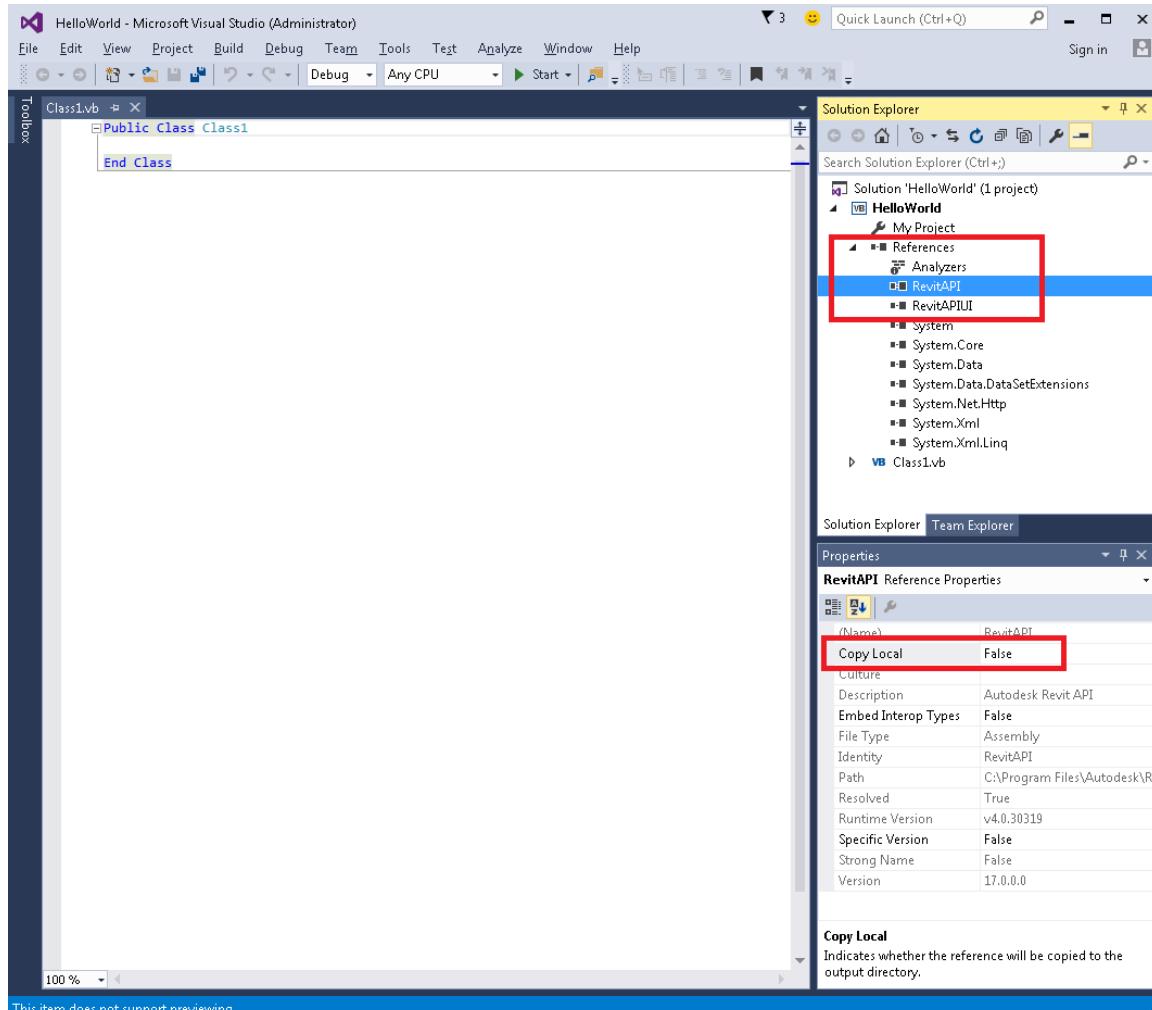


Figure: Add references

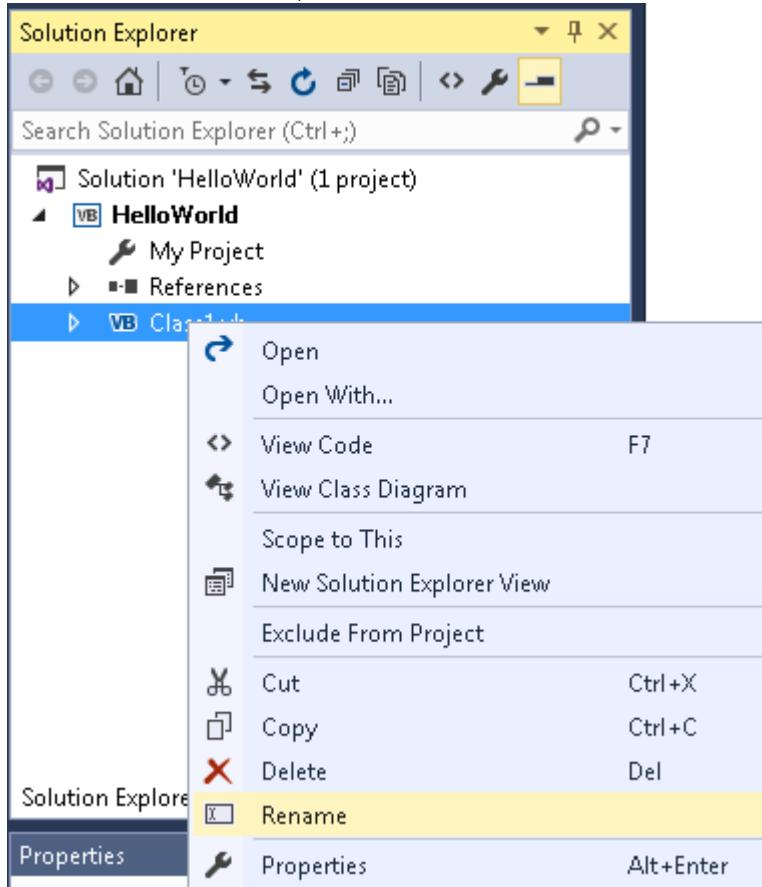
7. Click OK to close the Reference Manager dialog.
8. To complete the process, click RevitAPI under References in the Solution Explorer. Set Copy Local to False in the Properties frame. Repeat for RevitAPIUI.

6.2.3 Change the Class Name

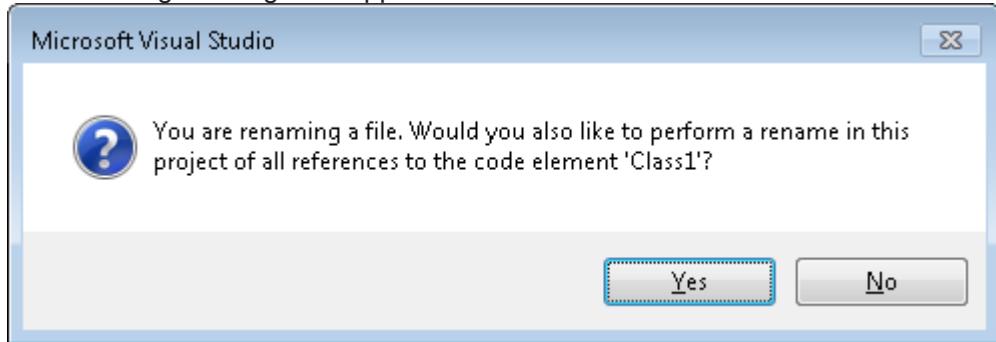
To change the class name, complete the following steps:

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. The following message will appear. Click Yes.



Figure

174: Change the class name

4. In the Solution Explorer, double-click HelloWorld.vb to open it for editing.

6.2.4 Add Code

Add the following code to create the add-in. 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.ReadOnly)> _

Public Class HelloWorld

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
```

6.2.5 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.

To create a manifest file

1. 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">

  <Name>HelloWorld</Name>

  <FullClassName>HelloWorld.HelloWorld</FullClassName>

  <Text>HelloWorld</Text>

  <Description>Show Hello World.</Description>

  <VisibilityMode>AlwaysVisible</VisibilityMode>

  <Assembly>C:\Samples\HelloWorld\HelloWorld\bin\Debug\HelloWorld.dll</Assembly>

  <AddInId>239BD853-36E4-461f-9171-C5ACEDA4E723</AddInId>

  <VendorId>ADSK</VendorId>

  <VendorDescription>Autodesk, Inc, www.autodesk.com</VendorDescription>

</AddIn>

</RevitAddIns>
```

Note: The FullClassName includes the Root namespace found on the Application tab of the properties for the project.

3. Save the file as HelloWorld.addin and put it in the following location:
 - C:\ProgramData\Autodesk\Revit\Addins{{RelYear}}\

Refer to [Add-In Integration](#) for more details using manifest files.

6.2.6 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.

6.2.7 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 20XX\Revit.exe.

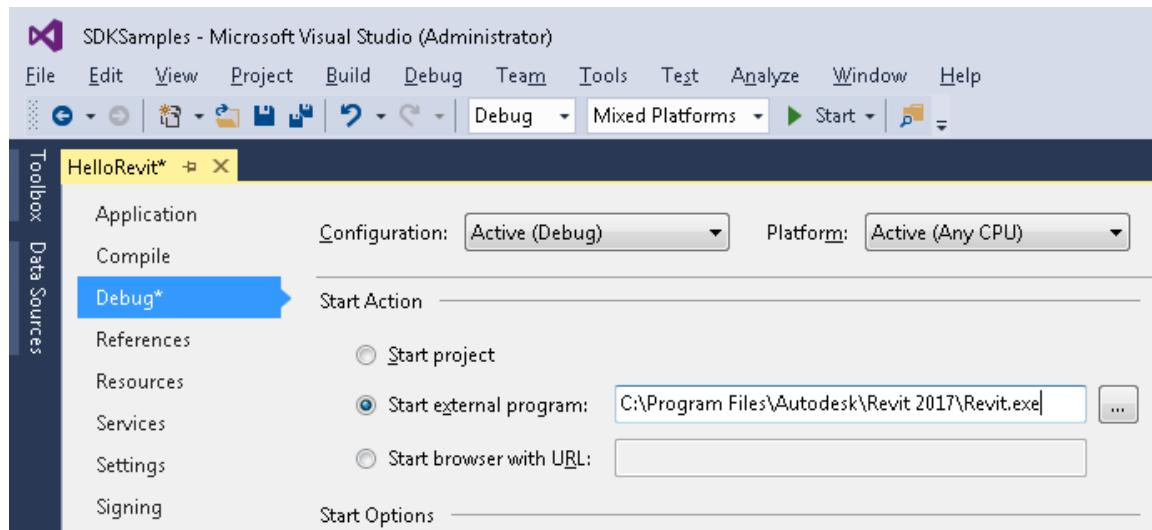


Figure 175: Set Debug environment

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.

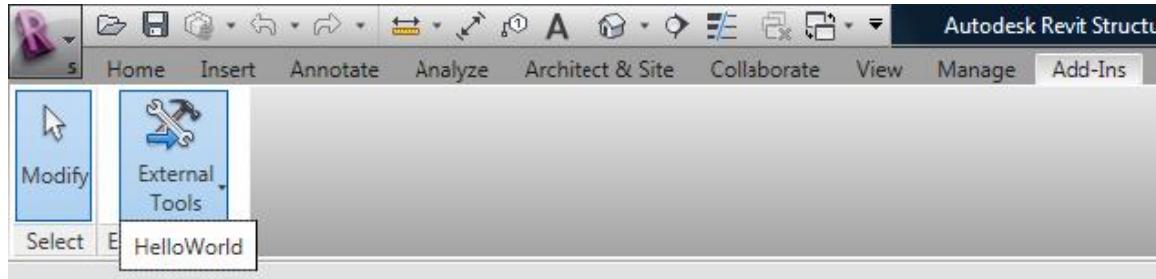


Figure 176: HelloWorld External Tools command*

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



Figure 177: TaskDialog message*

6.3 Material Properties Internal Units

Parameter Name in Dialog	API Parameter Name	Variable Type	Internal Database Units	Description
<i>Steel / Generic</i>				
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_TemperatureExp
Thermal Expansion coefficient Y	ThermalExpansionCoefficientY	double	UT_TemperatureExp
Thermal Expansion coefficient Z	ThermalExpansionCoefficientZ	double	UT_TemperatureExp
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_TemperatureExp	

Thermal Expansion coefficient Y	ThermalExpansionCoefficientY	double	UT_TemperatureExp	
Thermal Expansion coefficient Z	ThermalExpansionCoefficientZ	double	UT_TemperatureExp	
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 = $\text{ConcreteCompression} / \text{ShearStrengthReduction}$

Wood

#	Unit	Dimension	Internal Representation
Young Modulus	PoissonModulus	double	UT_Stress
Poisson ratio	ShearModulus	double	UT_Number
Shear modulus	ShearModulus	double	UT_Stress
Thermal Expansion coefficient	ThermalExpansionCoefficient	double	UT_TemperatureExp
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

1\.	UT_Number	No dimension	Simple number.
2\.	UT_Stress	(Mass) x (Length ₁) x (Time ₋₂)	Kg (mass) / (Foot*Second)
3\.	UT_TemperalExp	(Temperature ₋₁)	(1/° C)
4\.	UT_UnitWeight	(Mass) x (Length ₂) x (Time ₋₂)	Kg (mass) / (Foot*Second)

6.4 Concrete Section Definitions

6.4.1 Concrete-Rectangular Beam

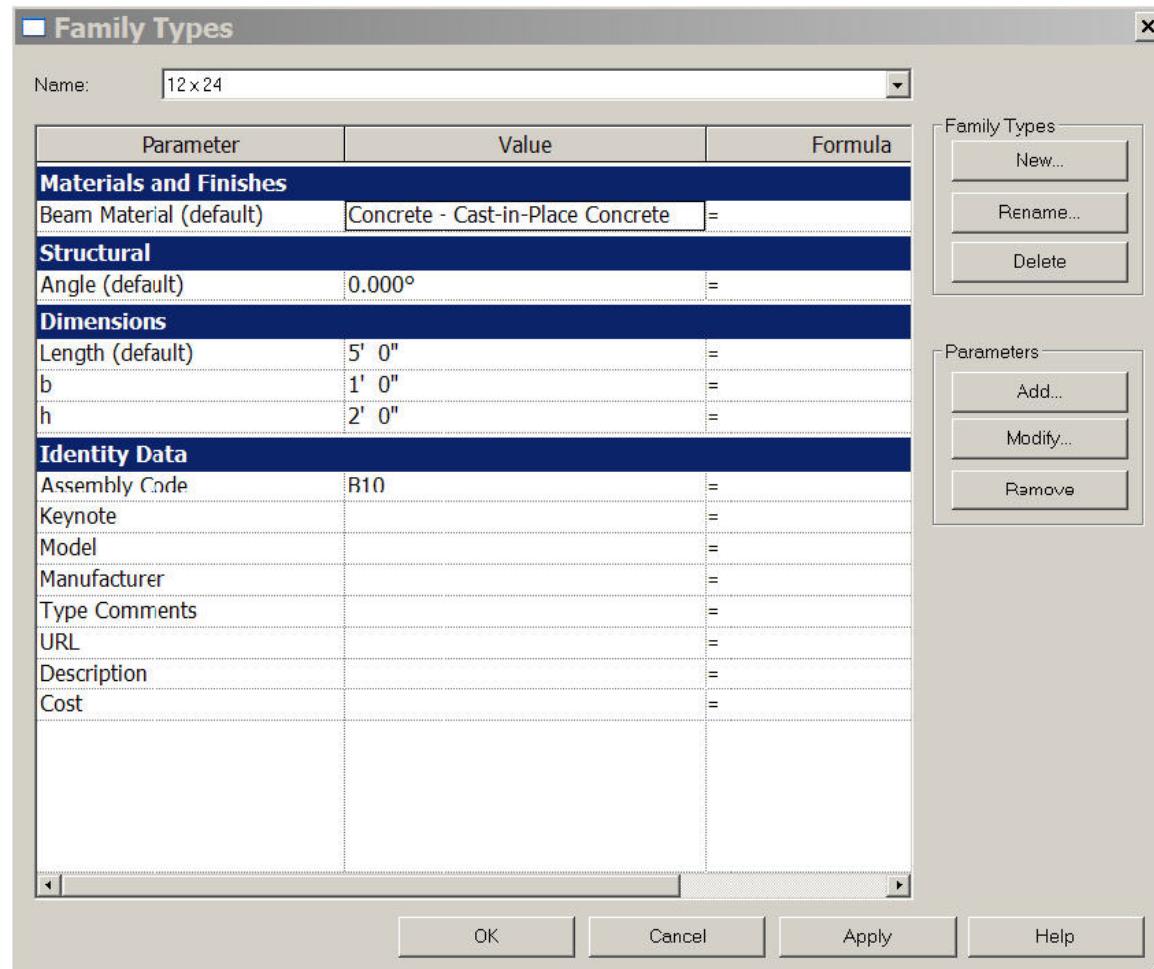


Figure 178: Concrete-Regangular Beam Parameters

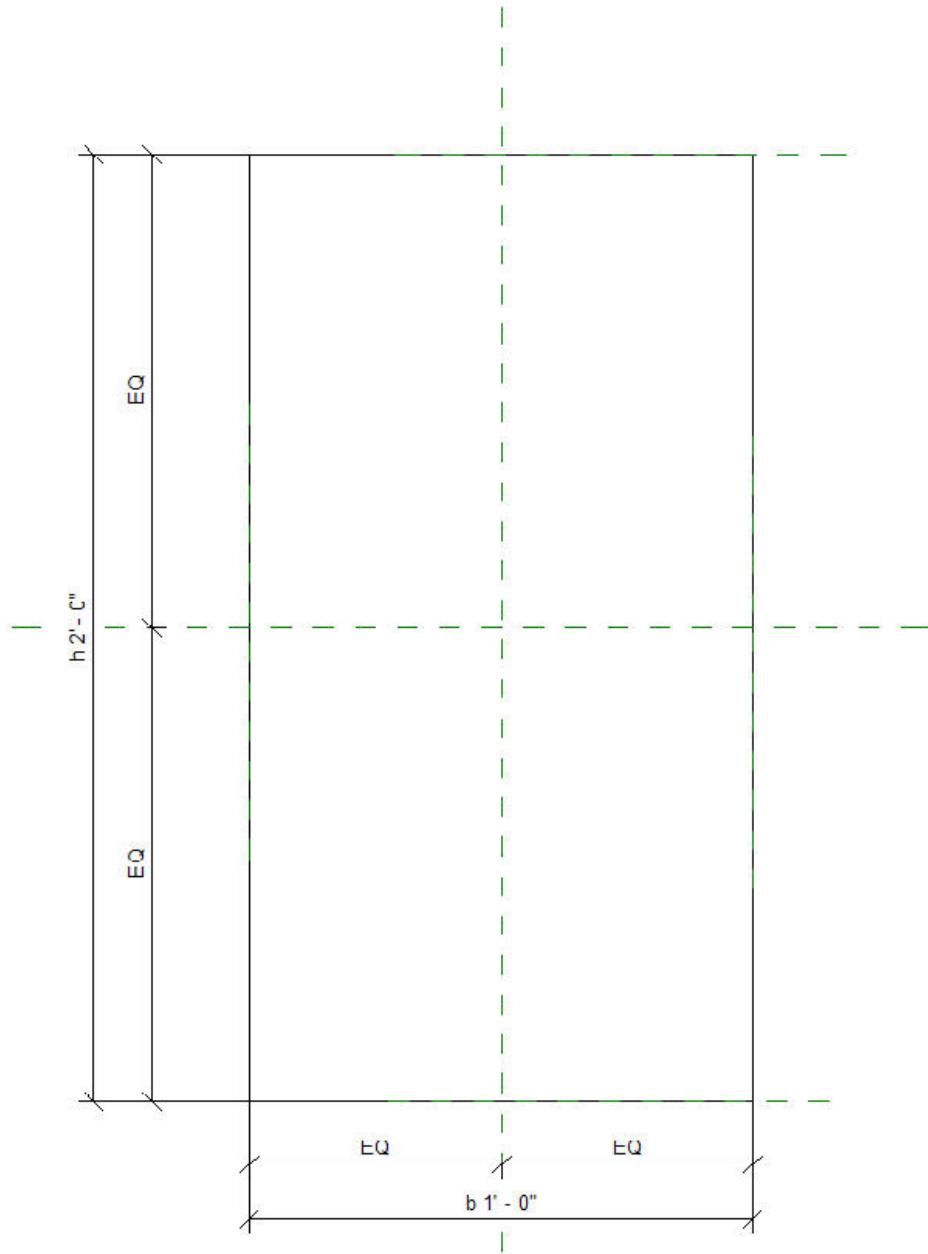


Figure 179: Concrete-Rectangular Beam Cross Section

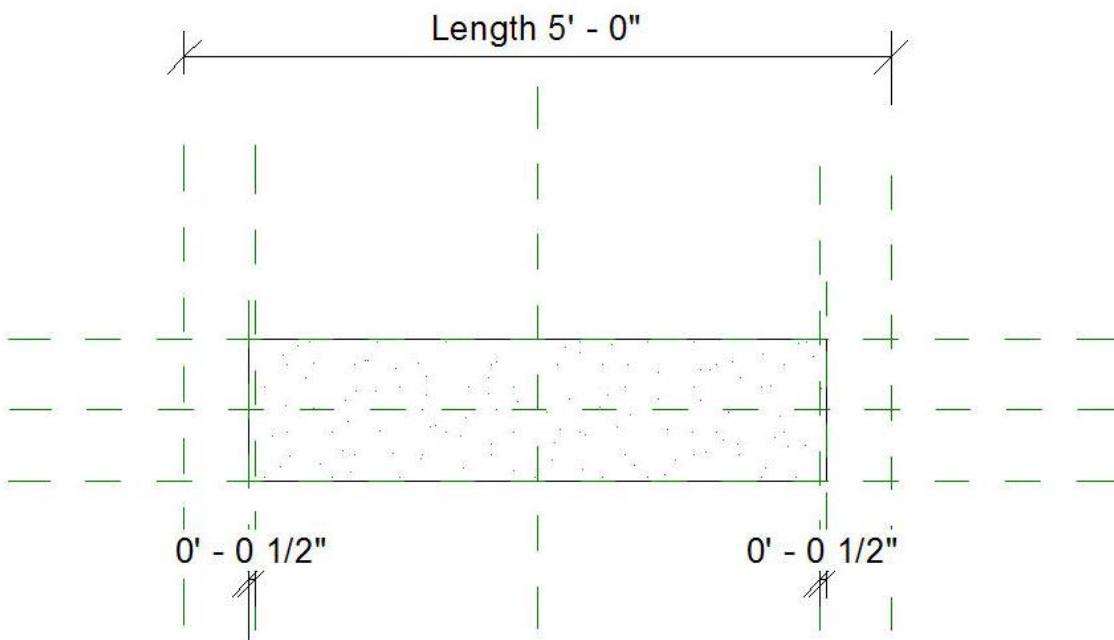


Figure 180: Concrete-Rectangular Beam

6.4.2 Precast-Rectangular Beam

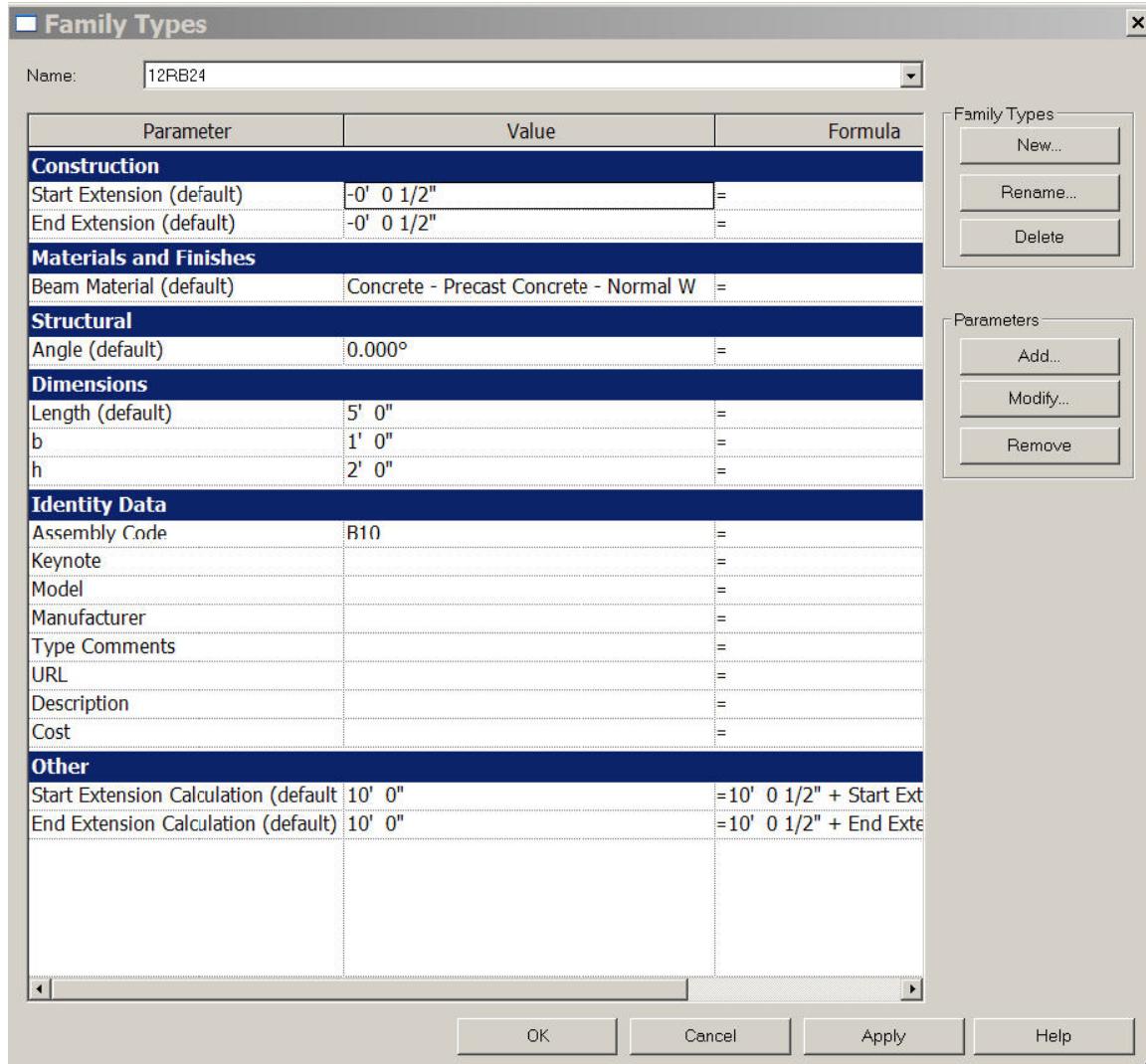


Figure 181: Precast-Rectangular Beam Properties

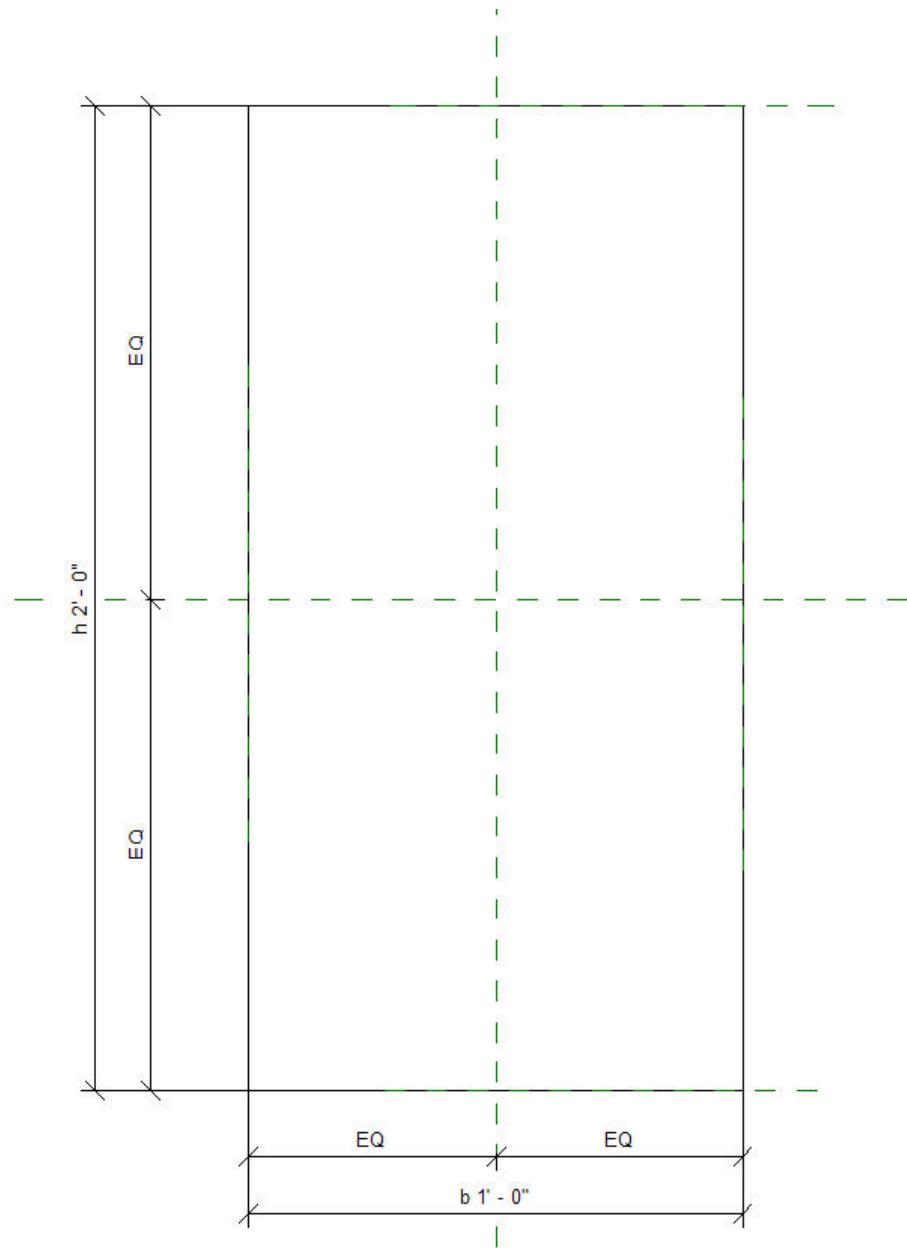


Figure 182: Precast-Rectangular Beam Cross Section

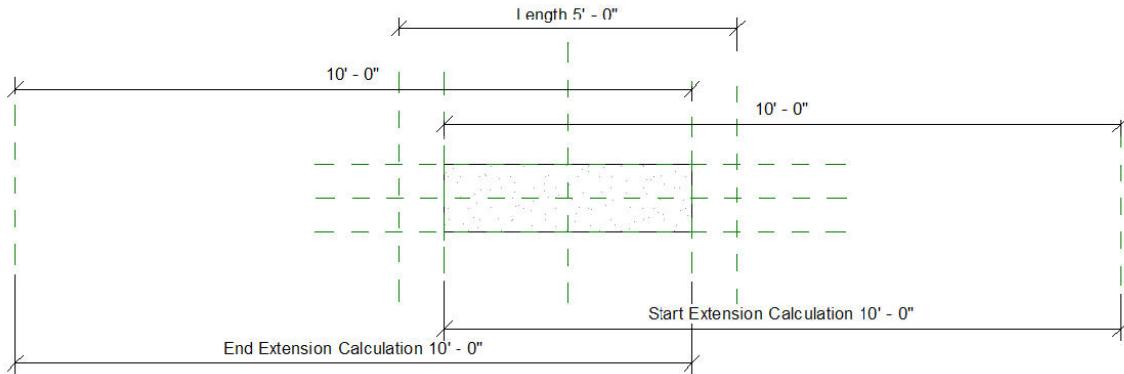


Figure 183: Precast-Rectangular Beam

6.4.3 Precast-L Shaped Beam

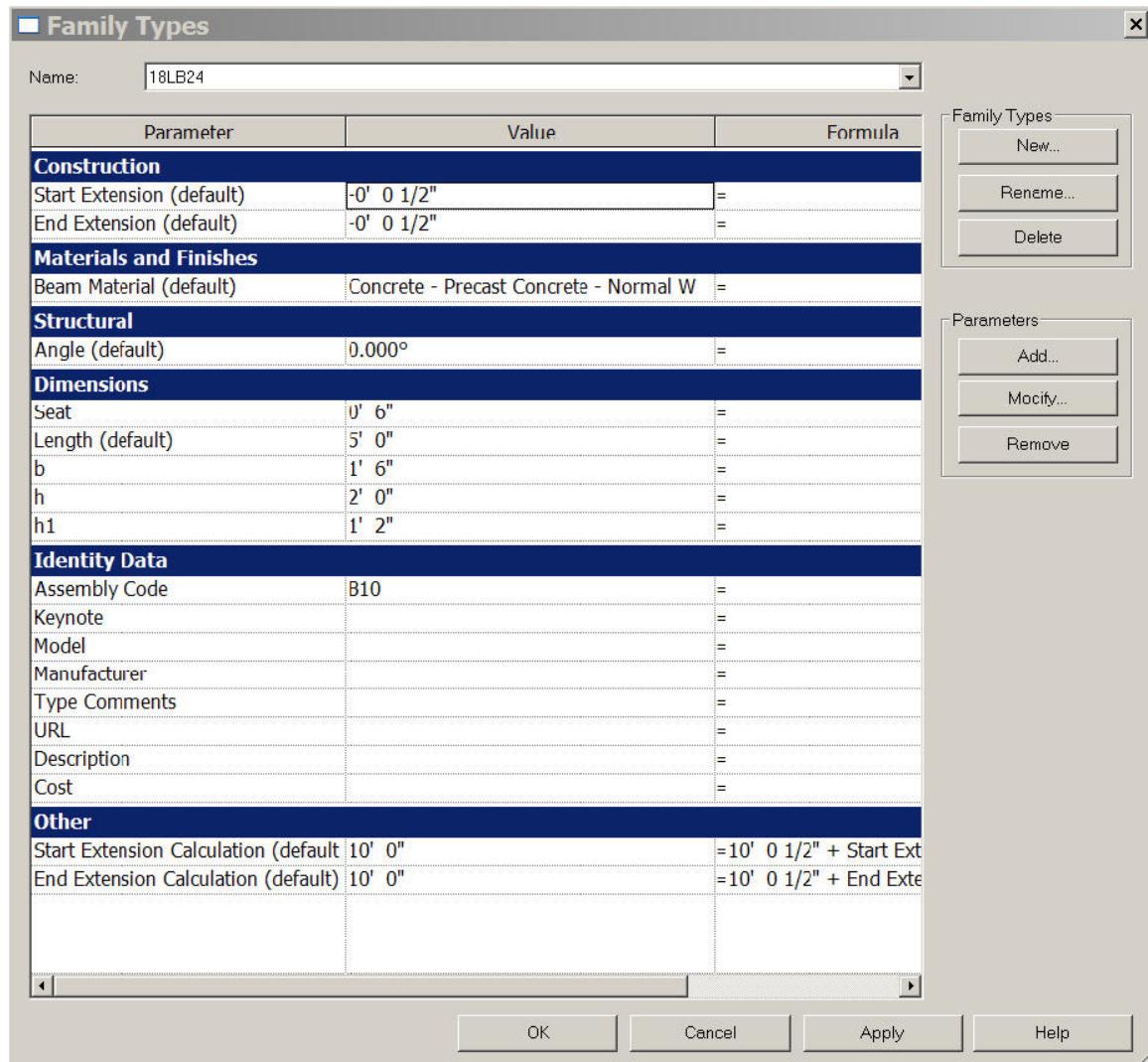


Figure 184: Precast-L Shaped Beam Properties

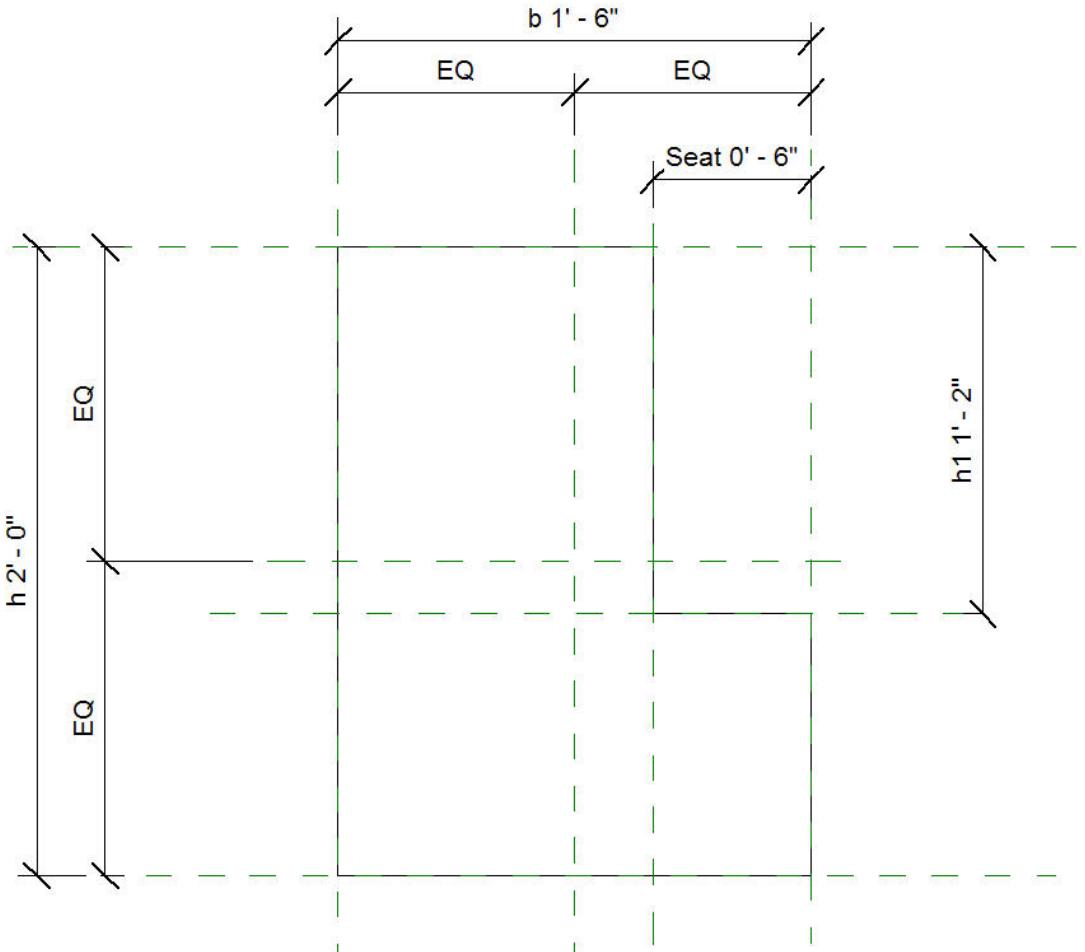


Figure 185: Precast-L Shaped Beam Cross Section

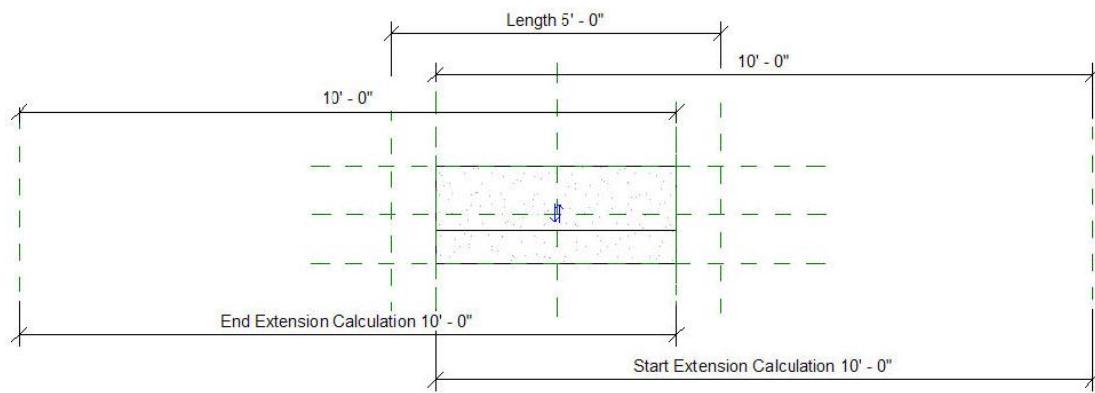


Figure 186: Precast-L Shaped Beam

6.4.4 Precast-Single Tee

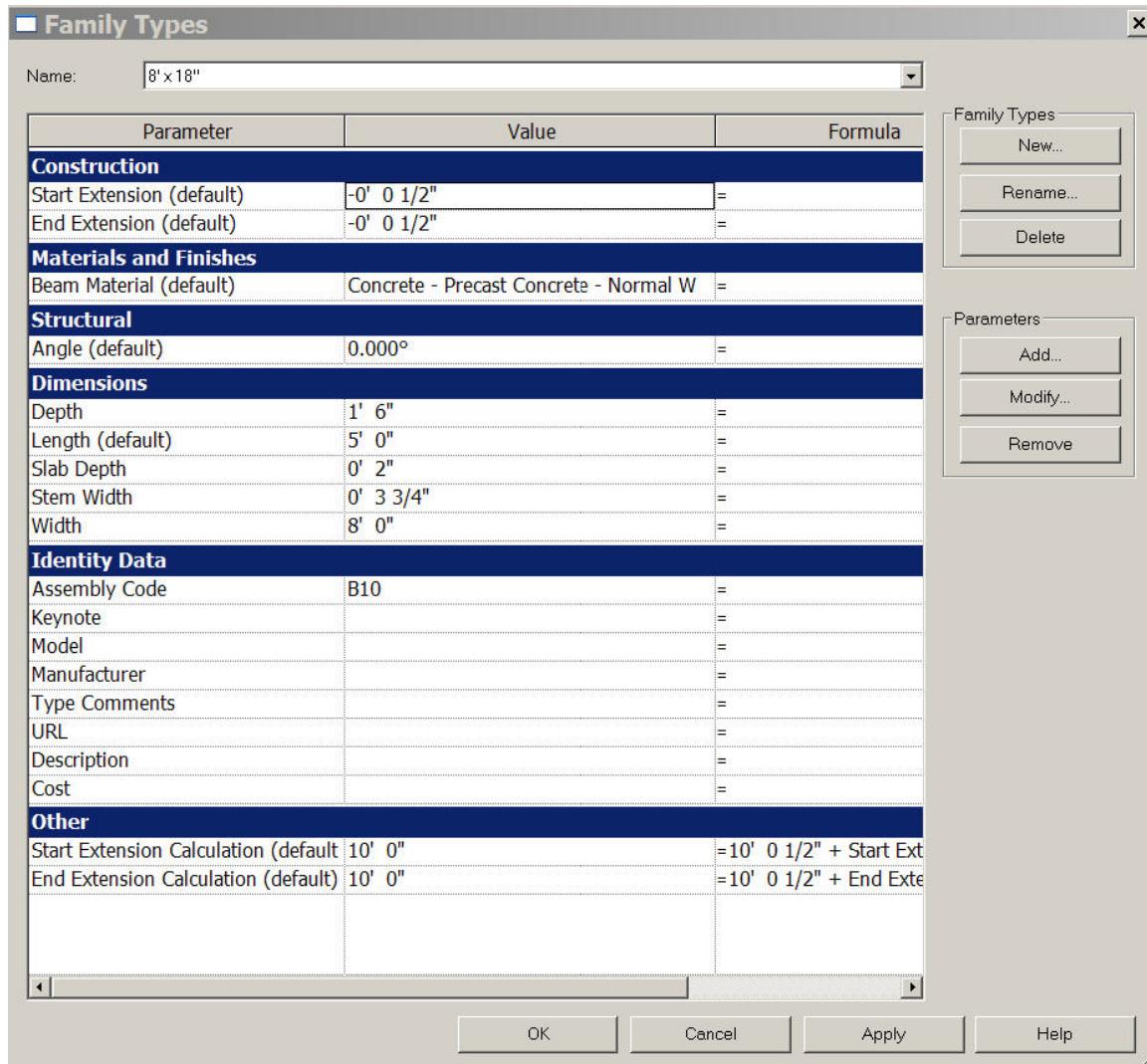


Figure 187: Precast-Single Tee Properties

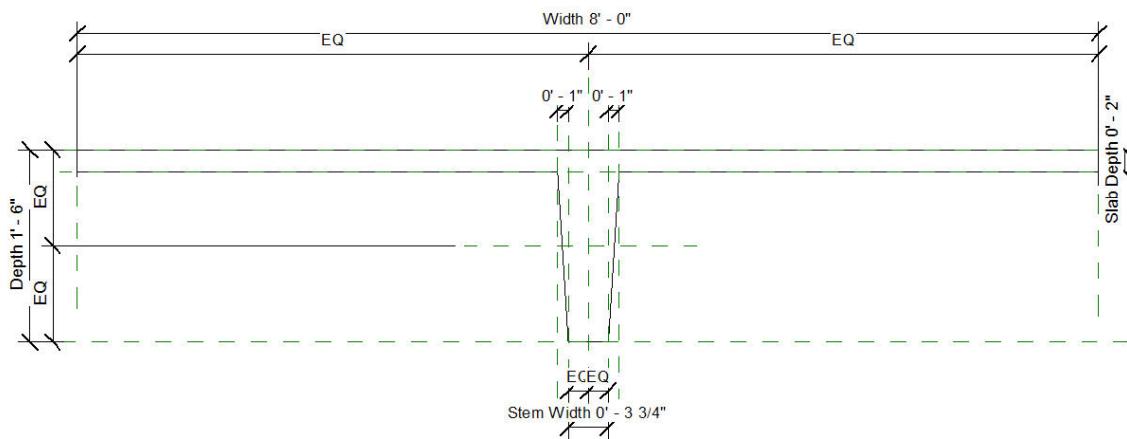


Figure 188: Precast-Single Tee Cross Section

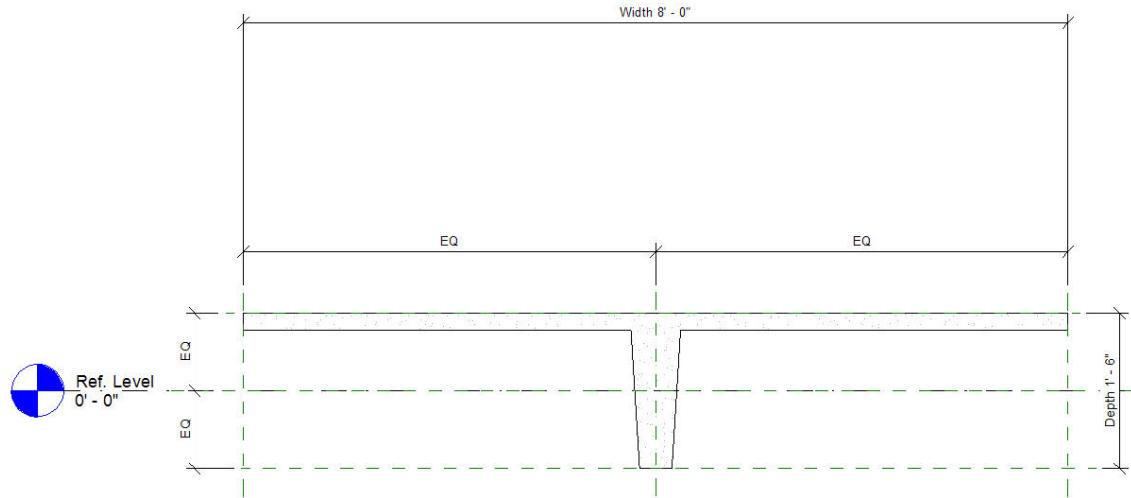


Figure 189: Precast-Single Tee Cross Section 2

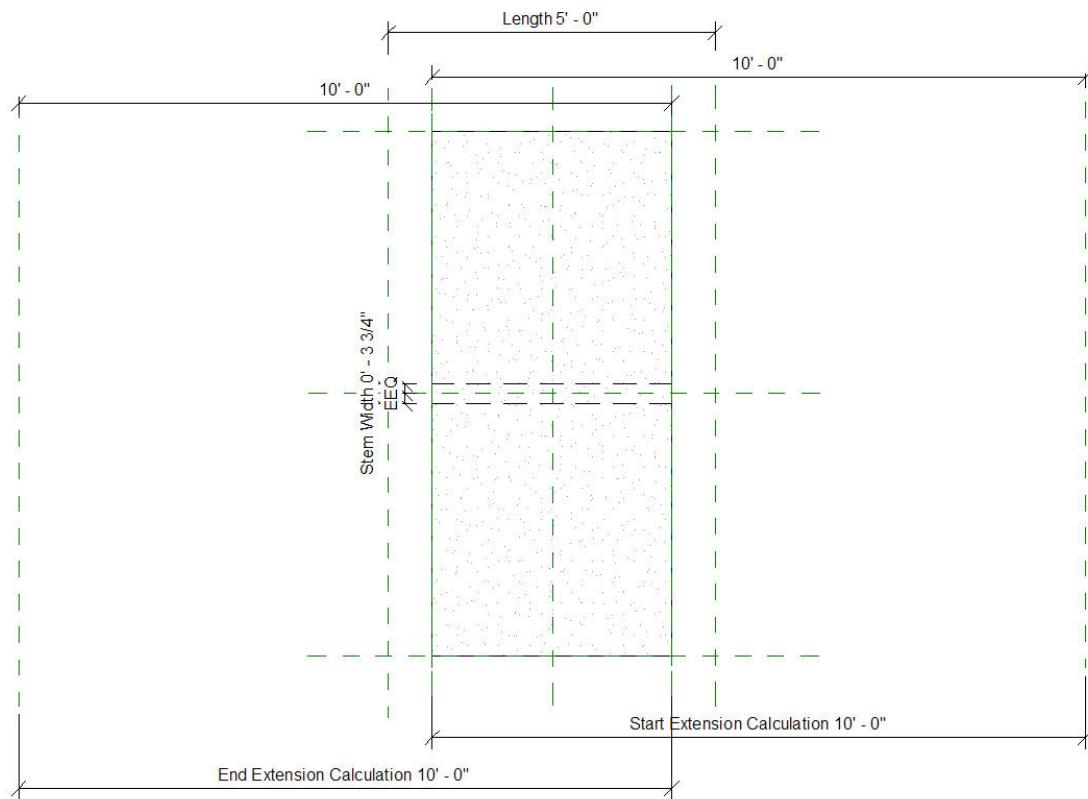


Figure 190: Precast-Single Tee

6.4.5 Precast-Inverted Tee

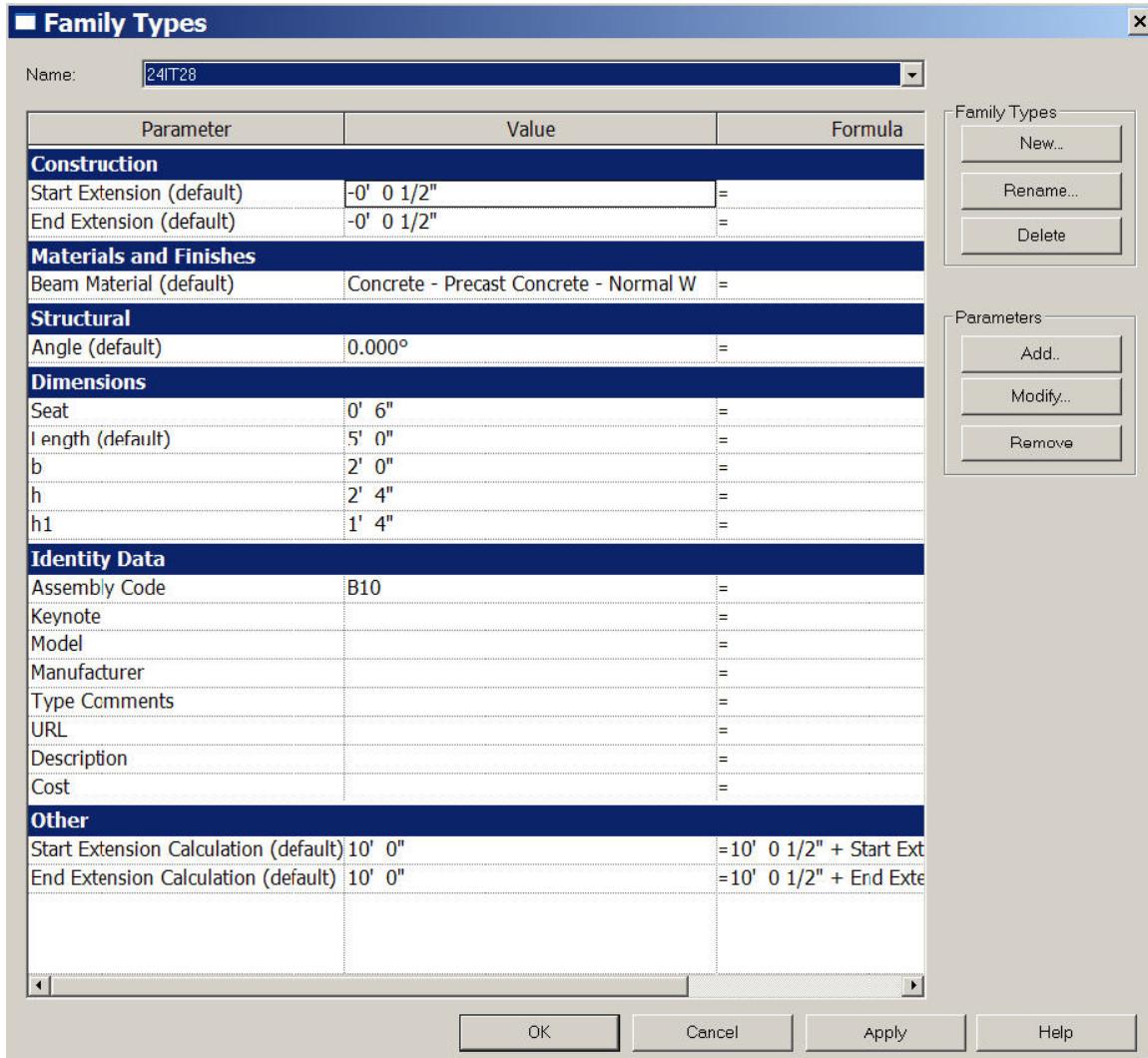


Figure 191: Precast-Inverted Tee Properties

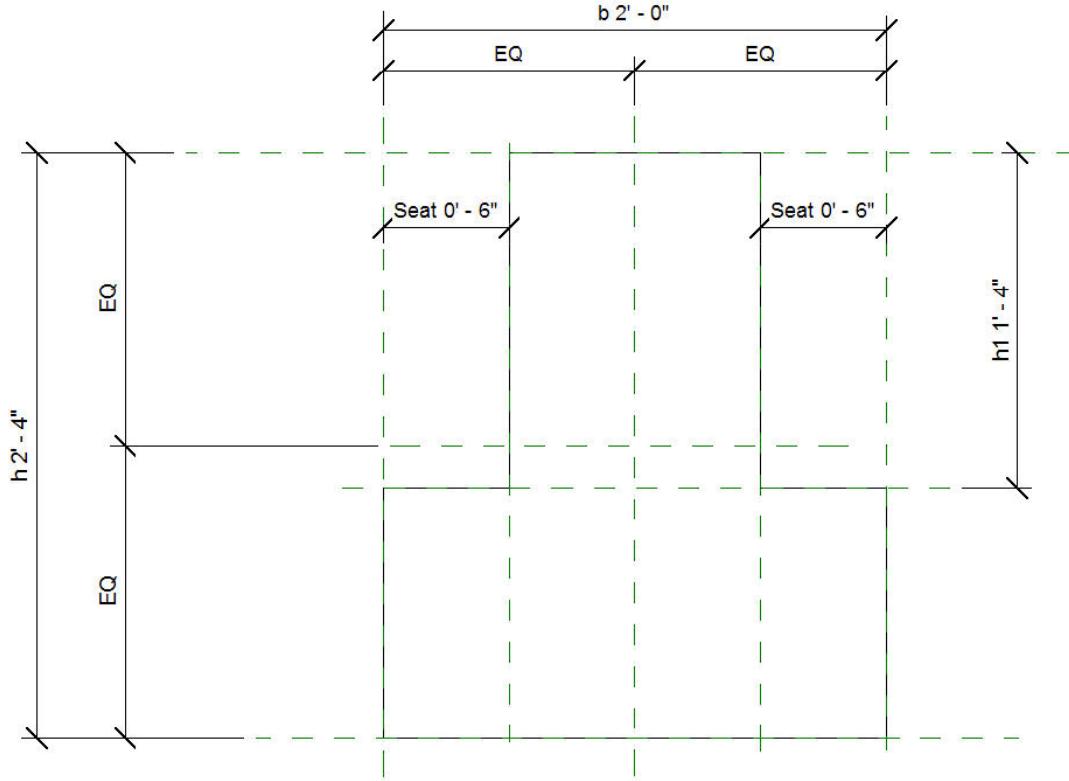


Figure 192: : Precast-Inverted Tee Cross Section

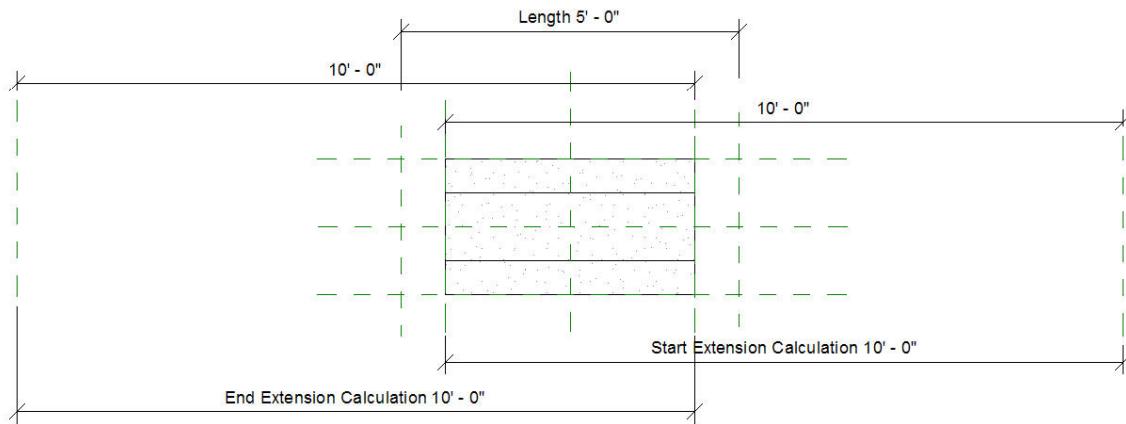


Figure 193: Precast-Inverted Tee

6.4.6 Precast-Double Tee

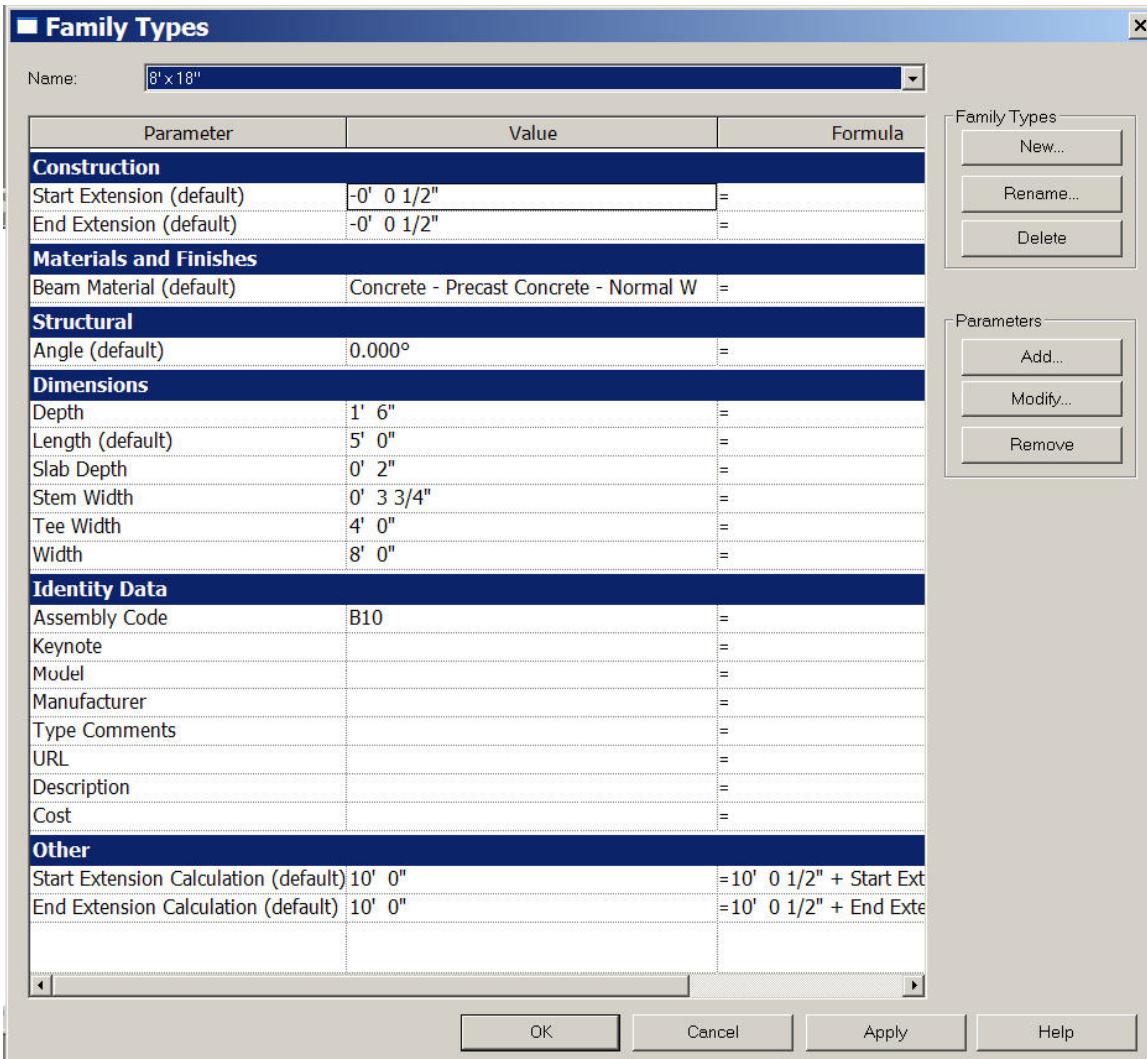


Figure 194: Precast-Double Tee

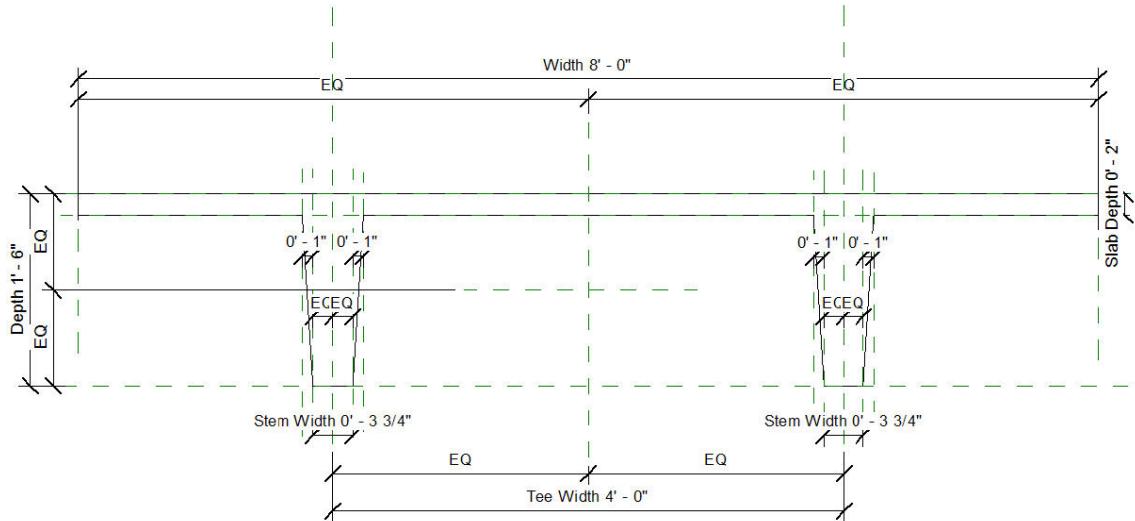


Figure 195: Precast-Double Tee Cross Section

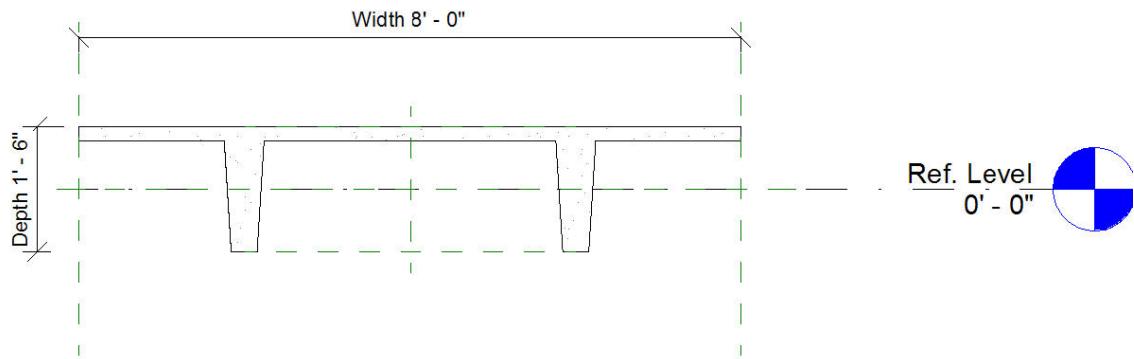


Figure 196: Precast-Double Tee Cross Section 2

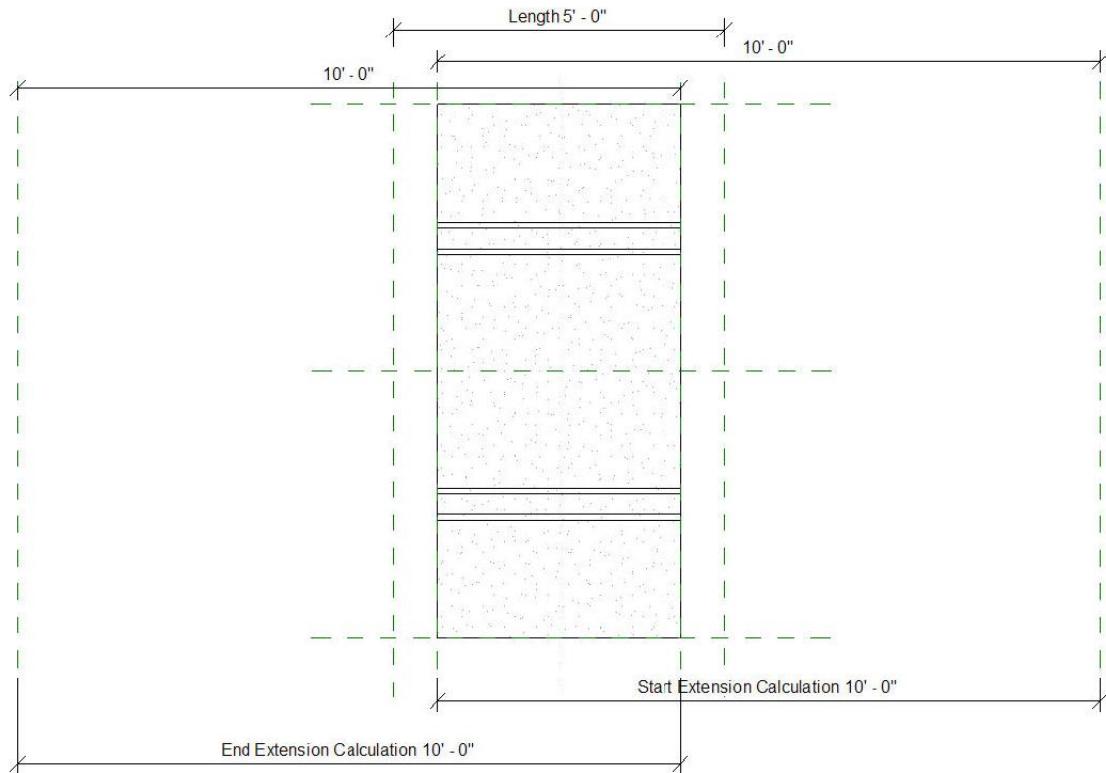


Figure 197: Precast-Double Tee

6.5 API User Interface Guidelines

6.5.1 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.

6.5.2 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.

6.5.3 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.

6.5.4 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

6.5.5 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.

6.5.6 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 Filter, 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.

6.5.7 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.

```
<table cellpadding="4" cellspacing="0" summary="" class="table" frame="border" border="1" rules="all">
```

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.

6.5.8 Dialog Guidelines

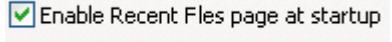
A dialog is a separate controllable area of the application that contains information and/or controls for editing information. Dialogs are the primary vehicle for obtaining input from and presenting information to the user. Use the following guidelines when deciding which dialog to use.

Dialog Type	Definition	Use When
Modal	Halts the rest of the application and waits for user input	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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 below for more on this topic

Dialog Controls

Control	Use When	Example
<u>CHECK BOX</u>	The choice being made (and its 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

New pages should be opened in:

- a new window
- a new tab

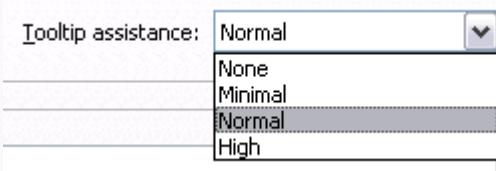
TEXT
BOX

This choice requires manually entering a numerical text value

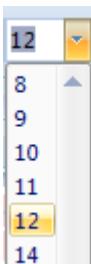
DROP
DOWN
LIST

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

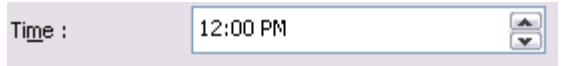
Use when the expected input or existing data will be in a specific range.

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



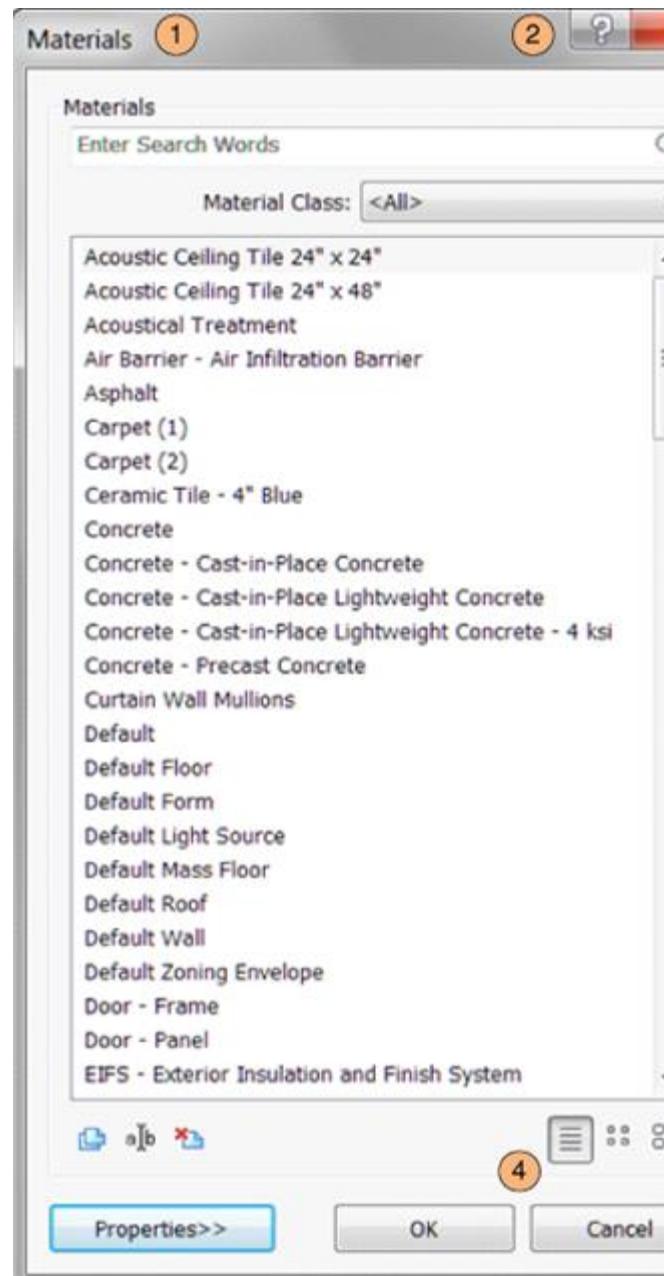
Laying Out a Dialog

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

documentatio
n

- Note that many legacy dialogs in Revit still have the Help button located in the bottom right of the dialog

3 *Contro
ls* 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	<i>Comm it Button s</i>	<p>See the Committing Changes section.</p> <p>The most frequently-used Commit buttons include:</p> <ul style="list-style-type: none"> • 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
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

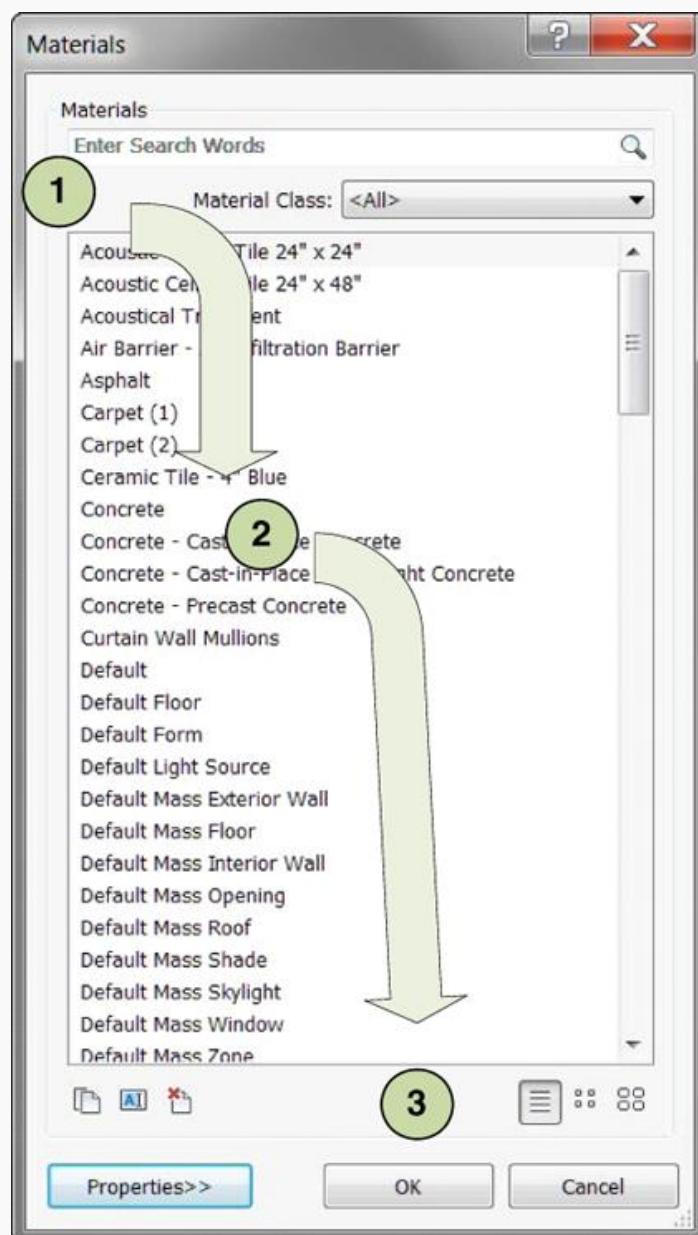
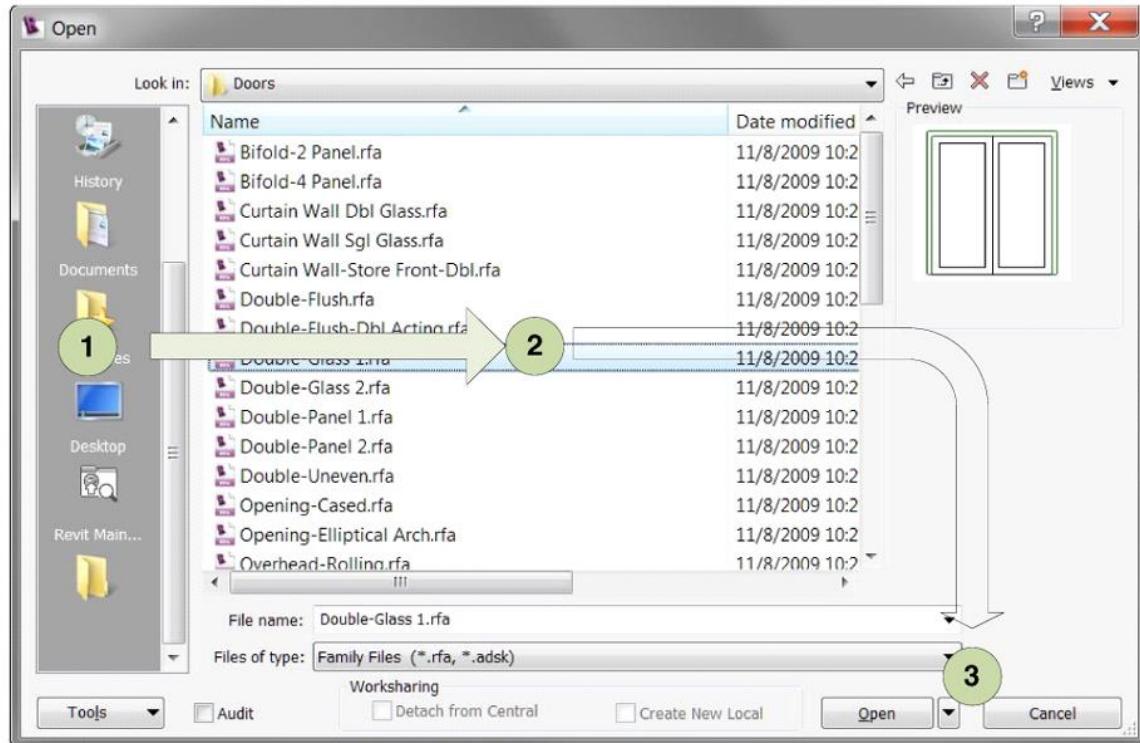


Figure 198 - Materials dialog

**Figure 199 - Revit File Open dialog**

Note: A top-down flow can also be used within this dialog, if the 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.

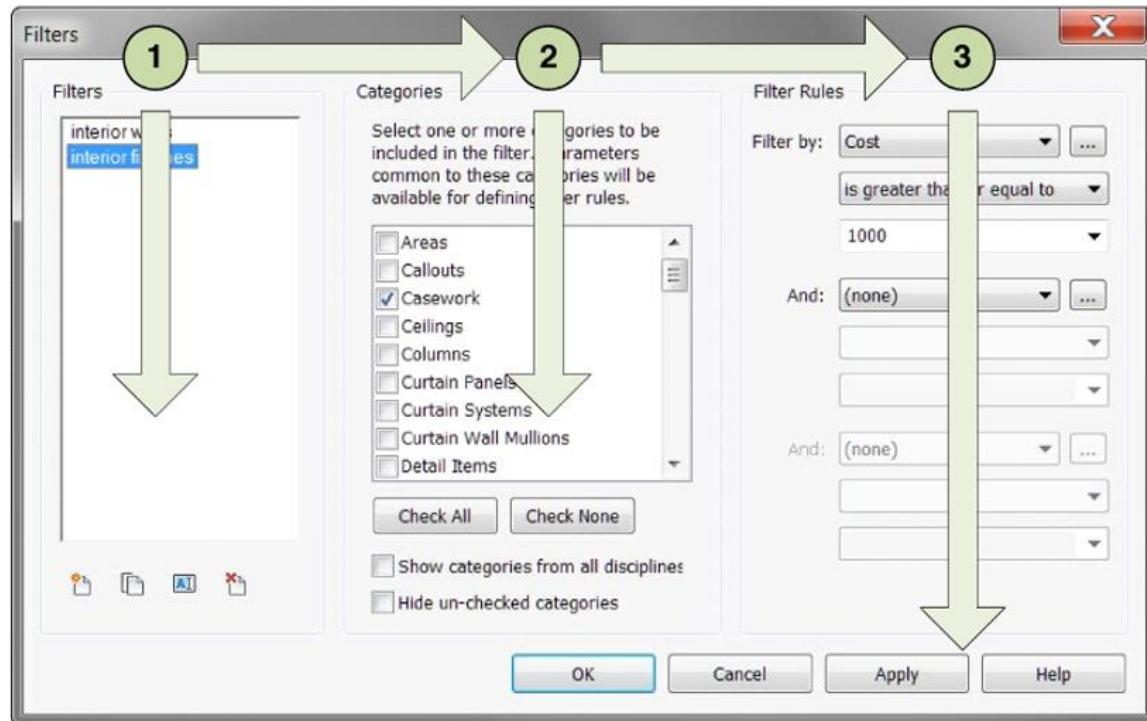


Figure 200 - Revit Filters dialog

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.

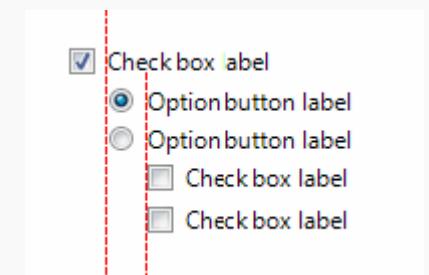
Spacing and Margins

The following table 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.

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 201 - 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.

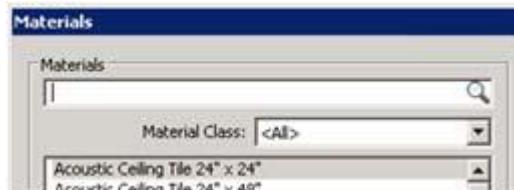


Figure 202 - Group box with no label and group box with one control and Group box with title that is redundant with dialog title

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:

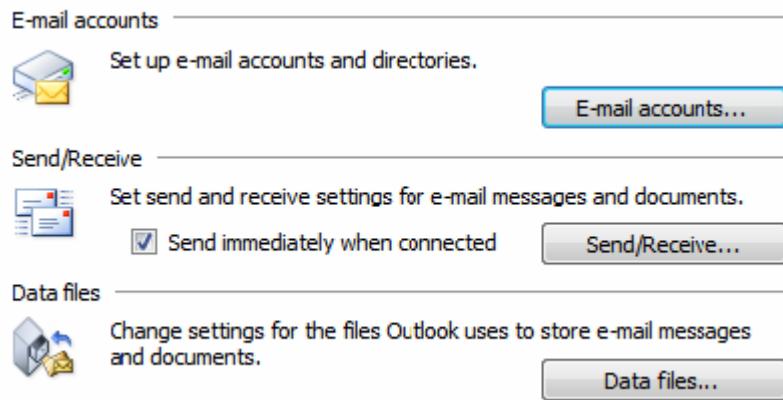


Figure 203 - Horizontal separators in Microsoft Outlook 2007

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

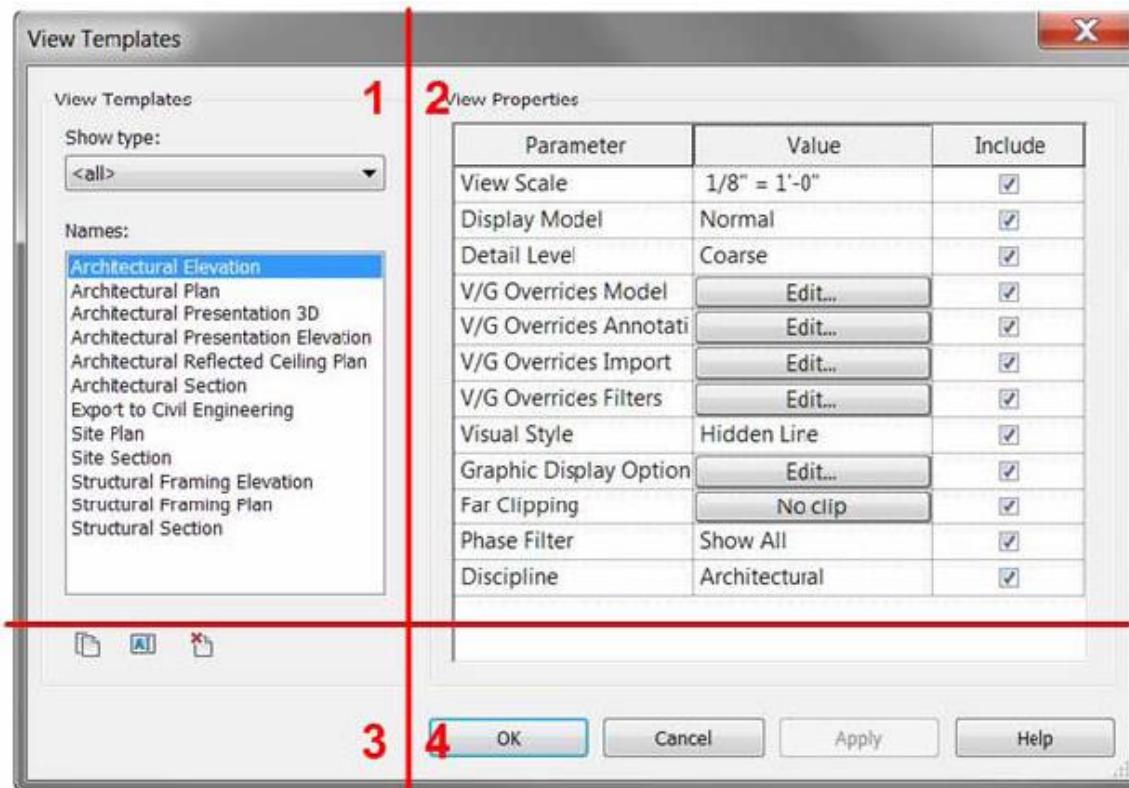


Figure 204 - Four square grid applied to Revit View Templates dialog to demonstrate how it should be resized

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
<i>Button</i>	Static	No	Yes
<i>Link</i>	Static	No	Yes
<i>Radio Button</i>	Static	No	Yes
<i>Spin Control</i>	Static	No	Yes, based on the element to which it is attached
<i>Slider</i>	Static	X Direction	Yes
<i>Scroll Bar</i>	Static	X Direction	Yes
<i>Tab</i>	Dynamic	X and Y Direction	Yes, but not smaller then the biggest control contained
<i>Progressive Disclosure</i>	Dynamic	X and Y Direction	Yes, but not smaller then the biggest control contained
<i>Check Box</i>	Static	No	Yes
<i>Drop-Down List</i>	Dynamic	X Direction	Yes but not smaller then the biggest text contained.
<i>Combo Box</i>	Dynamic	X and Y Direction	Yes, but not smaller then the biggest text contained
<i>List View</i>	Dynamic	X and Y Direction	Yes, but not smaller then the biggest text contained
<i>Text Box</i>	Dynamic	X and Y if multi-line	Yes
<i>Date Time Box</i>	Dynamic	X Direction	Yes, but not smaller then the biggest text contained

<i>Tree View</i>	Dynamic	X and Y Direction	Yes, but not smaller than the biggest text contained
<i>Canvas</i>	Dynamic	X and Y Direction	Yes
<i>Group Box</i>	Dynamic	X and Y Direction	Yes, but not smaller than the biggest control contained
<i>Progress Bar</i>	Static	X	Yes
<i>Status Bar</i>	Dynamic	X	Yes
<i>Table or data grid</i>	Dynamic	X and Y	Yes, table columns should grow proportionally in the X direction

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.

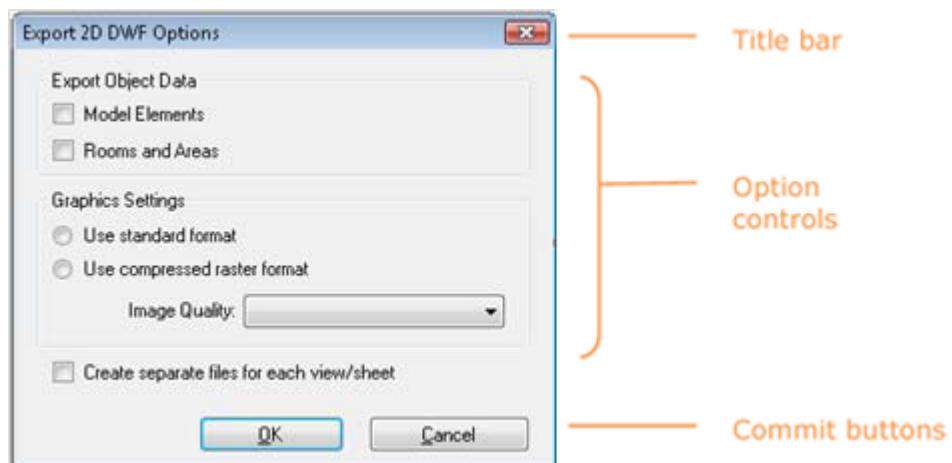


Figure 205 - The Export 2D DWF Options dialog

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.

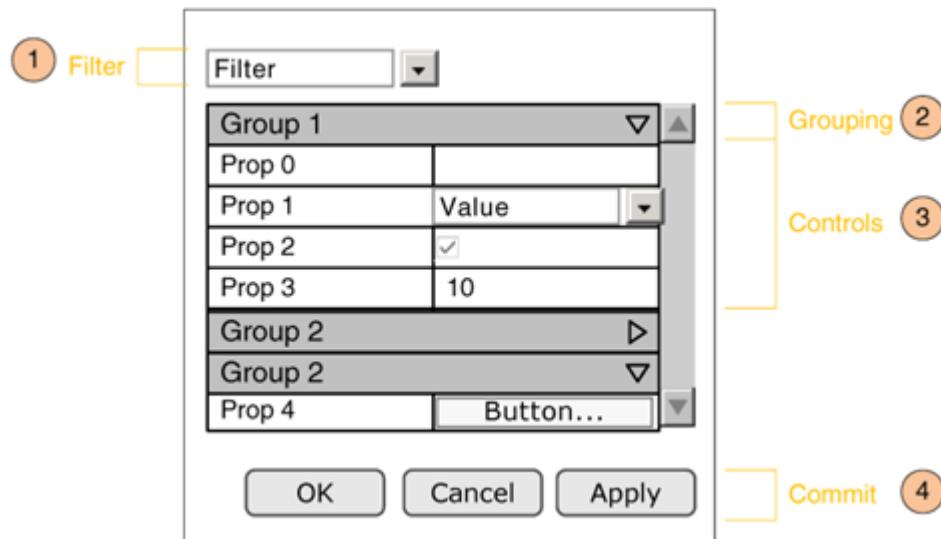


Figure 206 - Property Grid

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.

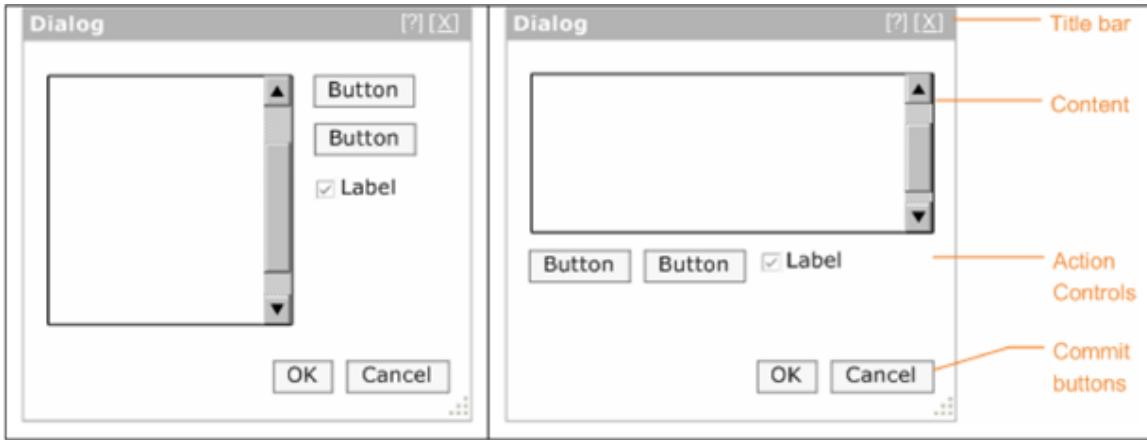


Figure 207 - Action controls to the right of content and action controls below content

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. Optionally provide the ability to search and/or filter the collection.

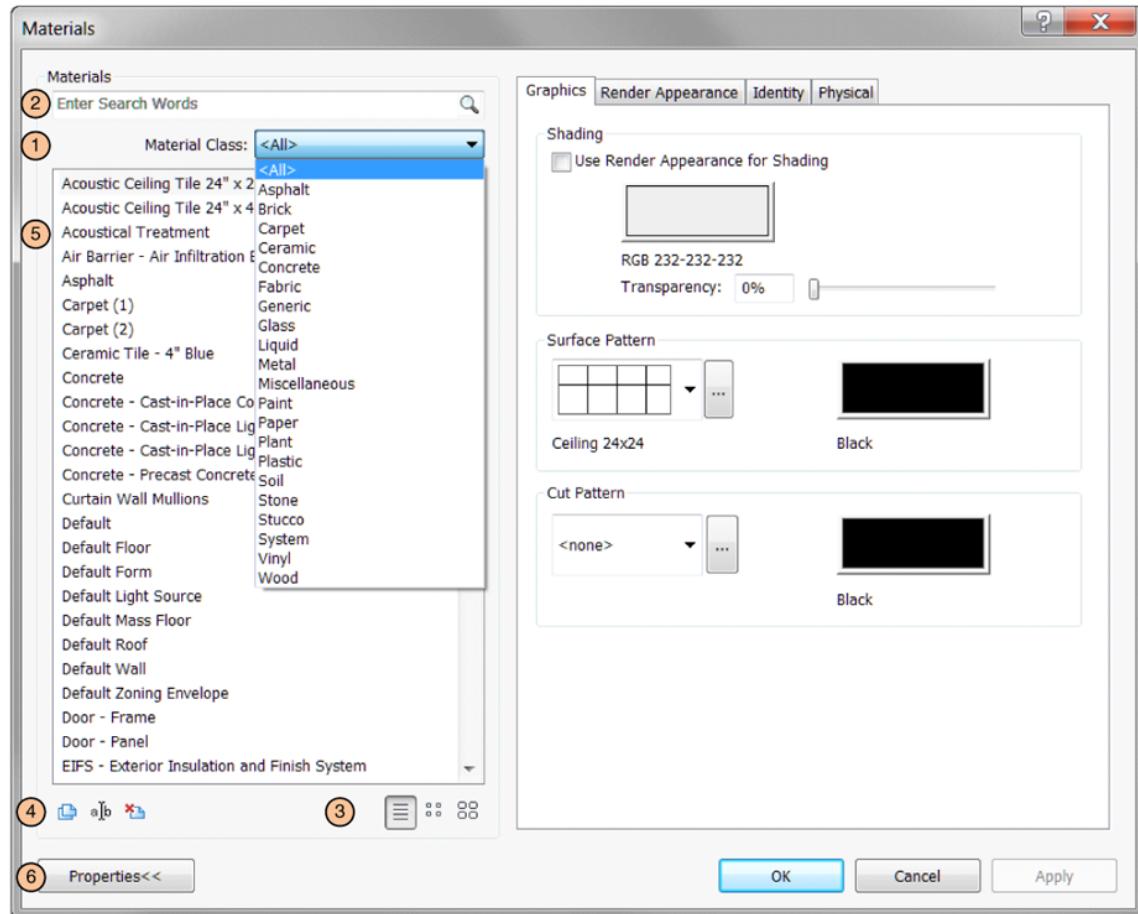


Figure 208 - Materials dialog from Revit

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	<p>This provides options for filtering the list view. This can be represented as a:</p> <ul style="list-style-type: none"> 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 list. Default set by designer Once a choice is selected, the collection will automatically update based on the 	No

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 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	<ul style="list-style-type: none"> A separate UI will be provided to edit, rename, delete or add items to the collection. 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.	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.

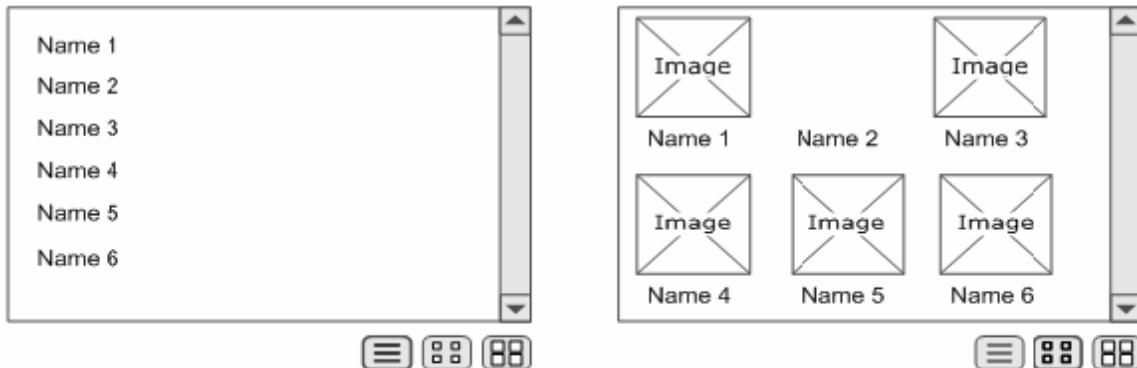


Figure 209 - List View, showing different ways of presenting the data

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 Windows User Experience Guidelines](#) should be followed when using a drop-down list.



Figure 210 - 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 Windows User Experience Guidelines](#) should be followed when using this option, including for how to decide between drop-down and combo box.

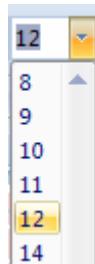


Figure 211 - The font selector in Microsoft Office 2007 is an example of a combo box

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 Windows User Experience Guidelines](#) should be followed when using a list box.

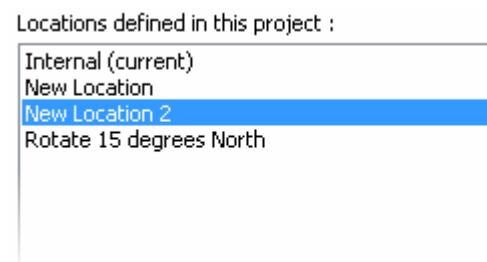


Figure 212 - List box

List View

Use when the data includes the option of list, details and/or graphical thumbnail previews, such as a Windows Open Dialog.

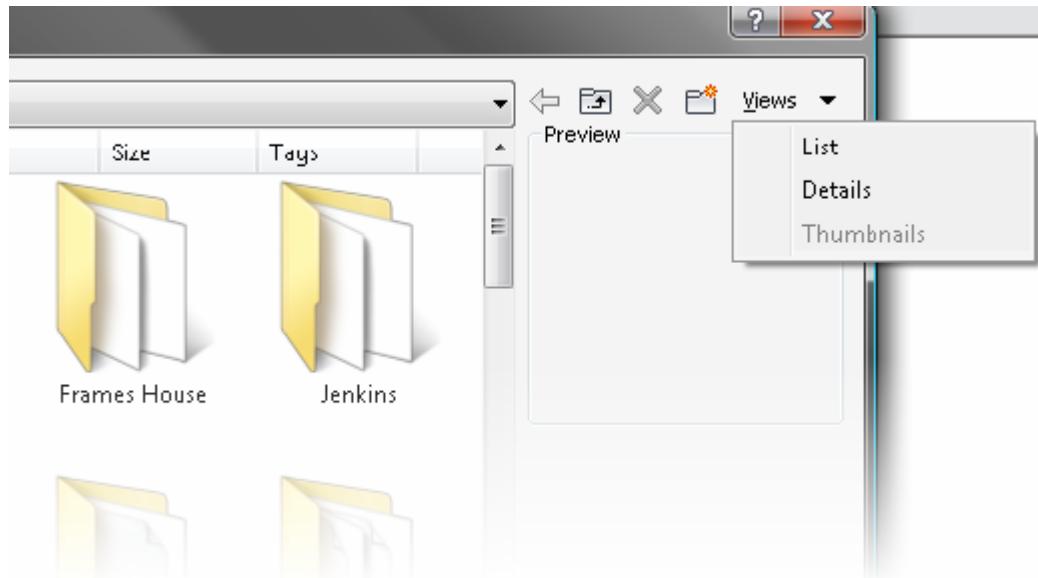


Figure 213 - List view

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.

Threshold (%)	Annual Design Conditions			
	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 214 - Good table example

Fi

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.

Threshold (%)	Annual Design Conditions			
	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 215 - 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 216 - 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 217 - 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.

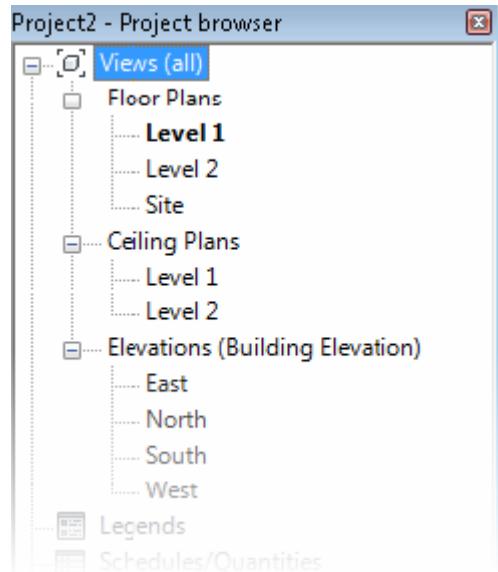


Figure 218 - The Revit Project Browser is a good example of a tree view

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.

Visibility	Projection/Surface	
	Lines	Patterns
<input checked="" type="checkbox"/> Areas		
<input checked="" type="checkbox"/> Casework		
<input checked="" type="checkbox"/> Hidden Lines		
<input checked="" type="checkbox"/> Ceilings		
<input checked="" type="checkbox"/> Common Edges		
<input checked="" type="checkbox"/> Hidden Lines	Override...	
<input checked="" type="checkbox"/> Columns		
<input checked="" type="checkbox"/> Curtain Panels		

Figure 219 - 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.

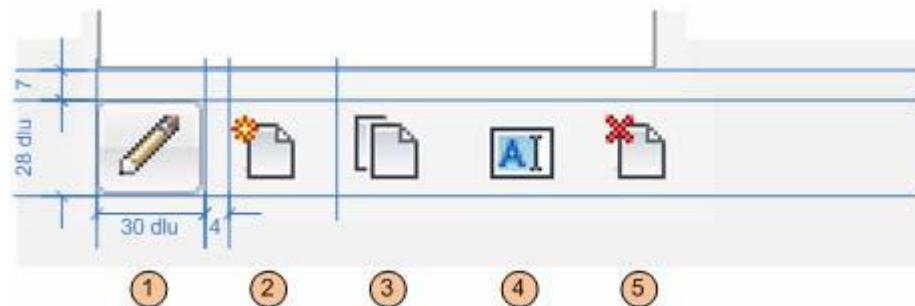


Figure 220 - The Edit Bar

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.

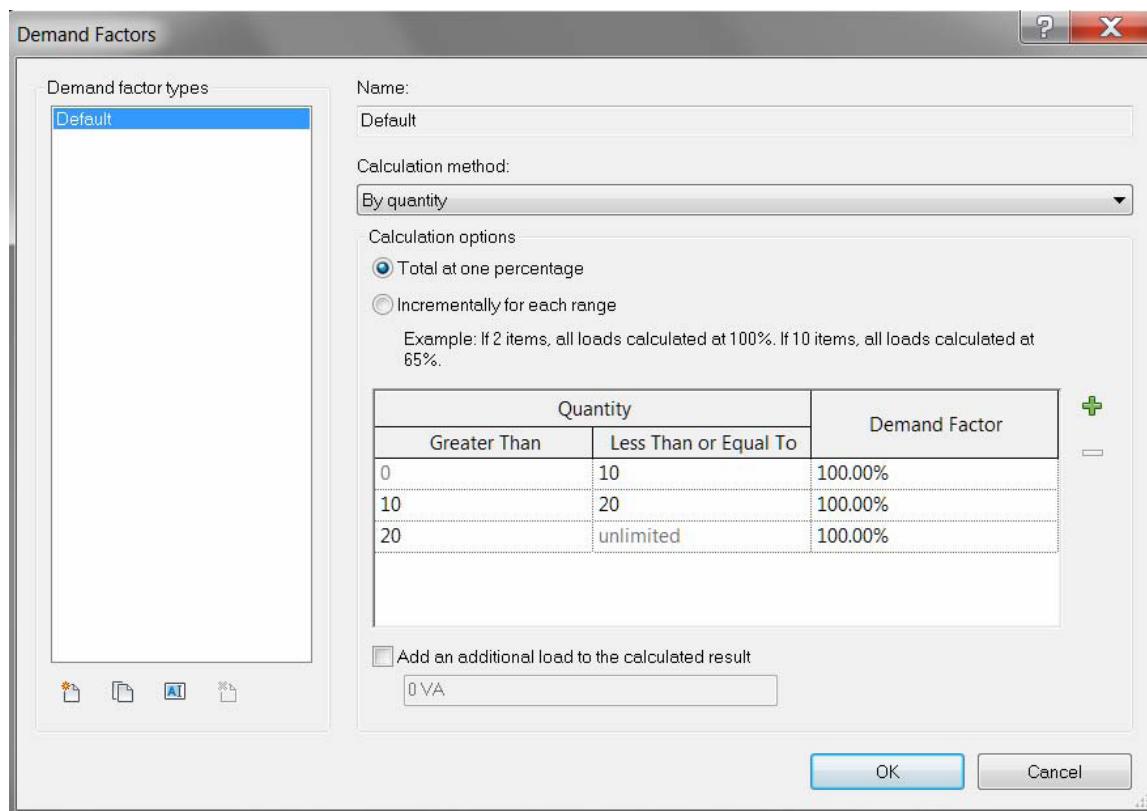


Figure 221 - Demand Factors dialog in Revit MEP 2011

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.

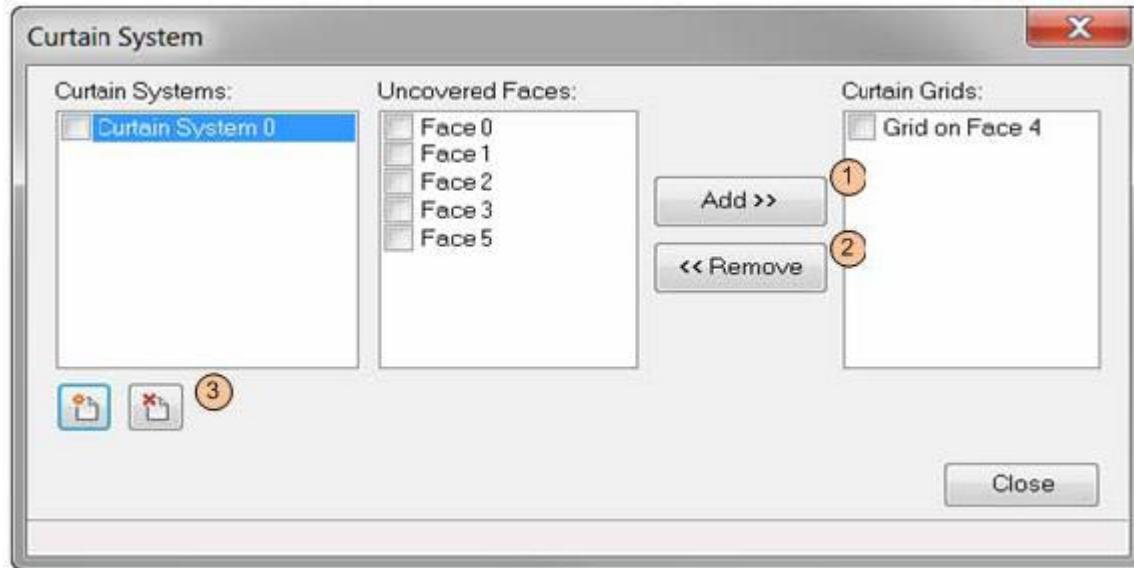


Figure 222 - The Curtain System SDK sample

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.

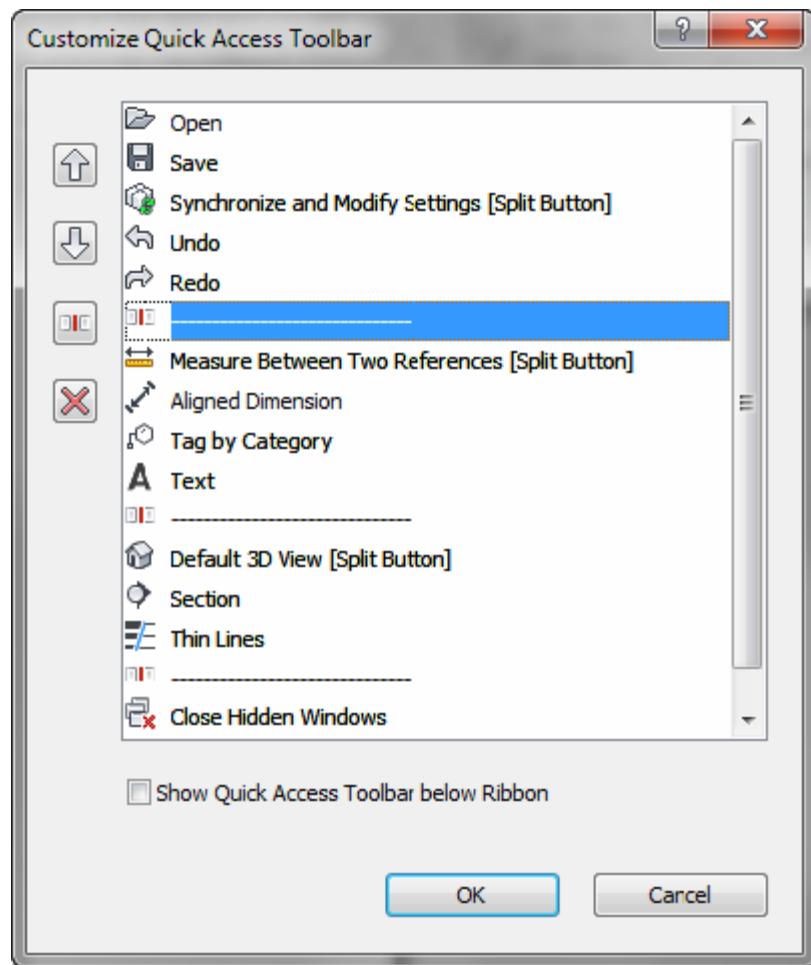


Figure 223 - Up/down buttons

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.

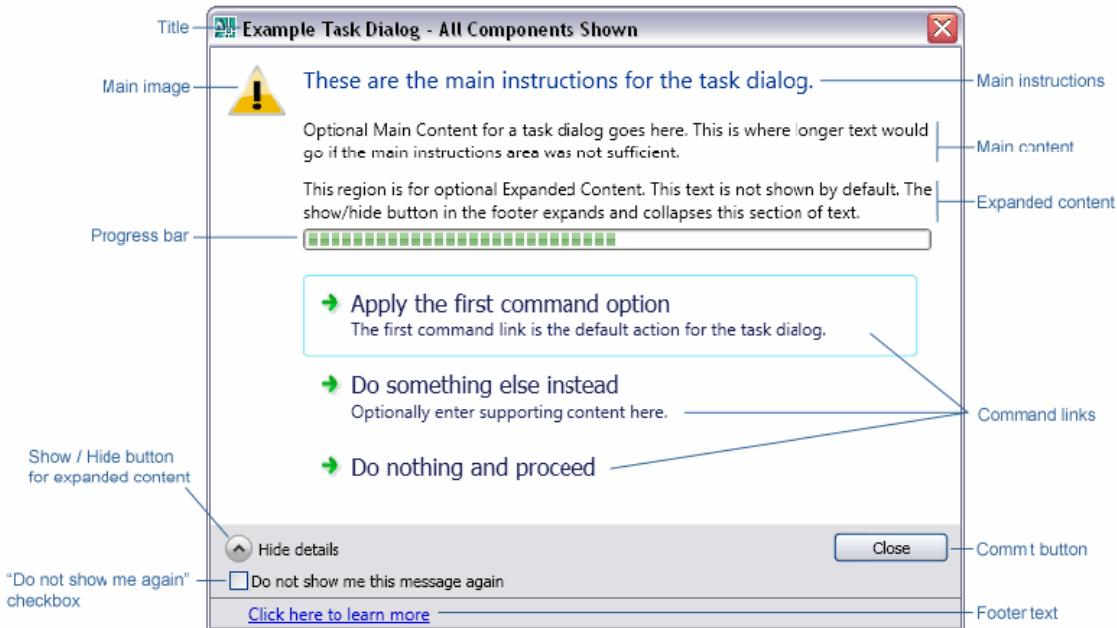


Figure 224 - A task dialog with all components visible

Note: 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, 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?"

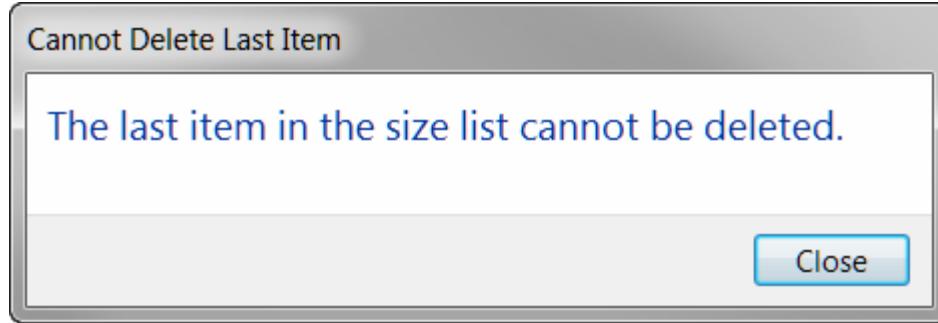


Figure 225 - A very simple task dialog with only main instructions for text

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

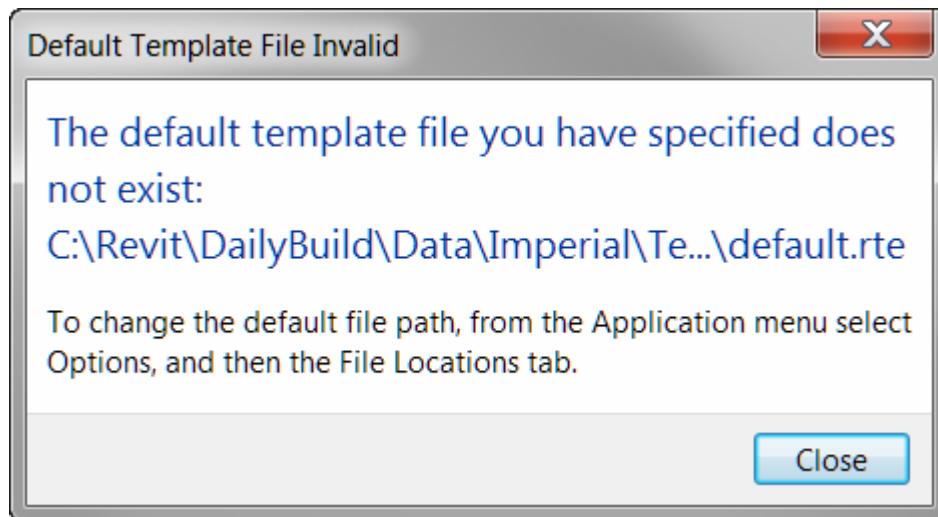


Figure 226 - A task dialog that uses both main instructions and main content

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

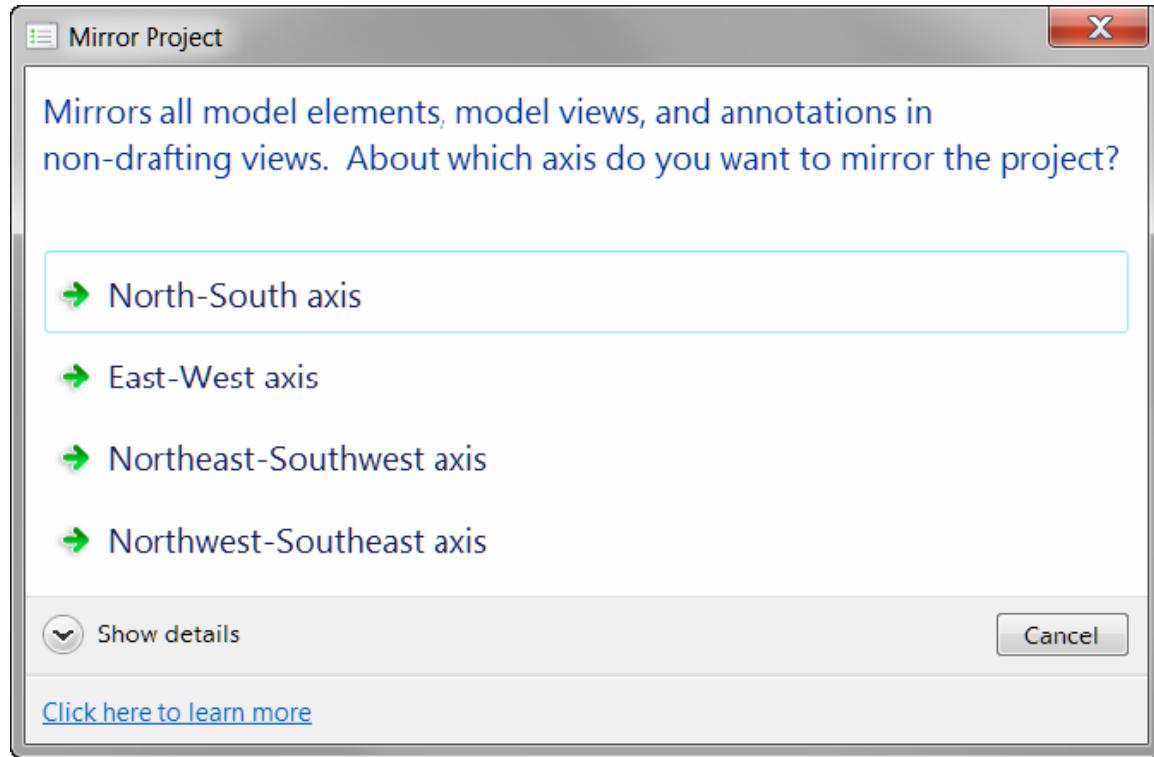


Figure 227 - The Show Details button displays additional Main Content text

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

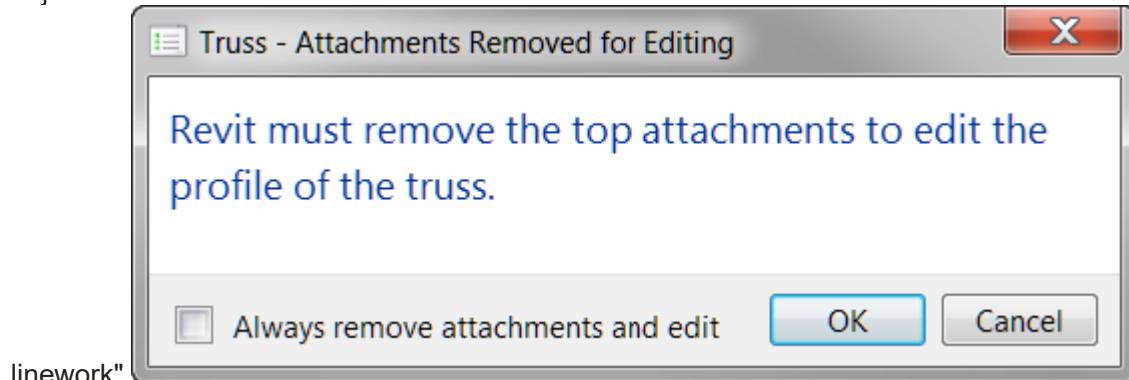


Figure 228 - Example of a task dialog using the DNSMA checkbox as an "Always..." checkbox for one choice

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.

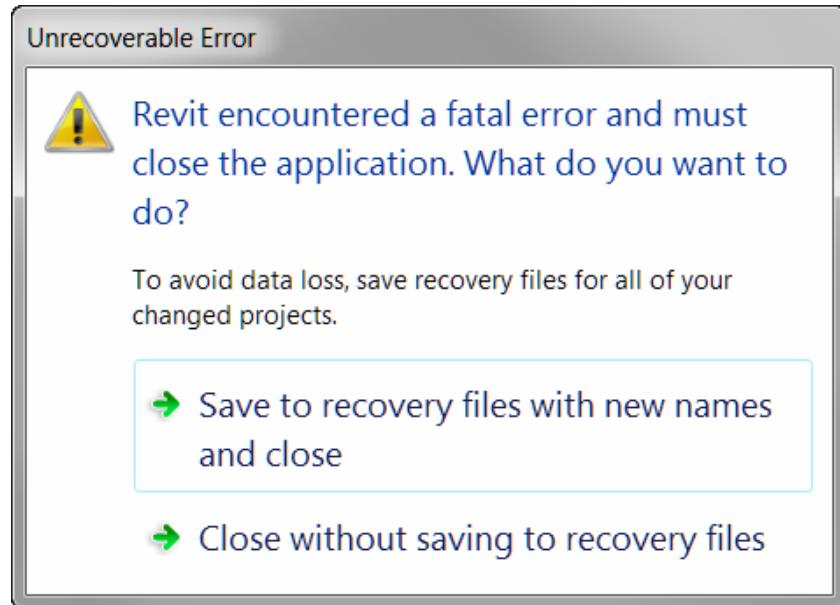


Figure 229 - A task dialog with two command links

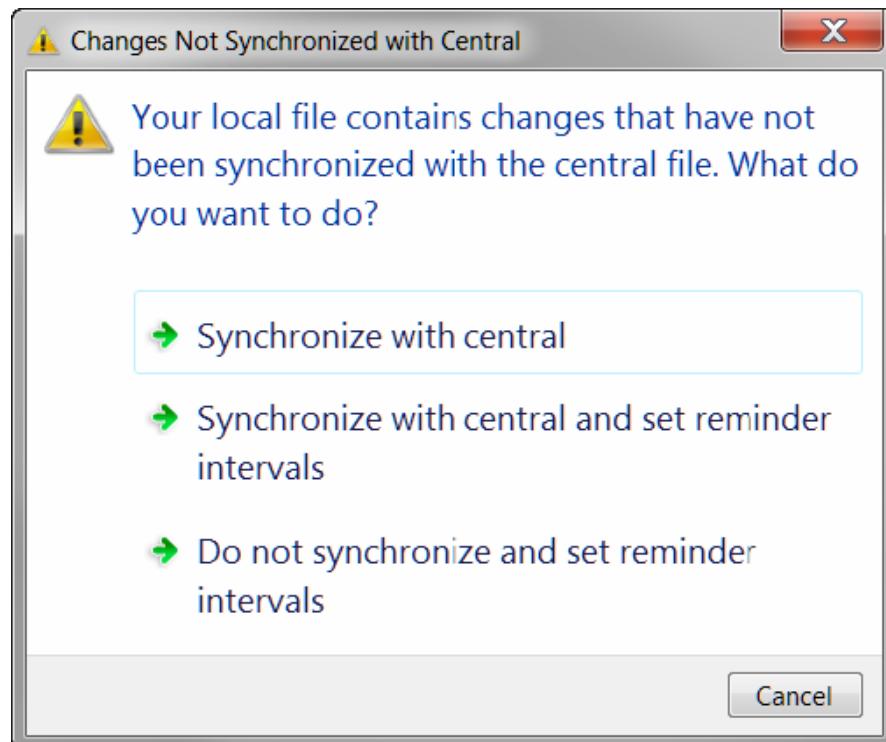


Figure 230 - Task Dialog with command links and a command button

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

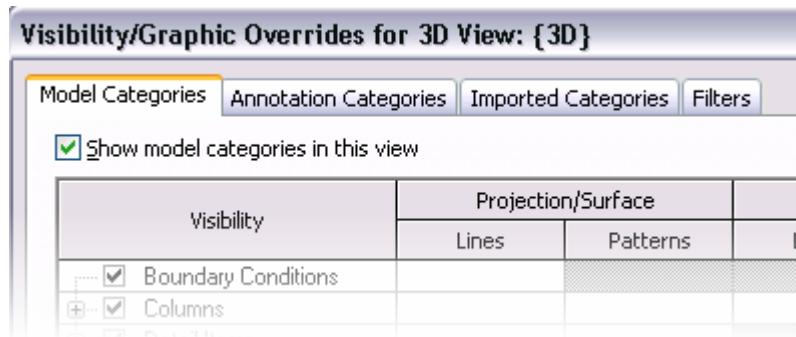


Figure 231 - Horizontal Tabs

Avoid more than one row of horizontal tabs. If a second row is needed, consider a vertical tab.

Variation B: Vertical Tabs

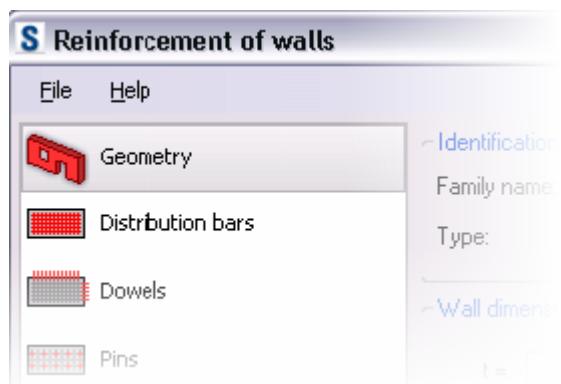


Figure 232 - Vertical Tabs

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.

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.

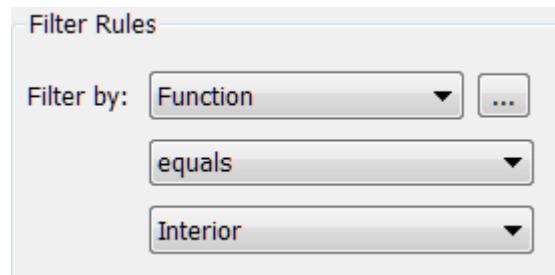


Figure 233 -Dialog launcher button, as implemented in the Revit View Filters dialog

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

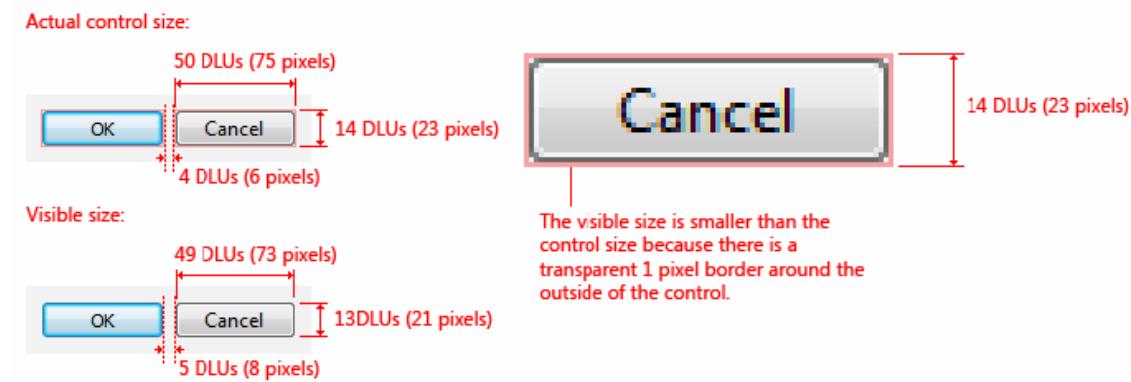


Figure 234 - Commit Button sizes (taken from Microsoft Windows User Experience Guidelines)

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.

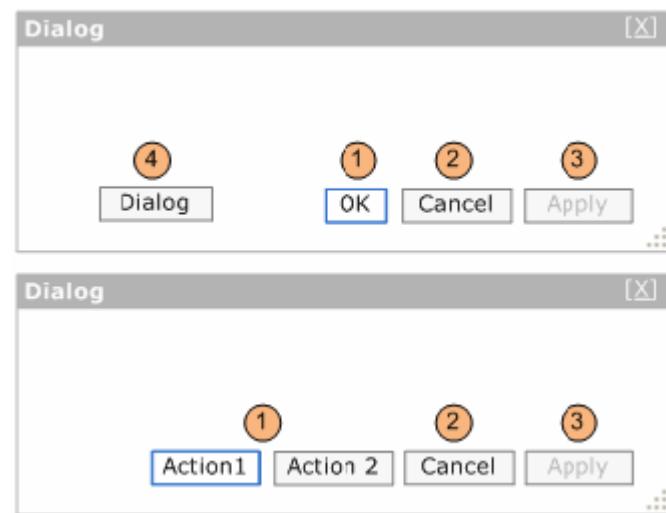


Figure 235 - Standard Commit Button layouts

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

6.5.9 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 2012 product line 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 236 - 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.)

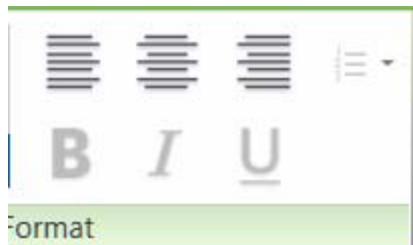


Figure 237 - The Format text panel from Revit 2011

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.

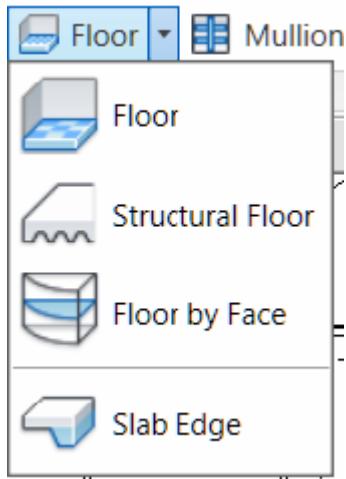


Figure 238 - Floor drop-down button in Revit

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

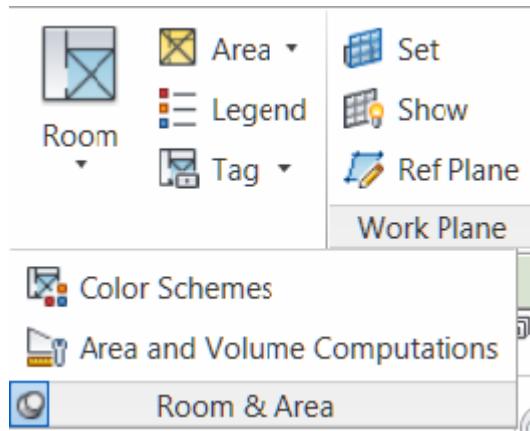


Figure 239 - Room & Area slide-out panel in Revit

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"

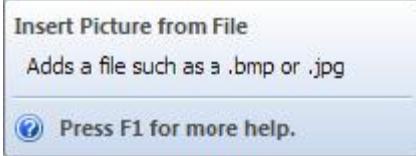
Writing/Wording Considerations

- 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

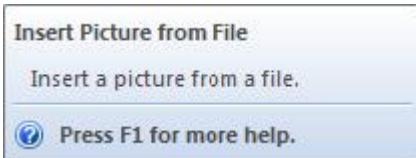
- Use only one space between sentences.
- Avoid repetitive text. The content in the tooltip should be unique and add value.
- Focus on the quality and understandability of the tooltip. Is the description clear? Is it helpful?
- Unless it's a system variable or command, do not use bold. Although bold is supported in Asian languages, it is strongly recommended to avoid using bold and italics, because of readability and stylistic issues.
- Avoid Dabbreviations. 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 .png". 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

6.5.10 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 Commands registered in a .addin manifest file will appear in this menu button.

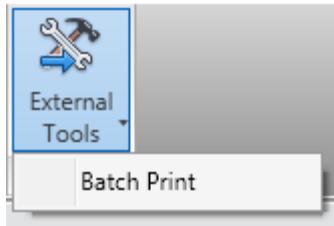


Figure 240 - External Tools menu-button on Add-Ins tab

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.

6.5.11 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
------------	---

7 FAQ

Questions and answers about the Revit API language environment, development tools, best practices, and other common issues.

Development Environment

What versions of Visual Studio do I need for Revit API development? Can I use Visual C# Community?

- To edit and debug your API applications, you need an interactive development environment such as Microsoft Visual Studio 2019 Professional or one of the MS Visual Studio Community Editions for C# or Visual Basic.NET

What languages are supported for Revit API development?

- C#, VB.NET, and C++/CLI are supported for addin development. C# is supported for macro development. Other CLR languages may work with the Revit API, but they are untested and unsupported. Note that for mixed managed/native applications using C++, Revit uses the Visual C++ Redistributables for Visual Studio 2019 and 2022. Add-ins compiled with other versions of the VC runtime may not work correctly since the Revit install does not include any other VC runtimes.

Addins

Is there a wizard to help generate Revit Addins?

- Yes. Please see <http://thebuildingcoder.typepad.com/blog/2014/05/add-in-wizards-for-revit-2015.html>

Why doesn't my C++ addin build?

- You may be missing the `<TargetFrameworkVersion>v4.5.2</TargetFrameworkVersion>` node in the `<PropertyGroup Label="Globals">` section of your .vcxproj file -- the Visual C++ default framework without this node is .NET 4.0.

Why doesn't my external command show up?

- There is a variety of reasons. Is your .addin manifest file properly installed to C:\ProgramData\Autodesk\Revit\Addins\ 20xx? Is the path to your DLL in the .addin correct?

Why isn't my application's OnStartup method called when Revit starts?

- Check the .addin manifest from the question above. Also, try checking to see if an exception is called in the constructor of your addin object when Revit starts.

API Usage

How do I reference an element in Revit?

- 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.

Can a model only use one shared parameter file?

- 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.

Do I need to distribute the shared parameters file with the model so other programs can use the shared parameters?

- 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.

Are shared parameter values copied when the corresponding element is copied?

- 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.

Are element Unique IDs (UID) universally unique and can they ever change?

- The element UIDs are not universally unique but they are unique within a model as are element IDs. They are not universally unique since they can be duplicated when creating one document by copying another. However, UniqueIDs are stable for referencing elements in a worksharing environment because they will not change. Unlike UniqueIDs, regular IDs

can be renumbered when synching with a central model, therefore they cannot be considered stable for referencing of elements.

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?

- Make sure you only call Document.Regenerate() as often as necessary. Although this method is required to make sure the elements in the Revit document reflect all changes, it can slow down your application. Keep in mind, too, that when a transaction is committed there is an automatic call to regenerate the document.

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.

- 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.

How do I access the saved models and content BMP?

- 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 [Revit API Developers Guide](#) for Autodesk Revit").

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.

Why is Element.Parameters taking so long to access?

- This property executes many calculations internally. Caching specific properties in a separate data structure may be a good idea, depending on your applications' needs.

Structural Engineering

Sometimes the default end releases of structural elements render the model unstable. How can I deal with this situation?

- 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.

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?

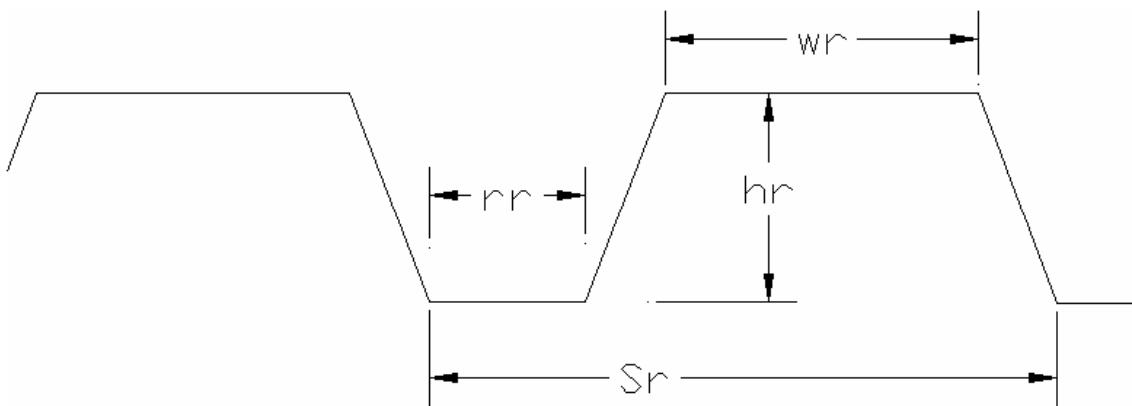
- Take a look at the RotateFramingObject example in the SDK. It has examples of how to get and change the beam braces and columns rotation angle.

How do I add new concrete beam and column sizes to a model?

- Take a look at the FrameBuilder sample code in the SDK

How do I view the true deck layer?

- 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.



How do I tell when I have a beam with a cantilever?

- 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:

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.

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."

- 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.

General

Is the API binary compatible between major releases?

- Partially. Any API marked obsolete will be supported for that release and removed in the following release. Therefore, your application must be up to date with the latest APIs at least every other release to remain compatible.