

Revit 2010 API

Developer's Guide

Version 1.0

The Autodesk logo is displayed vertically in white text on a black rectangular background. The word "Autodesk" is written in a sans-serif font, with a registered trademark symbol (®) at the top right.

March, 2009

Copyright. 2009 Autodesk, Inc.

All Rights Reserved

This publication, or parts thereof, may not be reproduced in any form, by any method, for any purpose.

AUTODESK, INC., MAKES NO WARRANTY, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS, AND MAKES SUCH MATERIALS AVAILABLE SOLELY ON AN "AS-IS" BASIS. IN NO EVENT SHALL AUTODESK, INC., BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF ACQUISITION OR USE OF THESE MATERIALS. THE SOLE AND EXCLUSIVE LIABILITY TO AUTODESK, INC., REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE, IF ANY, OF THE MATERIALS DESCRIBED HEREIN.

Autodesk, Inc., reserves the right to revise and improve its products as it sees fit. This publication describes the state of the product at the time of publication, and may not reflect the product at all times in the future.

Autodesk Trademarks

The following are registered trademarks or trademarks of Autodesk, Inc., in the USA and other countries: 3DEC (design/logo), 3December, 3December.com, 3ds Max, ActiveShapes, Actrix, ADI, Alias, Alias (swirl design/logo), AliasStudio, Alias|Wavefront (design/logo), ATC, AUGI, AutoCAD, AutoCAD Learning Assistance, AutoCAD LT, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, Autodesk, Autodesk Envision, Autodesk Insight, Autodesk Intent, Autodesk Inventor, Autodesk Map, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSnap, AutoSketch, AutoTrack, Backdraft, Built with ObjectARX (logo), Burn, Buzzsaw, CAiCE, Can You Imagine, Character Studio, Cinestream, Civil 3D, Cleaner, Cleaner Central, ClearScale, Colour Warper, Combustion, Communication Specification, Constructware, Content Explorer, Create>what's>Next> (design/logo), Dancing Baby (image), DesignCenter, Design Doctor, Designer's Toolkit, DesignKids, DesignProf, DesignServer, DesignStudio, Design|Studio (design/logo), Design Your World, Design Your World (design/logo), DWF, DWG, DWG (logo), DWG TrueConvert, DWG TrueView, DXF, EditDV, Education by Design, Extending the Design Team, FBX, Filmbox, FMDesktop, GDx Driver, Gmax, Heads-up Design, Heidi, HOOPS, HumanIK, i-drop, iMOUT, Incinerator, IntroDV, Kaydara, Kaydara (design/logo), LocationLogic, Lustre, Maya, Mechanical Desktop, MotionBuilder, ObjectARX, ObjectDBX, Open Reality, PolarSnap, PortfolioWall, Powered with Autodesk Technology, Productstream, ProjectPoint, Reactor, RealDWG, Real-time Roto, Render Queue, Revit, Showcase, SketchBook, StudioTools, Topobase, Toxik, Visual, Visual Bridge, Visual Construction, Visual Drainage, Visual Hydro, Visual Landscape, Visual Roads, Visual Survey, Visual Syllabus, Visual Toolbox, Visual Tugboat, Visual LISP, Voice Reality, Volo, and Wiretap. The following are registered trademarks or trademarks of Autodesk Canada Co. in the USA and/or Canada and other countries: Backburner, Discreet, Fire, Flame, Flint, Frost, Inferno, Multi-Master Editing, River, Smoke, Sparks, Stone, Wire.

Third Party Trademarks

All other brand names, product names or trademarks belong to their respective holders.

Third Party Software Program Credits

ACIS Copyright. 1989-2001 Spatial Corp. Portions Copyright. 2002 Autodesk, Inc.

Copyright. 1997 Microsoft Corporation. All rights reserved.

Flash 2 is a registered trademark of Macromedia, Inc. in the United States and/or other countries.

International CorrectSpell™ Spelling Correction System. 1995 by Lernout & Hauspie Speech Products, N.V. All rights reserved.

InstallShield™ 3.0. Copyright. 1997 InstallShield Software Corporation. All rights reserved.

PANTONE2 Colors displayed in the software application or in the user documentation may not match PANTONE-identified standards.

Consult current PANTONE Color Publications for accurate color.

PANTONE2 and other Pantone, Inc. trademarks are the property of Pantone, Inc., Pantone, Inc., 2002

Pantone, Inc. is the copyright owner of color data and/or software which are licensed to Autodesk, Inc., to distribute for use only in combination with certain Autodesk software products. PANTONE Color Data and/or Software shall not be copied onto another disk or into memory unless as part of the execution of this Autodesk software product.

Portions Copyright. 1991-1996 Arthur D. Applegate. All rights reserved.

Portions of this software are based on the work of the Independent JPEG Group.

RAL DESIGN. RAL, Sankt Augustin, 2002

RAL CLASSIC. RAL, Sankt Augustin, 2002

Representation of the RAL Colors is done with the approval of RAL Deutsches Institut für Gütesicherung und Kennzeichnung e.V. (RAL German Institute for Quality Assurance and Certification, re. Assoc.), D-53757 Sankt Augustin.

Typefaces from the Bitstream2 typeface library copyright 1992.

Typefaces from Payne Loving Trust. 1996. All rights reserved.

AutoCAD 2008 is produced under a license of data derived from DIC Color Guide2 from Dainippon Ink and Chemicals, Inc. Copyright . Dainippon Ink and Chemicals, Inc. All rights reserved. DIC Color Guide computer color simulations used in this product may not exactly match DIC Color Guide, DIC color Guide Part 2 identified solid color standards. Use current DIC Color Guide Manuals for exact color reference. DIC and DIC Color Guide are registered trademarks of Dainippon Ink and Chemicals, Inc.

Printed manual and help produced with Idiom WorldServer™.

WindowBlinds: DirectSkin™ OCX. Stardock®

AnswerWorks 4.0.; 1997-2003 WexTech Systems, Inc. Portions of this software. Vantage-Knexys. All rights reserved.

The Director General of the Geographic Survey Institute has issued the approval for the coordinates exchange numbered TKY2JGD for Japan Geodetic Datum 2000, also known as technical information No H1-N0.2 of the Geographic Survey Institute, to be installed and used within this software product (Approval No.: 646 issued by GSI, April 8, 2002).

Portions of this computer program are copyright . 1995-1999 LizardTech, Inc. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

Portions of this computer program are Copyright.; 2000 Earth Resource Mapping, Inc.

OSTN97. Crown Copyright 1997. All rights reserved.

OSTN02. Crown copyright 2002. All rights reserved.

OSGM02. Crown copyright 2002. . Ordnance Survey Ireland, 2002.

FME Objects Engine. 2005 SAFE Software. All rights reserved.

GOVERNMENT USE

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in FAR 12.212 (Commercial Computer Software-Restricted Rights) and DFAR 227.7202 (Rights in Technical Data and Computer Software), as applicable.

Table of Contents

1	Welcome to the Revit Platform API	12
1.1	Introduction to the Revit Platform API	12
1.2	What Can You Do with the Revit Platform API?	12
1.3	Requirements.....	12
1.4	Install and Learn the Revit-Based Product.....	13
1.5	Installation	13
1.6	Supported Programming Languages	13
1.7	User Manual	13
1.8	Documentation Conventions.....	15
1.9	Object Model Diagram	15
1.10	What's new in this release	15
2	Getting Started	16
2.1	Walkthroughs.....	16
2.2	Walkthrough: Hello World	16
2.3	Walkthrough: Add Hello World Ribbon Panel.....	23
2.4	Walkthrough: Retrieve Selected Elements.....	25
2.5	Walkthrough: Retrieve All Elements	26
2.6	Walkthrough: Retrieve Filtered Elements	27
3	Add-In Integration	29
3.1	Overview	29
3.2	External Commands.....	30
3.3	External Application	35
3.4	Ribbon Panels and Controls.....	38
3.5	Using the Add-in Manager.....	40
4	Application and Document	42
4.1	Application Functions	42
4.2	Document Functions	43
4.3	Document and File Management	45
4.4	Settings.....	46
4.5	Units	48
5	Elements Essentials	49
5.1	Element Classification	49
5.2	Other Classifications.....	54
5.3	APIObject	59
5.4	Element Retrieval	60
5.5	General Properties	64

6	Parameters	69
6.1	Walkthrough: Get Selected Element Parameters	69
6.2	Definition.....	71
6.3	BuiltInParameter	73
6.4	StorageType	73
6.5	AsValueString() and SetValueString().....	75
6.6	Parameter Relationships	75
6.7	Adding Parameters to Elements	76
7	Collections	77
7.1	Interface	77
7.2	Collections and Iterators.....	78
8	Editing Elements.....	82
8.1	Moving Elements	82
8.2	Rotating elements.....	85
8.3	Mirror.....	87
8.4	Group.....	88
8.5	Array	89
8.6	Delete	90
8.7	SuspendUpdating.....	92
9	Walls, Floors, Roofs and Openings.....	94
9.1	Walls.....	94
9.2	Floors and Foundations.....	96
9.3	Roofs	99
9.4	Curtains	101
9.5	Other Elements	101
9.6	CompoundStructure.....	102
9.7	Opening	103
10	Family Instances	108
10.1	Identifying Elements	108
10.2	Family	109
10.3	FamilyInstances	110
10.4	Code Samples	119
11	Family Creation.....	125
11.1	Family Documents	125
11.2	Family Item Factory	126
11.3	Family Element Visibility.....	132
11.4	Family Manager	133
12	Conceptual Design	136

12.1	Point and curve objects	136
12.2	Forms	139
12.3	Rationalizing a Surface	145
13	Datum and Information Elements	151
13.1	Levels	152
13.2	Grids	154
13.3	Phase	157
13.4	Design Options	158
14	Annotation Elements	160
14.1	Dimensions and Constraints	161
14.2	Detail Curve	166
14.3	Tags	167
14.4	Text	169
14.5	Annotation Symbol	169
15	Sketching	171
15.1	The 2D Sketch Class	171
15.2	3D Sketch	174
15.3	ModelCurve	182
16	Views	185
16.1	Overview	185
16.2	The View3D Class	192
16.3	ViewPlan	198
16.4	ViewDrafting	198
16.5	ViewSection	199
16.6	ViewSheet	201
17	Material	204
17.1	General Material Information	204
17.2	Material Management	210
17.3	Element Material	212
18	Geometry	220
18.1	Example: Retrieve Geometry Data from a Wall	220
18.2	Geometry Node Class	222
18.3	Geometry Helper Class	226
18.4	Collection Classes	236
18.5	Example: Retrieve Geometry Data from a Beam	237
19	Place and Locations	239
19.1	Place	239
19.2	City	240

19.3	ProjectLocation.....	240
19.4	Project Position	241
20	Shared Parameters	246
20.1	Definition File.....	246
20.2	Definition File Access.....	248
20.3	Binding.....	252
21	Transactions	257
21.1	Usage	257
21.2	Boundaries	260
22	Events.....	262
22.1	Application Events	262
22.2	Document Events	263
22.3	Registering Events	263
22.4	Canceling Events	264
23	Revit Architecture.....	265
23.1	Rooms.....	265
23.2	Energy Data.....	277
24	Revit Structure	279
24.1	Structural Model Elements	279
24.2	AnalyticalModel	289
24.3	Loads	295
24.4	Analysis Link.....	296
25	Revit MEP	298
25.1	MEP Element Creation	298
25.2	Connectors	303
25.3	Family Creation	305
A.	Glossary.....	308
B.	FAQ.....	310
C.	Hello World for VB.NET	313
D.	Material Properties Internal Units	318
E.	Concrete Section Definitions.....	321
F.	API Ribbon Layout Guidelines.....	334

Table of Code Samples

Code Region 2-1: Getting Started	18
Code Region 2-2: Modifying the Revit.ini.....	19
Code Region 2-3: Exceptions in Execute()	21
Code Region 2-4: Using try catch in execute:	22
Code Region 2-5: Adding a ribbon panel	23
Code Region 2-6: Adding an external application to Revit.ini	24
Code Region 2-7: Retrieving selected elements.....	25
Code Region 2-8: Retrieving all elements	26
Code Region 2-9: Retrieve filtered elements	27
Code Region 3-1: Retrieving the Active Document.....	30
Code Region 3-2: Setting an error message string	31
Code Region 3-3: Highlighting walls	32
Code Region 3-4: Prompting the user	34
Code Region 3-5: Revit.ini ExternalCommands Section	35
Code Region 3-6: OnShutdown() and OnStartup().....	36
Code Region 3-7: DialogBoxShowing Event	36
Code Region 3-8: Using ControlledApplication.....	36
Code Region 3-9: Revit.ini ExternalApplications section.....	37
Code Region 3-10: Ribbon panel and controls	38
Code Region 5-1: Getting categories from document settings	55
Code Region 5-2: Getting element category	57
Code Region 5-3: Accessing APIObject.IsReadOnly.....	59
Code Region 5-4: Comparing Elements	60
Code Region 5-5: Filtering elements	61
Code Region 5-6: Creating a logical filter	61
Code Region 5-7: Changing selected elements.....	62
Code Region 5-8: Adding selected elements with PickOne() and WindowSelect().....	63
Code Region 5-9: Setting ElementID	65
Code Region 5-10: Using ElementID	65
Code Region 5-11: UniqueID	65
Code Region 5-12: Assigning Level.....	67
Code Region 6-1: Getting selected element parameters	69
Code Region 6-2: Finding a parameter based on definition type	71
Code Region 6-3: Getting a parameter based on BuiltInParameter	73
Code Region 6-4: Checking a parameter's StorageType	74
Code Region 6-5: Using Parameter.SetValueString()	75

Code Region 6-6: Parameter relationship example	76
Code Region 7-1: Using ElementSet and ElementIterator	78
Code Region 7-2: Using collections.....	80
Code Region 8-1: Using Move()	82
Code Region 8-2: Moving using Location	83
Code Region 8-3: Moving using Curve	84
Code Region 8-4: Moving using Point.....	84
Code Region 8-5: Using Rotate().....	86
Code Region 8-6: Rotating based on location curve	86
Code Region 8-7: Rotating based on location point.....	87
Code Region 8-8: Mirroring a column.....	87
Code Region 8-9: Creating a Group	88
Code Region 8-10: Naming a Group	88
Code Region 8-11: Deleting an element based on object.....	90
Code Region 8-12: Deleting an element based on ElementId	91
Code Region 8-13: Deleting an ElementSet	91
Code Region 8-14: Deleting an ElementSet based on Id	91
Code Region 8-15: Using SuspendUpdating	92
Code Region 8-16: Nesting SuspendUpdating	93
Code Region 9-1: NewSlab()	97
Code Region 9-2: Reverting a slab's shape	99
Code Region 9-3: Creating a footprint roof	100
Code Region 9-4: Modifying a gutter	101
Code Region 9-5: Getting the CompoundStructureLayer Material	102
Code Region 9-6: Retrieving existing opening properties.....	105
Code Region 9-7: NewOpening()	106
Code Region 10-1: Getting SubComponents and SuperComponent from FamilyInstance	113
Code Region 10-2: Batch creating family instances.....	116
Code Region 10-3: Creating tables	120
Code Region 10-4: Creating a beam	121
Code Region 10-5: Creating doors.....	122
Code Region 10-6: Creating FamilyInstances using reference directions.....	123
Code Region 11-1: Category of open Revit Family Document.....	125
Code Region 11-2: Creating a new Family document	125
Code Region 11-3: Getting nested Family symbols in a Family.....	126
Code Region 11-4: Creating a new Extrusion	127
Code Region 11-5: Creating a new Sweep	128
Code Region 11-6: Assigning a subcategory	129

Code Region 11-7: Creating a Dimension	130
Code Region 11-8: Labeling a dimension.....	131
Code Region 11-9: Setting family element visibility	132
Code Region 11-10: Getting the types in a family.....	133
Code Region 11-11: Editing Family Types.....	134
Code Region 12-1: Creating a new CurveByPoints	136
Code Region 12-2: Using Reference Lines to create Form	137
Code Region 12-3: Creating an extrusion form.....	139
Code Region 12-4: Creating a loft form.....	141
Code Region 12-5: Moving a profile.....	143
Code Region 12-6: Moving a sub element	144
Code Region 12-7: Dividing a surface	145
Code Region 12-8: Patterning a surface	147
Code Region 12-9: Editing a curtain panel family	148
Code Region 13-1: Retrieving all Levels	153
Code Region 13-2: Creating a new Level.....	154
Code Region 13-3: Using the Grid class	155
Code Region 13-4: NewGrid()	156
Code Region 13-5: Creating a grid with a line or an arc	156
Code Region 13-6: Displaying all supported phases	158
Code Region 13-7: Using design options	158
Code Region 14-1: Distinguishing permanent dimensions from constraints.....	161
Code Region 14-2: NewDimension()	166
Code Region 14-3: Duplicating a dimension with NewDimension().....	166
Code Region 14-4: NewDetailCurve() and NewModelCurve()	166
Code Region 14-5: Creating an IndependentTag	168
Code Region 14-6: NewAnnotationSymbol()	170
Code Region 14-7: Using addLeader() and removeLeader().....	170
Code Region 15-1: Creating a new SketchPlane	174
Code Region 15-2: Creating a new model curve.....	183
Code Region 15-3: Getting a specific Curve from a ModelCurve	184
Code Region 16-1: Determining the View class type	189
Code Region 16-2: Determining the View type.....	190
Code Region 16-3: Counting elements in the active view.....	191
Code Region 16-4: NewView3D()	196
Code Region 16-5: Creating a 3D view.....	196
Code Region 16-6: Showing the Section Box	196
Code Region 16-7: Hiding the Section Box	197

Code Region 16-8: NewViewPlan()	198
Code Region 16-9: Creating a floor plan and ceiling plan.....	198
Code Region 16-10: NewViewSection()	199
Code Region 16-11: AddView()	201
Code Region 16-12: Creating a sheet view	201
Code Region 17-1: Downcasting using RTTI.....	204
Code Region 17-2: Determining material type	206
Code Region 17-3: Getting a material parameter	207
Code Region 17-4: Setting the fill pattern	209
Code Region 17-5: Getting a material by name.....	210
Code Region 17-6: Duplicating a material.....	211
Code Region 17-7: Adding a new Material	211
Code Region 17-8: Removing a material	212
Code Region 17-9: Getting an element material from a parameter	213
Code Region 17-10: Getting window materials walkthrough.....	218
Code Region 18-1: Creating Geometry.Options	220
Code Region 18-2: Retrieving faces and edges.....	221
Code Region 18-3: Getting curves from an instance	223
Code Region 18-4: Getting solid information from an instance	224
Code Region 18-5: Drawing the geometry of a mass	225
Code Region 18-6: Getting a solid from a GeometryObject	226
Code Region 18-7: Transformation example	227
Code Region 18-8: Using the Reflection property	228
Code Region 18-9: Rotating BoundingBoxXYZ.....	235
Code Region 18-10: Getting solids and curves from a beam	237
Code Region 19-1: Retrieving the ProjectLocation object.....	242
Code Region 19-2: Creating a project location	244
Code Region 19-3: Deleting a project location	244
Code Region 20-1: Parameter definition file example.....	246
Code Region 20-2: Shared Parameter FAMILYTYPE example	247
Code Region 20-3: Creating a shared parameter file.....	250
Code Region 20-4: Getting the definition file from an external parameter file	250
Code Region 20-5: Traversing parameter entries	251
Code Region 20-6: Changing parameter definition group owner	251
Code Region 20-7: Adding type parameter definitions using a shared parameter file	253
Code Region 20-8: Adding instance parameter definitions using a shared parameter file	255
Code Region 21-1: Using transactions	258
Code Region 21-2: Transaction populating Geometry and AnalyticalModel properties.....	259

Code Region 21-3: Transaction boundaries.....	260
Code Region 22-1: Registering Application.DocumentOpened	263
Code Region 22-2: Canceling an Event	264
Code Region 23-1: Creating a room.....	266
Code Region 23-2: Creating and inserting a room into a plan circuit	266
Code Region 23-3: Setting room bounding	270
Code Region 23-4: Using a transaction to update room boundary.....	272
Code Region 23-5: Getting a room from the family instance	276
Code Region 23-6: Getting a room's dimensions	276
Code Region 23-7: Using the gbXMLParamElem class	278
Code Region 24-1: Distinguishing between column, beam and brace	279
Code Region 24-2: Using BuiltInCategory.OST_StructuralFraming.....	280
Code Region 24-3: NewAreaReinforcement()	281
Code Region 24-4:NewPathReinforcement()	282
Code Region 24-5: Downcasting Face to PlanarFace	282
Code Region 24-6: Getting Normal and Origin	282
Code Region 24-7: Creating a truss over two columns	283
Code Region 24-8: Creating rebar with a specific layout.....	285
Code Region 24-9: Getting boundary condition type and geometry.....	286
Code Region 24-10: Getting AnalyticalModel.....	290
Code Region 24-11: NewLoadCombination().....	296
Code Region 25-1: Creating a new Pipe	298
Code Region 25-2: Changing pipe diameter.....	299
Code Region 25-3: Adding a duct between two connectors	299
Code Region 25-4: Creating a new mechanical system.....	302
Code Region 25-5: Determine what is attached to a connector	304
Code Region 25-6: Adding a pipe connector	305
Code Region 25-7: Hello World in VB.NET	315
Code Region 25-8: Revit.ini ExternalCommands Section	316
Code Region 25-9: MsgBox("Hello World")	317

1 Welcome to the Revit Platform API

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

1.1 Introduction to the Revit Platform API

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

Revit Architecture 2010, Revit Structure 2010, and Revit MEP 2010 all contain the Revit Platform API so that you can integrate your applications into Revit. The three APIs are very similar and are jointly referred to as the Revit Platform API. Before using the API, learn to use Revit and its features so that you can better understand the relevant areas related to your programming. Learning Revit can help you:

Maintain consistency with the Revit UI and commands.

Design your add-in application seamlessly.

Master API classes and class members efficiently and effectively.

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

1.2 What Can You Do with the Revit Platform API?

You can use the Revit Platform API to:

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

1.3 Requirements

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

1. A working understanding of Revit Architecture 2010, Revit Structure 2010, and Revit MEP 2010.
2. You must know at least one Common Language Specification compliant language. For more information about CLS-compliant languages, review the information at the following link: [http://msdn.microsoft.com/en-us/library/12a7a7h3\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/12a7a7h3(VS.80).aspx)
3. Microsoft Visual Studio 2008, or Microsoft Visual Studio 2008 Express Edition. Alternatively, you can use the built-in Visual Studio Tools for Applications (VSTA) IDE in Revit. You must

install VSTA from the Revit DVD separately. For more information on writing applications using VSTA, see the section "Creating Macros with Revit VSTA" in the Revit 2010 User's Guide that can be accessed through the Revit application.

4. Microsoft .NET Framework 3.5 if not already installed with Visual Studio.
5. The Revit Software Developer's Kit (SDK) which you can download from the Autodesk Developer Network (ADN) or the Revit installation CD/DVD
(<DVD_Drive>:\Utilities\Common\Software Development Kit).

1.4 Install and Learn the Revit-Based Product

The Revit Platform API requires the Microsoft .NET Framework 3.5 SDK. The examples in this document use Microsoft Visual Studio 2008 for the Integrated Development Environment (IDE).

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

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

www.autodesk.com/revitbuilding

www.autodesk.com/revitstructure

www.autodesk.com/revitsystems

www.autodesk.com/bim

www.revitcity.com

www.augi.com

<http://discussion.autodesk.com>

Select = Autodesk Revit Architecture

Then Select = Autodesk Revit API

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

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

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

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

1.5 Installation

The Revit Platform API is installed with Revit Architecture, Revit Structure, and Revit MEP. Any .NET based application will reference the RevitAPI.dll located in the Revit Program directory.

1.6 Supported Programming Languages

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

1.7 User Manual

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

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

1.7.1 Introduction to the Revit Platform API

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

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

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

1.7.2 Basic Topics

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

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

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

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

[Parameters](#) - Most Element information is stored as Parameters. This chapter discusses Parameter functionality.

[Collection](#) - Utility collection types such as Array, Map, Set collections, and related Iterators.

1.7.3 Element Topics

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

[Editing Elements](#) - Learn how to move, rotate, delete, mirror, group, and array one element or a set of elements.

[Wall, Floors, Roofs and Openings](#) - Discusses Elements, their corresponding Symbols 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.

[Datum and Information Elements](#) - Learn how to set up grids, add levels, use design options, and more.

[Annotation Elements](#) - Discusses document annotation including adding dimensions, detail curves, tags, and annotation symbols.

[Sketching](#) - Sketch functions include 2D and 3D sketch classes such as SketchPlane, ModelCurve, GenericForm, and more.

[Views](#) - Learn about the different ways to view models and components and how to manipulate the view in the API.

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

1.7.4 Advanced Topics

Before reviewing the advanced topics, be sure to review the basic and Element topics.

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

[Place and Locations](#) – Defines the project location including city, country, latitude, and longitude.

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

[Transaction](#) – Introduces the two uses for Transaction and the limits that you must consider when using Transaction.

[Events](#) – Discusses how to take advantage of Revit Events.

1.7.5 Product Specific

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

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

[Revit Structure](#) - Discusses the APIs specific to Revit Structure.

[Revit MEP](#) – Discusses the APIs specific to Revit MEP.

1.7.6 Other

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

[Appendix](#) – Additional information such as Frequently Asked Questions, Using Visual Basic.Net for programming, and more.

1.8 Documentation Conventions

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

1.8.1 Indexed Properties

Some Revit Platform API class properties are “indexed”, or described as overloaded in the API help file (RevitAPI.chm). For example, the Document.Element property has two overloads. In the text of this document, these are referred to as properties, although you access them as if they were methods in C# code by pre-pending the property name with “get_” or “set_”. For example, to use the Document.Element(String) property overload, you use Document.get_Element(String).

1.9 Object Model Diagram

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

1.10 What's new in this release

Please see the Getting Started Revit API 2010.doc included in the Revit 2010 SDK for information about changes and new features.

2 Getting Started

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

2.1 Walkthroughs

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

[Walkthrough: Hello world](#) - Illustrates how to create an add-in using the Revit Platform API.

[Walkthrough: Add Hello World Ribbon Panel](#) - Illustrates how to add a custom ribbon panel.

[Walkthrough: Retrieve Selected Elements](#) - Illustrates how to retrieve selected elements.

[Walkthrough: Retrieve All Elements](#) - Illustrates how to retrieve all document elements.

[Walkthrough: Retrieve Filtered Elements](#) - Illustrates how to retrieve elements based on filter criteria.

2.2 Walkthrough: Hello World

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

The Hello World walkthrough covers the following topics:

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

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

2.2.1 Create a New Project

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

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

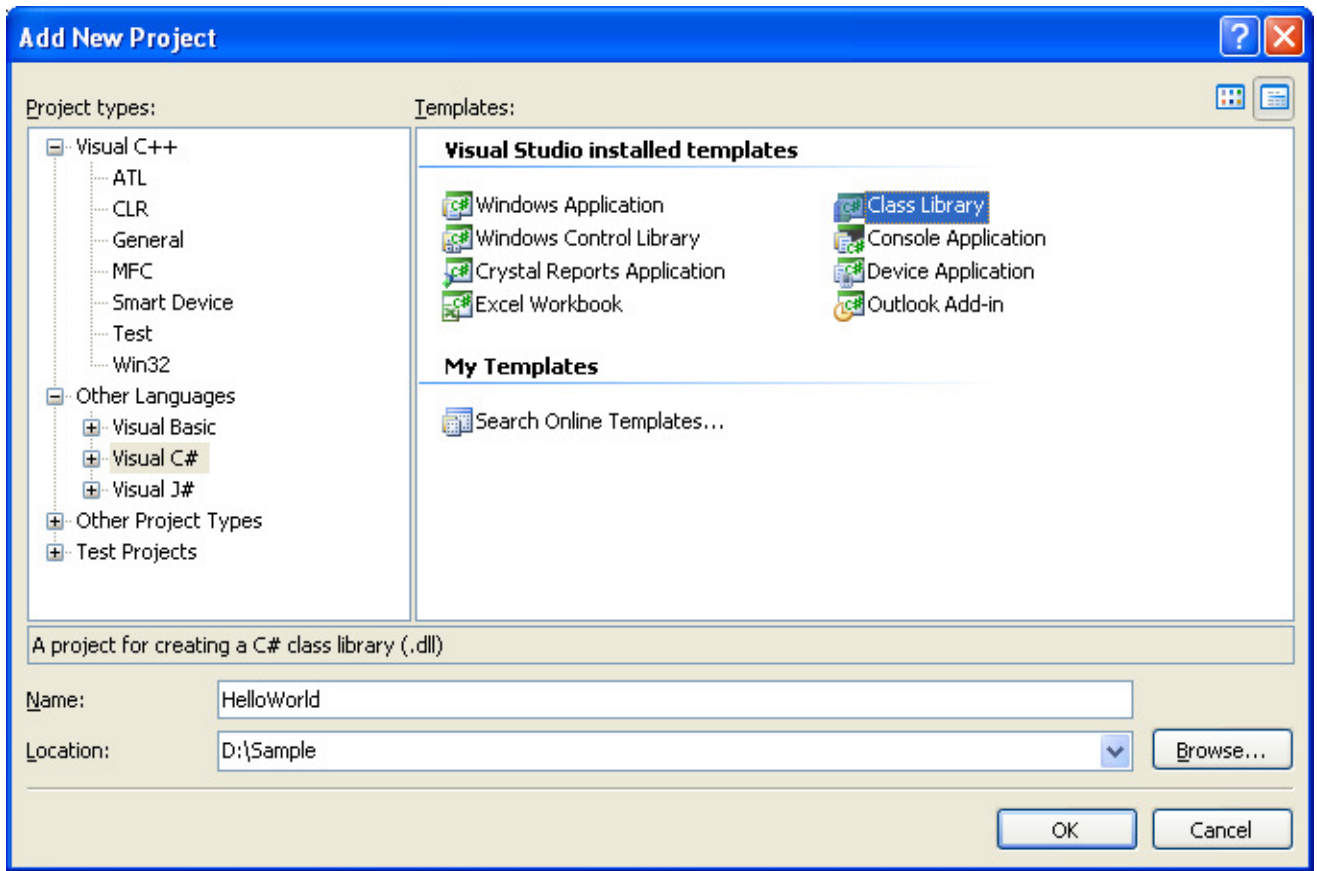


Figure 1: Add New Project

2.2.2 Add References

1. To add the RevitAPI reference:
 - From the View menu select Solution Explorer if the Solution Explorer window is not open.
 - In the Solution Explorer, right-click References to display a context menu.
 - From the context menu, click Add Reference. The Add Reference dialog box appears (see Figure 2: Add Reference below).
 - In the Add Reference dialog box, click the Browse tab. Locate the folder where Revit is installed and click the RevitAPI.dll. For example, the installed folder location is usually C:\Program Files\Autodesk Revit Architecture 2010\Program\RevitAPI.dll.
 - Click OK to select the .dll and close the dialog box. RevitAPI appears in the Solution Explorer reference tree.
 - **Note:** You should always set the Copy Local property of RevitAPI.dll to false for new projects. This saves disk space, and prevents the Visual Studio debugger from getting confused about which copy of the DLL to use. Right-click the RevitAPI.dll, select Properties, and change the Copy Local setting from true (the default) to false.
2. Add the System.Windows.Forms 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 System.Windows.Forms.

- Click OK to close the dialog box. System.Windows.Forms appears in the Solution Explorer reference tree.

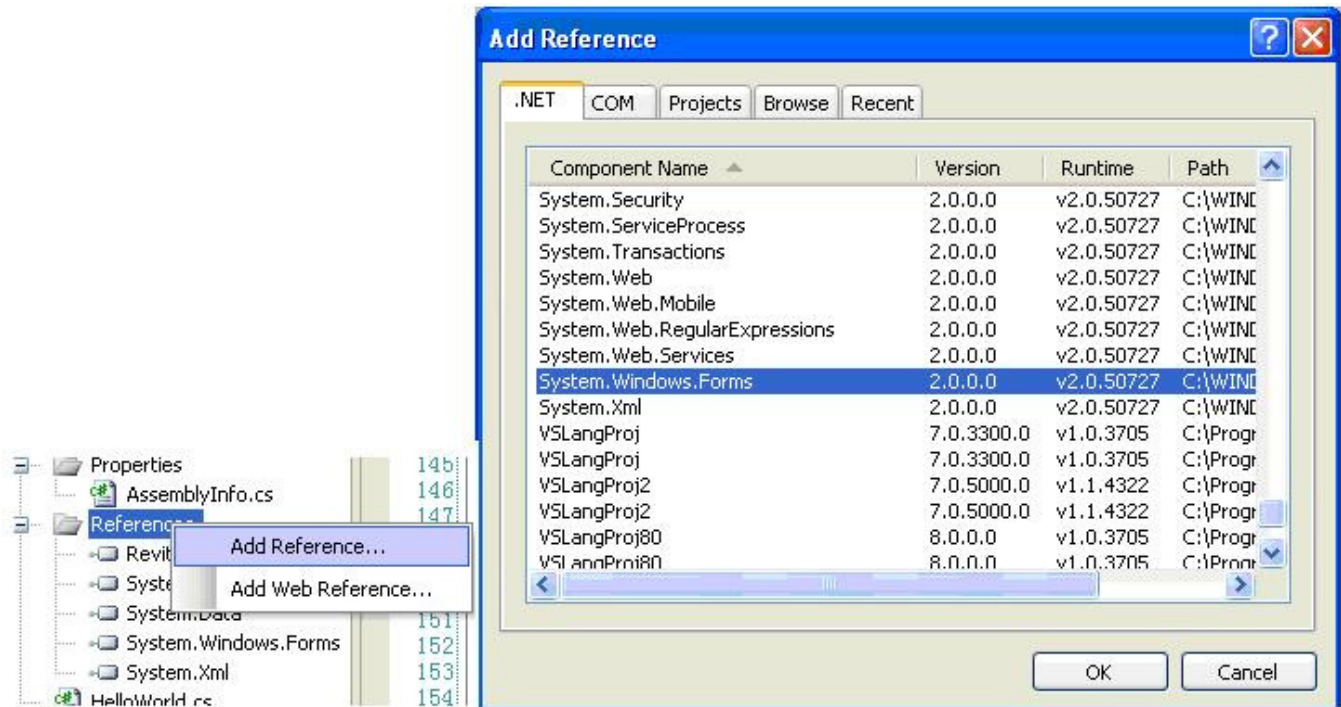


Figure 2: Add Reference

2.2.3 Add Code

Add the following code to create the add-in:

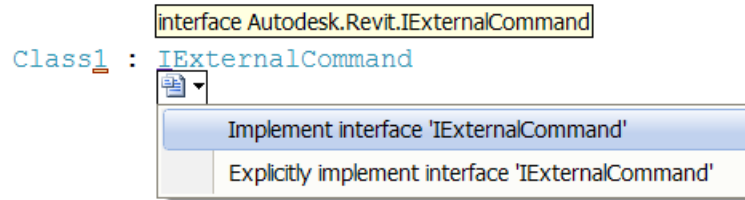
Code Region 2-1: Getting Started

```
using System;
using System.Windows.Forms;

using Autodesk.Revit;
namespace HelloWorld
{
    public class Class1:IExternalCommand
    {
        public IExternalCommand.Result Execute(ExternalCommandData revit,
            ref string message,ElementSet elements)
        {
            MessageBox.Show("Hello World");
            return IExternalCommand.Result.Succeeded;
        }
    }
}
```

Note: To use the MessageBox class, you must use the System.Windows.Forms namespace after you add the reference.

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



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

2.2.4 Build the Program

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

2.2.5 Modify the Revit.ini file

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

1. To edit the Revit.ini file, open it for edit in Notepad. The Revit.ini file is usually located in the Revit installation directory on your computer. For example: C:\Program Files\Autodesk Revit Architecture 2010\Program.
2. Add the following to the end of the existing code:

Code Region 2-2: Modifying the Revit.ini

```
[ExternalCommands]
ECCount=1
ECClassName1= HelloWorld.Class1
ECAssembly1= D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll
ECName1= HelloWorld
ECDescription1=Implementation of HelloWorld within Autodesk Revit
```

Note: ECAssembly1 is the path to the assembly, D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll in this particular example.

Refer to the [Add-in Integration](#) chapter for more details about the Revit.ini file.

2.2.6 Debug the Add-in

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

1. In the Solution Explorer window, right-click the HelloWorld project to display a context menu.
2. From the context menu, click Properties. The Properties window appears.
3. Click the Debug tab.
4. Under the Start Action section, click Start external program and browse to the Revit.exe file. By default, the file is located at the following path, C:\Program Files\Autodesk Revit Structure 2010\Program\Revit.exe.

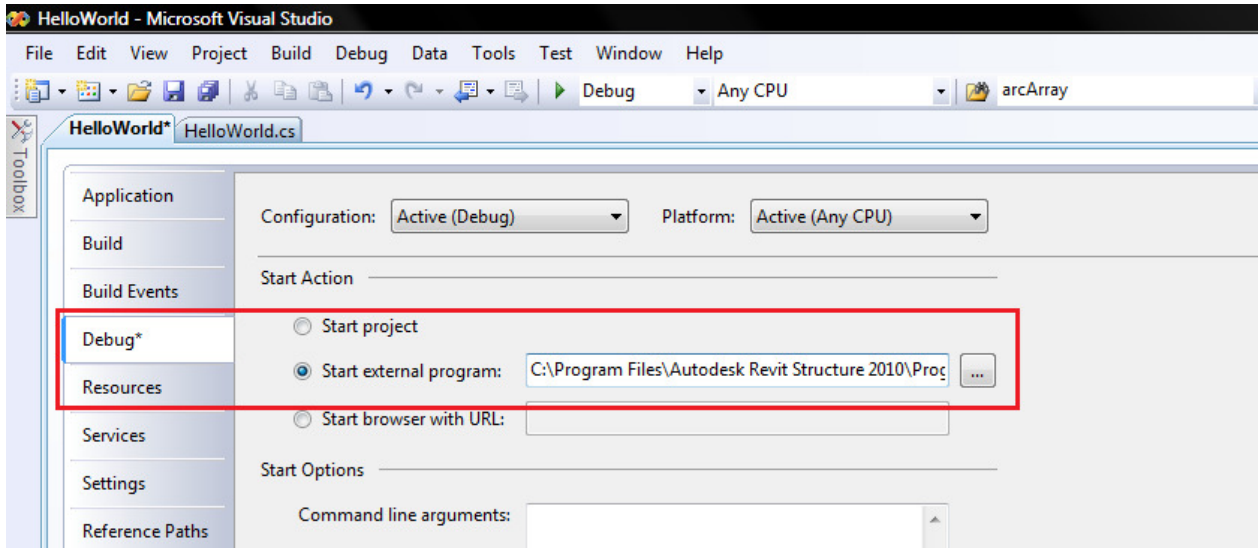


Figure 3: Set debug environment

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

```
MessageBox.Show("Hello World");
```

6. Press F5 to start the debug procedure.

Test debugging:

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

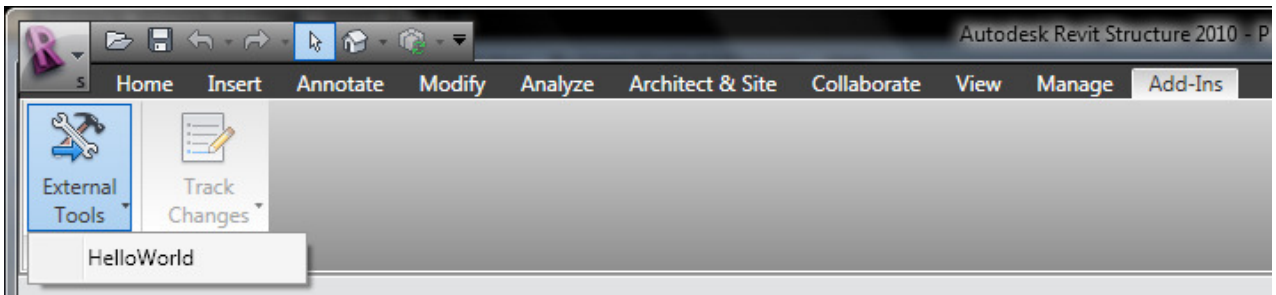


Figure 4: HelloWorld External Tools command

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



Figure 5: System message

2.2.7 Troubleshooting

Q: My add-in application will not compile.

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

Q: Why isn't my add-in application displayed under External Tools?

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

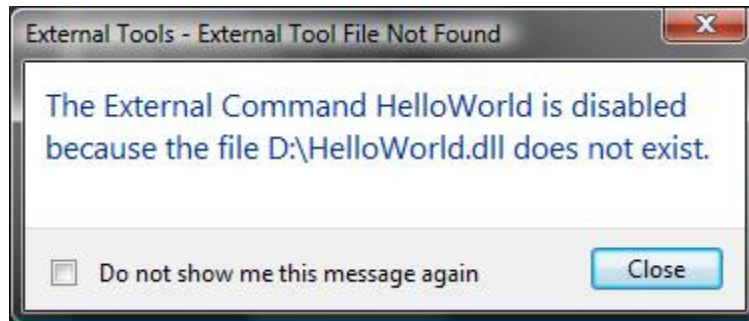


Figure 6: External Tools Error Message

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

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

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

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

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

For example:

Code Region 2-3: Exceptions in Execute()

```
Command: IExternalCommand
{
```

```

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

```

The following two exceptions clearly identify the problem:

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

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

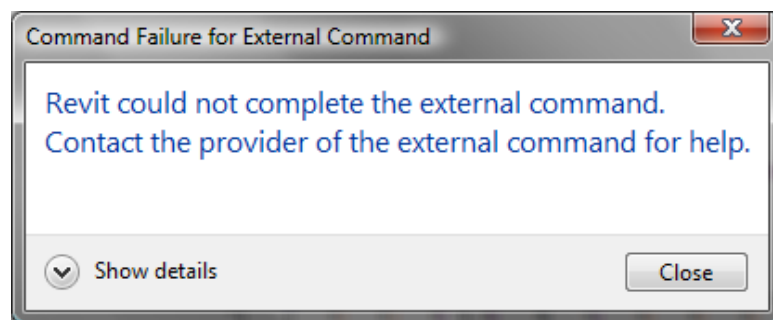


Figure 7: Unhandled exception in External Command

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's an example:

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

```

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

    try
    {
        // Do some stuff
    }

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

    return IExternalCommand.Result.Succeeded;
}

```

2.3 Walkthrough: Add Hello World Ribbon Panel

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

2.3.1 Create a New Project

Complete the following steps to create a new project:

1. Create a C# project in Visual Studio using the Class Library template.
2. Type AddPanel as the project name.
3. Add the RevitAPI reference using the directions in the previous walkthrough, Walkthrough: Hello World.
4. Add the PresentationCore and WindowsBase references following similar steps as for adding the System.Windows.Forms reference in the previous walkthrough.

2.3.2 Change the Class Name

To change the class name, complete the following steps:

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

2.3.3 Add Code

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

Add the following code for the ribbon panel:

Code Region 2-5: Adding a ribbon panel

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

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

        PushButton pushButton = ribbonPanel.AddPushButton("HelloWorld", "Hello World",
            @"D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll", "HelloWorld.Class1");

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

```

        BitmapImage largeImage = new BitmapImage(uriImage);
        pushButton.LargeImage = largeImage;

        return IExternalApplication.Result.Succeeded;
    }

    public IExternalApplication.Result OnShutdown(ControlledApplication application)
    {
        return IExternalApplication.Result.Succeeded;
    }
}

```

2.3.4 Build the Application

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

2.3.5 Modify the Revit.ini File

To invoke the application in Revit, modify the Revit.ini file to register it into Revit.

1. Open the Revit.ini file for edit using Notepad. The .ini file is usually in the Revit installation directory on your computer. For example, it is usually located at the following path
C:\Program Files\Autodesk Revit Architecture 2010\Program.
2. Add the following information to the end of the file:

Code Region 2-6: Adding an external application to Revit.ini

```

[ExternalApplications]
EACount = 1
EAClassName1 = AddPanel.CsAddPanel
EAAssembly1 = D:\Sample\AddPanel\AddPanel\bin\Debug\AddPanel.dll

```

Note: The AddPanel.dll file is in the default file folder in a new folder called Debug (D:\Sample\HelloWorld\bin\Debug\AddPanel.dll). Use the file path to evaluate ECAsssembly1.

Refer to the [Add-in Integration](#) chapter for more information about the Revit.ini file.

2.3.6 Debugging

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

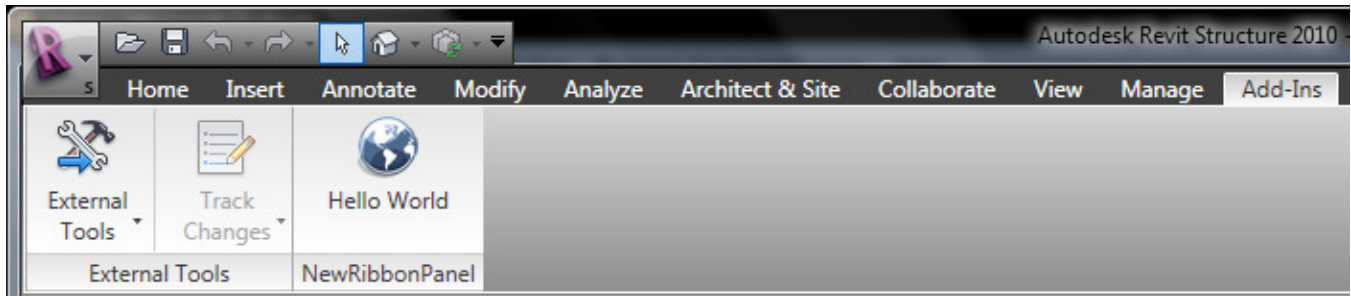


Figure 8: Add a new ribbon panel to Revit

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



Figure 9: Hello World dialog box

2.4 Walkthrough: Retrieve Selected Elements

Although the Hello World project is a very simple application, it gives you a good idea of how you can add new applications. 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 `Application.ActiveDocument.Selection.Elements` property to retrieve the selected object.

The selected object is a Revit `elementSet`. You can get all of the information about the selected object from the `elementSet`. Use the `IEnumerator` interface or `foreach` loop to search the `elementSet`.

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

Code Region 2-7: Retrieving selected elements

```
public class Document_Selection : IExternalCommand
{
    public IExternalCommand.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.
            Document document = commandData.Application.ActiveDocument;
```

```

        // Get the element selection of current document.
        Selection selection = document.Selection;
        ElementSet collection = selection.Elements;

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

            MessageBox.Show(info, "Revit", MessageBoxButtons.OK);
        }
    }
    catch (Exception e)
    {
        message = e.Message;
        return IExternalCommand.Result.Failed;
    }

    return IExternalCommand.Result.Succeeded;
}
}

```

After you get the selected elements, you can get the properties or parameters for the elements. For more information, see the [Parameter](#) chapter.

2.5 Walkthrough: Retrieve All Elements

Similar to retrieving selected elements, you can quickly get all project objects from the active document using the `Application.ActiveDocument.Elements` property. To query one or more elements, you can iterate through all elements and test each one.

Note: This is seldom a good approach, as it can take a long time to iterate through all the elements in a document. It is much faster to iterate through a filtered list of elements; see the next walkthrough for an example.

Code Region 2-8: Retrieving all elements

```

// Get all the walls in current document.
System.Collections.ArrayList walls = new System.Collections.ArrayList();

ElementIterator iterator = document.Elements;
iterator.Reset();

```

```

while (iterator.MoveNext())
{
    if (iterator.Current is Autodesk.Revit.Elements.Wall)
    {
        walls.Add(iterator.Current);
    }
}

// Format the prompt string
String prompt = null;
if (0 == walls.Count)    // if there are not any wall
{
    prompt = "No walls in current document.";
}
else
{
    prompt = "The ids of the walls in the current document are:";
    foreach (Autodesk.Revit.Elements.Wall wall in walls)
    {
        // output the id of the walls
        prompt += "\n\t" + wall.Id.Value;
    }
}

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

```

2.6 Walkthrough: Retrieve Filtered Elements

You can use a filter to select only elements that meet certain criteria, which has better performance than iterating through all the elements in a document. For more information on creating and using element filters, see [Iterating the Elements Collection](#).

This example iterates through all the elements of type Level in a document, and displays a dialog listing the names of the levels.

Code Region 2-9: Retrieve filtered elements

```

// Create a Filter to get all the doors in the document
Autodesk.Revit.Creation.Filter filterCreator =
    document.Application.Create.Filter;
TypeFilter familyInstanceFilter =
    filterCreator.NewTypeFilter(typeof(FamilyInstance), true);
CategoryFilter doorsCategoryfilter =
    filterCreator.NewCategoryFilter(BuiltInCategory.OST_Doors);
Filter doorInstancesFilter =
    filterCreator.NewLogicAndFilter(familyInstanceFilter, doorsCategoryfilter);

// Apply the filter to the elements in the active document
ElementIterator iterator = document.get_Elements(doorInstancesFilter);

```

```
String prompt = "The ids of the doors in the current document are:";
iterator.Reset();
while (iterator.MoveNext())
{
    FamilyInstance door = iterator.Current as FamilyInstance;
    // output the id of the doors
    prompt += "\n\t" + door.Id.Value;
}

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


3 Add-In Integration

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

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

This chapter focuses on the following:

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

3.1 Overview

The Revit Platform API is set up based on Revit application functionality. Revit is not dependent on the API; RevitAPI is a class Library that only works when Revit is running. With the powerful API, you can add API based add-ins to extend Revit with the types and methods in the RevitAPI.

As the following picture shows, Revit Architecture, Revit Structure, and Revit MEP are specific to Architecture, Structure, and MEP respectively. In addition, Architecture also has other more widely used code, such as families.

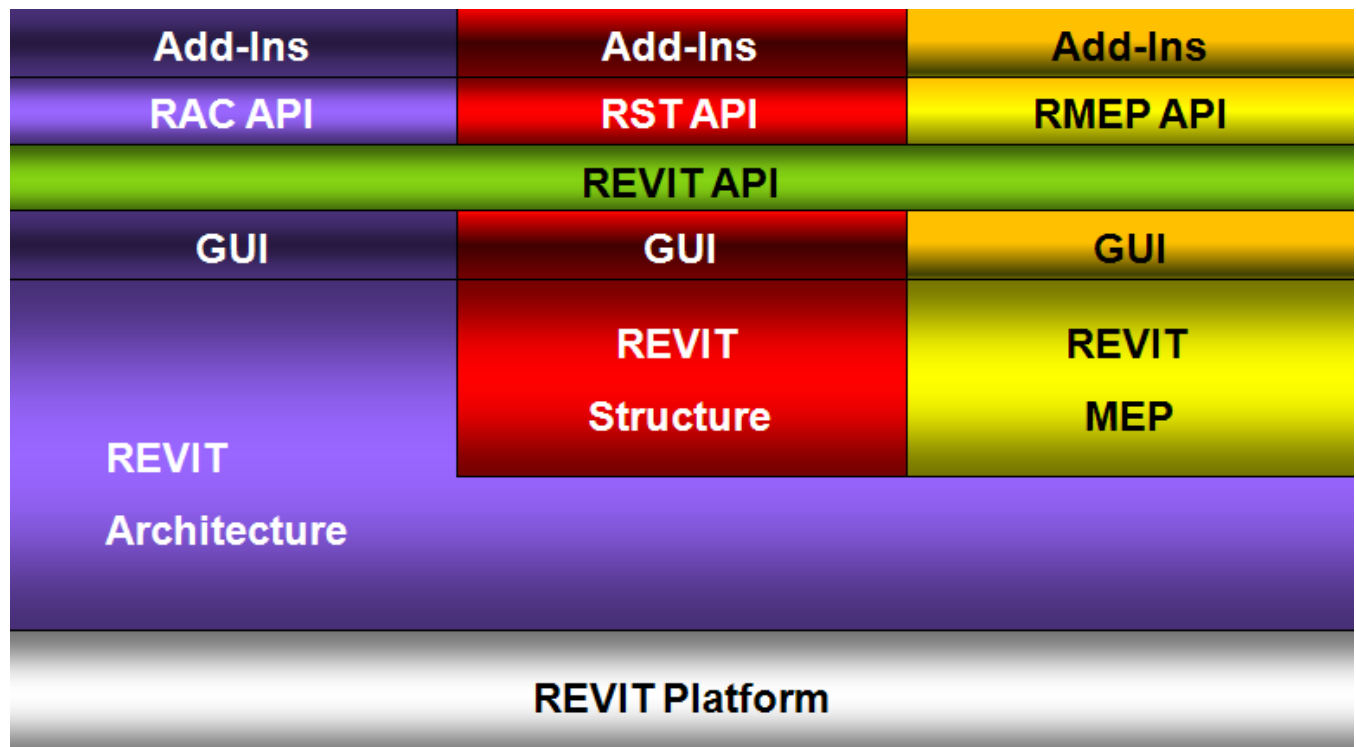


Figure 10: Revit, RevitAPI and Add-ins

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

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

3.2 External Commands

Developers can add functionality by implementing External Commands which appear in the External Tools menu-button. Revit finds external commands listed in the `Revit.ini` file.

3.2.1 Loading and Running External Commands

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

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

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

- Revit reads `Revit.ini` 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.

Note: You can load and run external commands using the Add-in Manager utility provided with the Revit Platform SDK instead of manually editing the `Revit.ini` file. See 3.5 Using the Add-in Manager for more information.

3.2.2 `IExternalCommand`

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

The `Execute()` method has three parameters:

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

3.2.2.1 `commandData` (`ExternalCommandData`)

The `ExternalCommandData` object contains references to `Application` and `View` which is 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.ActiveDocument;
```

The following table illustrates the `ExternalCommandData` public properties

Property	Description
Application (Autodesk.Revit.Application)	Retrieves an object that represents the current Application for external command.
Data (Autodesk.Revit.Collections.StringStringMap)	A data map that can be used to read and write data to the Revit journal file.
IsReadOnly (bool)	Informs the developer of the read and write capabilities of the object. (Inherited from APIObject.)
View (Autodesk.Revit.Elements.View)	Retrieves an object that represents the View external commands work on.

Table 1: ExternalCommandData public properties**3.2.2.2 message (String):**

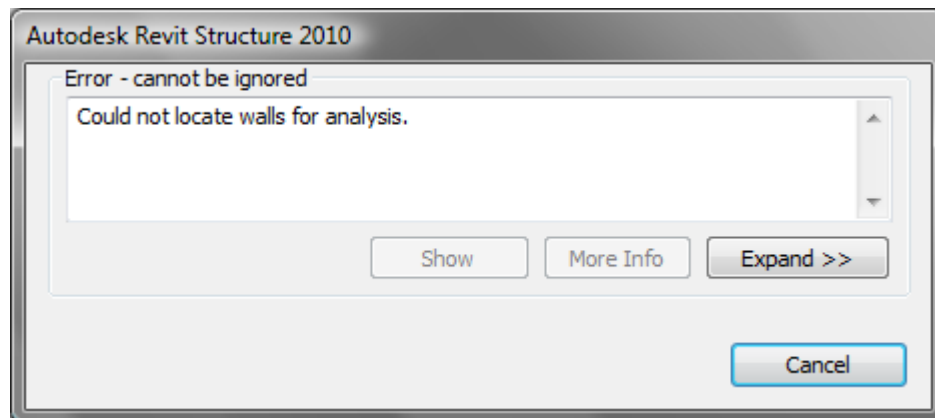
Error messages are returned by an external command using the "ref" parameter message. The string-type parameter is set in the external command process. When `IFunctionResult.Failed` or `IFunctionResult.Cancelled` is returned, and the message parameter is set, an error dialog appears.

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

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

**Figure 11: Error message dialog box**

3.2.2.3 elements (ElementSet):

Whenever `IExternalCommand.Result.Failed` or `IExternalCommand.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 Autodesk.Revit.IExternalCommand.Result Execute(
        Autodesk.Revit.ExternalCommandData commandData, ref string message,
        Autodesk.Revit.ElementSet elements)
    {
        message = "Please note the highlighted Walls.";
        ElementIterator elemItor =
            commandData.Application.ActiveDocument.get_Elements(typeof(Wall));

        elemItor.Reset();
        while (elemItor.MoveNext())
        {
            Autodesk.Revit.Element elem = elemItor.Current as Autodesk.Revit.Element;
            elements.Insert(elem);
        }

        return Autodesk.Revit.IExternalCommand.Result.Failed;
    }
}
```

The following picture displays the result of the previous code.

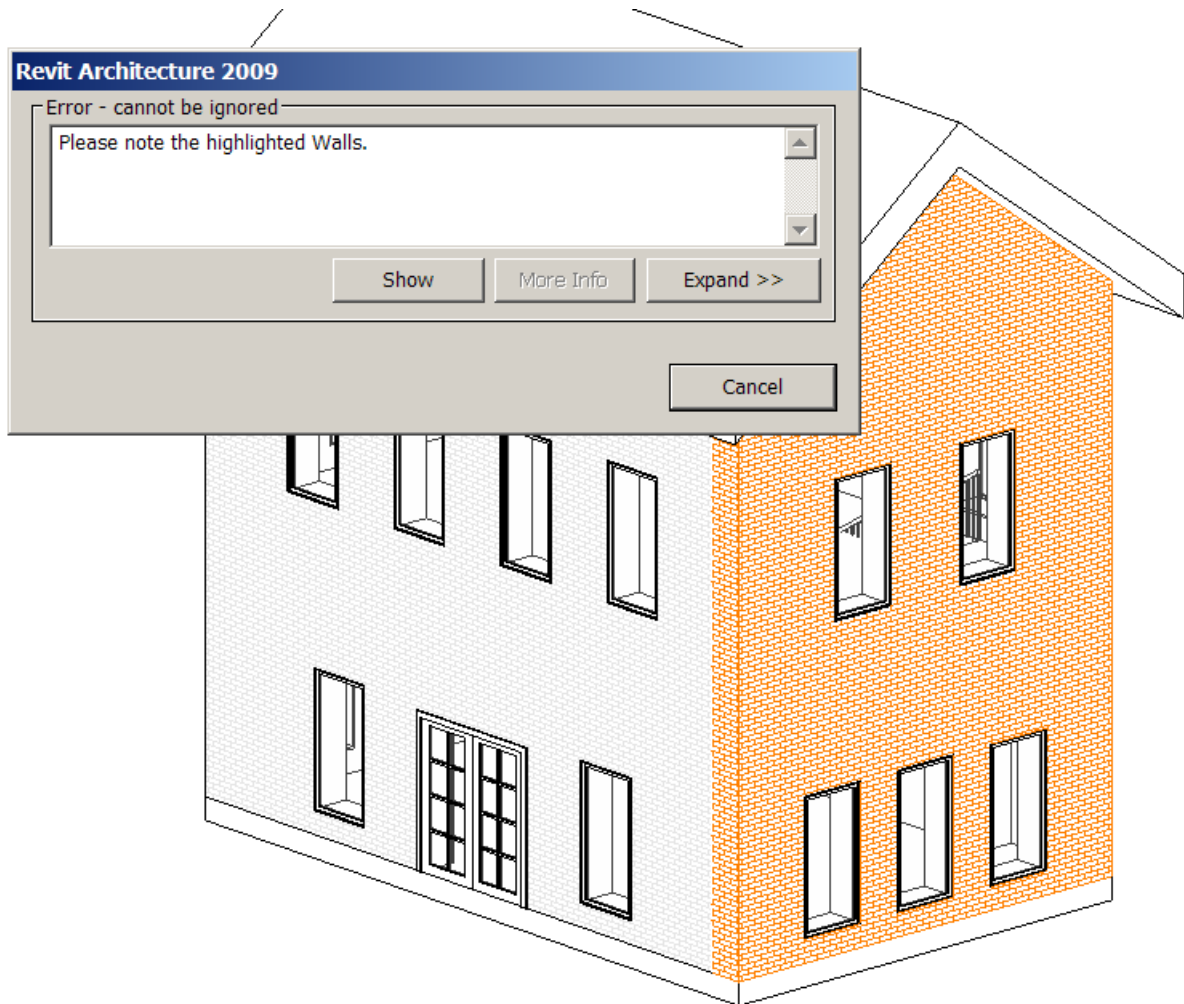


Figure 12: Error message dialog box and highlighted elements

3.2.2.4 Return

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

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

Table 2: IEExternalCommand.Result

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

Code Region 3-4: Prompting the user

```

public IExternalCommand.Result Execute(ExternalCommandData commandData,
    ref string message, ElementSet elements)
{
    try
    {
        Document doc = commandData.Application.ActiveDocument;

        // Delete selected elements
        ElementIdSet ids = doc.Delete(doc.Selection.Elements);

        DialogResult result = MessageBox.Show(
            "Click Yes to return Succeeded. Selected members will be deleted.\n" +
            "Click No to return Failed. Selected members will not be deleted.\n" +
            "Click Cancel to return Cancelled. Selected members will not be deleted.",
            "Revit", MessageBoxButtons.YesNoCancel);

        if (DialogResult.Yes == result)
        {
            return IExternalCommand.Result.Succeeded;
        }
        else if (DialogResult.No == result)
        {
            elements = doc.Selection.Elements;
            message = "Failed to delete selection.";
            return IExternalCommand.Result.Failed;
        }
        else
        {
            return IExternalCommand.Result.Cancelled;
        }
    }
    catch
    {
        message = "Unexpected Exception thrown.";
        return IExternalCommand.Result.Failed;
    }
}

```

3.2.3 Revit.ini [ExternalCommands] Section

In the Revit.ini [ExternalCommands] section, you can specify external tools that are loaded on start-up. There can only be one [ExternalCommands] in the revit.ini file.

Example:

Code Region 3-5: Revit.ini ExternalCommands Section

```
[ExternalCommands]
ECCount=1
ECClassName1=Project1.Class1
ECAssembly1=C:\Project1\Project1.dll
ECName1="My Tool"
ECDescription1="Implementation of My Tool within Revit"
```

The following table describes the entries in the Revit.ini section:

- **ECCount** - Set the ECCount property to the total number of external commands available.
- **ECClassName#** - The ECClassName property is used to provide the name of the class that implements the IExternalCommand interface.
- The command object is the full class name (with the namespace), such as MyNamespace.MyClass.
- The # is the command number, such as ECClassName1.
- **ECAssembly#** - The ECAssembly property is used for .NET based objects only.
- This property identifies the path to the Assembly containing the .NET command object.
- The path can be relative to the Revit installation directory or fully qualified.
- The # identifies the command number, such as ECAssembly1. The command number must match the command number used for the ECClassName property.
- In the previous example, the fully qualified path for the assembly is C:\Project1\Project1.dll.
- **ECName#** - The ECName property sets a short name for the external command in the External Tools menu-button. The # identifies the command number, such as ECName1. The command number must match the command number used for the ECClassName property. In the previous example, the menu item for command #1 is "My Tool".
- **ECDescription#** - The ECDescription property is an optional string that appears in the Revit status bar when the mouse moves over the menu item. The help string is a one-line description of the external command. In the previous example, the message "Implementation of My Tool within Revit" appears in the status bar when the mouse moves over the My Tool menu item.

Note: The Revit.ini file is read when Revit starts. It is not read at any other point during application execution.

3.3 External Application

Developers can add functionality through External Applications as well as External Commands. Revit recognizes external applications listed in the Revit.ini. Ribbon panels are customized using the External Application. Ribbon panel buttons are bound to an External command.

3.3.1 IExternalApplication

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

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

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

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

```
public interface IExternalApplication
{
    IExternalApplication.Result OnShutdown(ControlledApplication application);
    IExternalApplication.Result OnStartup(ControlledApplication application);
}
```

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

Code Region 3-7: DialogBoxShowing Event

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

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

Code Region 3-8: Using ControlledApplication

```
public class Application_DialogBoxShowing : IExternalApplication
{
    public IExternalApplication.Result OnStartup(ControlledApplication application)
    {
        // Register events
        application.DialogBoxShowing += new
            EventHandler<Autodesk.Revit.Events.DialogBoxShowingEventArgs>(AppDialogShowing);
        return IExternalApplication.Result.Succeeded;
    }

    public IExternalApplication.Result OnShutdown(ControlledApplication application)
    {
        // unregister events
        application.DialogBoxShowing -= new
            EventHandler<Autodesk.Revit.Events.DialogBoxShowingEventArgs>(AppDialogShowing);
        return IExternalApplication.Result.Succeeded;
    }

    void AppDialogShowing(object sender,
        Autodesk.Revit.Events.DialogBoxShowingEventArgs args)
    {
        // Get the help id of the showing dialog
        int dialogId = args.HelpId;

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



```

// Show the prompt message, and allow the user to close the dialog directly.
DialogResult result = MessageBox.Show(promptInfo, "Revit",
                                     MessageBoxButtons.OKCancel);

if (DialogResult.Cancel == result)
{
    // Do not show the revit dialog
    args.OverrideResult(1);
}
else
{
    // Continue to show the revit dialog
    args.OverrideResult(0);
}
}
}

```

3.3.2 Revit.ini [ExternalApplications] Section

In the Revit.ini, you list external applications in the [ExternalApplications] section.

Note: You can load and run external applications using the Add-in Manager utility provided with the Revit Platform SDK instead of manually editing the Revit.ini file. See 3.5 Using the Add-in Manager for more information.

The following code example illustrates how to specify an external application:

Code Region 3-9: Revit.ini ExternalApplications section

```

[ExternalApplications]
EACount=1
EAClassName1=EA1.Class1
EAAssembly1=D:\EA1\bin\Debug\EA1.dll

```

- EACount - Set the EACount property to the total number of external applications available.
- EAClassName# - The EAClassName property is used to provide the application object name that supports the IExternalApplication interface described earlier in this chapter.
- For .NET objects it is the full class name (with the namespace), such as MyNamespace.MyClass.
- The # identifies the application number such as EAClassName1.
- EAAssembly - The EAAssembly property is used for .NET based objects only.
- This property identifies the Assembly path containing the .NET application object.
- The path can be relative to the Revit installation directory or fully qualified.
- The # identifies the application number such as EAAssembly1. The application number must match the application number used for the EAClassName property.
- In the previous example, the Assembly has the fully qualified path D:\EA1\bin\Debug\EA1.dll.

3.4 Ribbon Panels and Controls

Revit provides API solutions to integrate custom ribbon panels and controls. These APIs are used with `IExternalApplication`. All custom ribbon panels will appear on the Add-Ins tab in Revit. Panels can include buttons, both large and small, which can be either simple push buttons or drop-down buttons containing multiple commands. Panels can also include vertical separators to help separate commands into logical groups.

Please see the [API Ribbon Layout Guidelines](#) for information on developing a user interface that is compliant with the standards used by Autodesk.

3.4.1 Create a New Ribbon Panel and Controls

The following sample illustrates how you can create ribbon panels and various ribbon panel controls. The following sections describe these controls in more detail.

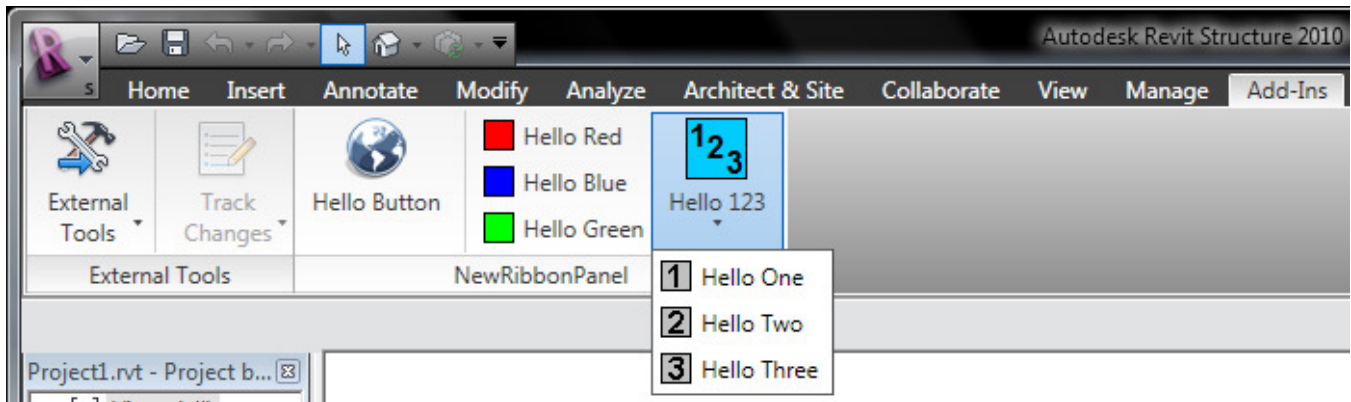


Figure 13: New ribbon panel and controls

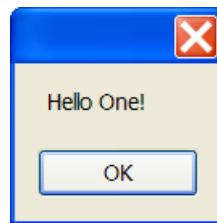


Figure 14: Dialog triggered by Hello One command

The following code demonstrates the ribbon panel and controls in the previous figures. The external command location is `D:\Sample\HelloWorld\bin\Debug\Hello.dll` in an assembly containing the External Command Types:

- `Hello.HelloButton`
- `Hello.HelloOne`
- `Hello.HelloTwo`
- `Hello.HelloThree`
- `Hello.HelloRed`
- `Hello.HelloBlue`
- `Hello.HelloGreen`

Code Region 3-10: Ribbon panel and controls

```
public IExternalApplication.Result OnStartup(ControlledApplication application)
{
```

```

// add new ribbon panel
RibbonPanel ribbonPanel = application.CreateRibbonPanel("NewRibbonPanel");

//Create a push button in the ribbon panel "NewRibbonPanel"
string assembly = @"D:\Sample\HelloWorld\bin\Debug\Hello.dll";
PushButton pushButton = ribbonPanel.AddPushButton("Hello Button",
                                                    "Hello Button", assembly, "Hello.HelloButton");

// create bitmap image for button
Uri uriImage = new Uri(@"D:\Sample\HelloWorld\bin\Debug\39-Globe_32x32.png");
BitmapImage largeImage = new BitmapImage(uriImage);

// assign bitmap to button
pushButton.LargeImage = largeImage;

// assign a small bitmap to button which is used if command
// is moved to Quick Access Toolbar
Uri uriSmallImage =
    new Uri(@"D:\Sample\HelloWorld\bin\Debug\39-Globe_16x16.png");
BitmapImage smallImage = new BitmapImage(uriSmallImage);

// assign small image to button
pushButton.Image = smallImage;

// add a vertical separator bar to the panel
ribbonPanel.AddSeparator();

// define 3 new buttons to be added as stacked buttons
PushButtonData buttonRed = new PushButtonData("Hello Red", "Hello Red",
                                                assembly, "Hello.HelloRed");
buttonRed.Image =
    new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Red.bmp"));
PushButtonData buttonBlue = new PushButtonData("Hello Blue", "Hello Blue",
                                                assembly, "Hello.HelloBlue");
buttonBlue.Image =
    new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Blue.bmp"));
PushButtonData buttonGreen = new PushButtonData("Hello Green", "Hello Green",
                                                assembly, "Hello.HelloGreen");
buttonGreen.Image =
    new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Green.bmp"));

// add 3 stacked buttons to the panel
ribbonPanel.AddStackedButtons(buttonRed, buttonBlue, buttonGreen);

// add a pull-down button to the panel
PulldownButton pulldownButton =
    ribbonPanel.AddPulldownButton("Hello", "Hello 123");
pulldownButton.LargeImage =

```

```

        new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Hello.bmp"));

// add some menu items to the pull-down button and assign bitmaps to them
PushButton buttonOne = pulldownButton.AddItem("Hello One", assembly,
                                                "Hello.HelloOne");

buttonOne.LargeImage =
    new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\One.bmp"));
PushButton buttonTwo = pulldownButton.AddItem("Hello Two", assembly,
                                                "Hello.HelloTwo");

buttonTwo.LargeImage =
    new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Two.bmp"));
PushButton buttonThree = pulldownButton.AddItem("Hello Three", assembly,
                                                "Hello.HelloThree");

buttonThree.LargeImage =
    new BitmapImage(new Uri(@"D:\Sample\HelloWorld\bin\Debug\Three.bmp"));

return IExternalApplication.Result.Succeeded;
}

```

3.4.2 Large Buttons

There are two types of buttons you can add to a panel: simple push buttons or drop-down buttons. The Hello Button button in Figure 13 is a push button. When the button is pressed, the corresponding command is triggered.

Drop-down buttons expand to display two or more commands in a drop-down menu. The Hello 123 button in Figure 13 is a drop-down button. In the Revit API, drop-down buttons are referred to as `PulldownButtons`.

You can associate an image with a large button using the `LargeImage` property. The best size is 32x32 pixels, but larger images will be adjusted to fit the button. Large buttons should also have a 16x16 pixel image set for the `Image` property. This image is used if the command is moved to the Quick Access Toolbar. If the `Image` property is not set, no image will be displayed if the command is moved to the Quick Access Toolbar. Note that if an image larger than 16x16 pixels is used, it will NOT be adjusted to fit the toolbar.

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

3.4.3 Stacked Buttons

For the sake of panel size, you can also add small buttons in stacks of two or three. Each button in the stack can be either a push button or drop-down button depending on how the buttons are defined. Stacked buttons should have an image associated through their `Image` property, rather than `LargeImage`. A 16x16 image is ideal for small stacked buttons.

3.5 Using the Add-in Manager

The Revit Platform SDK includes a utility called the Add-in Manager, which makes it possible to load and run external commands and applications without having to edit the `Revit.ini` file. The installer for this utility is located in the SDK, in the Add-in Manager directory. Read the Add-In Manager Read Me.doc file, located in this directory, for more information about installing and using the Add-in Manager.

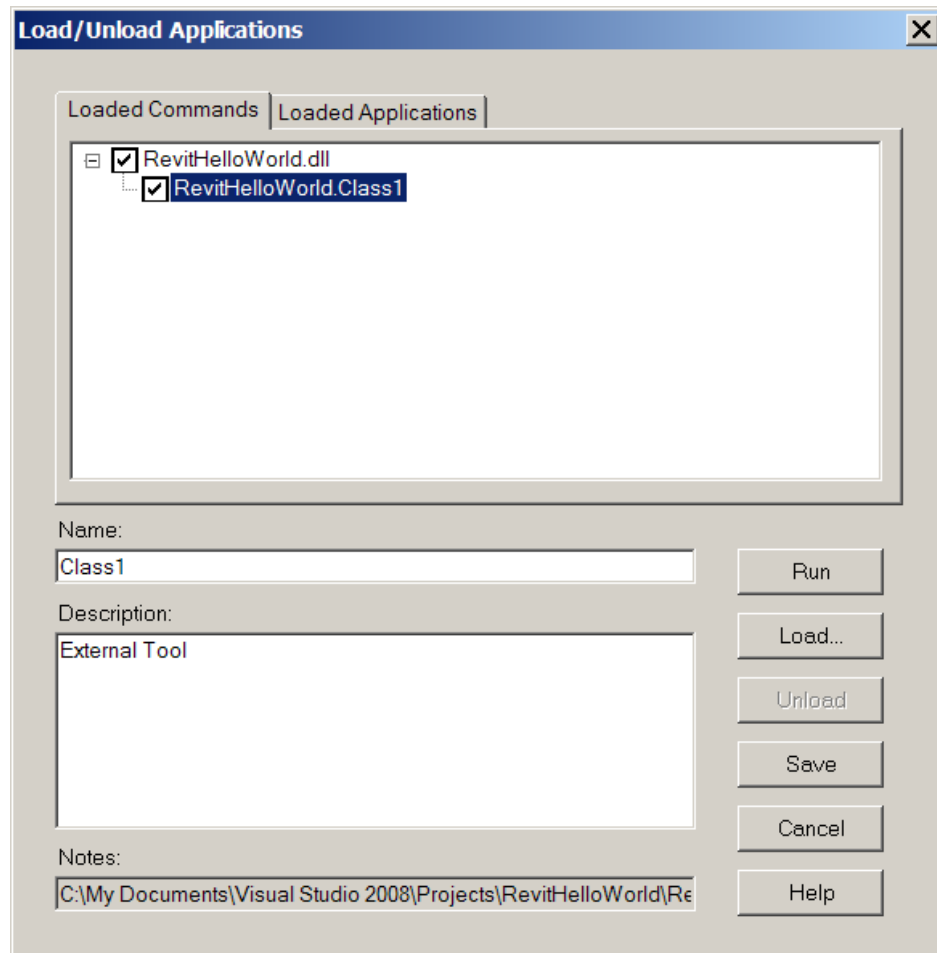


Figure 15: The Add-in Manager

4 Application and Document

The Revit Platform API uses a Multi-Document model similar to Microsoft Office. The top level classes in the Revit Platform API are Application and Document.

- The Application object refers to an individual Revit session, providing access to documents, options, and other application-wide data and settings.
- 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.
- The active, or top, view is the active project and the active document.

The relationship between the classes Application, Document, Element, and View are illustrated in the following diagram:

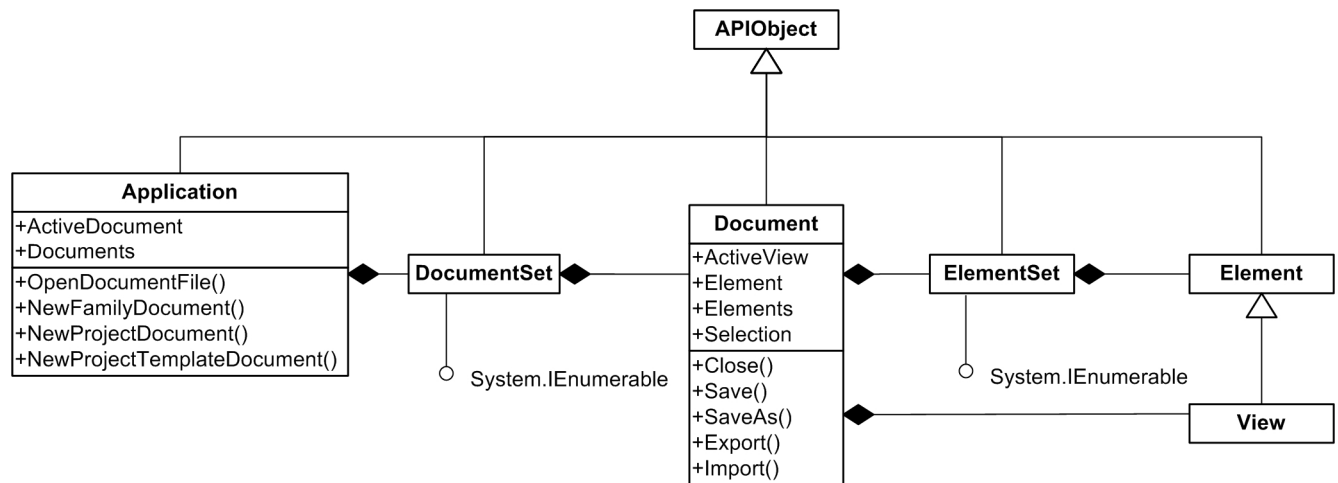


Figure 16: Application-Document-View-Element diagram

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

4.1 Application Functions

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

4.1.1 Application Version Information

Application object properties include VersionBuild, VersionNumber and VersionName. Each property returns Revit version information.

4.1.2 Application-wide Settings

The Options property provides configuration functions corresponding to some of the commands on the Manage tab in Revit. For more details, see the [Settings](#) section in this chapter.

4.1.3 Document Management

The Application class provides methods to open and create a document. For more details, see the [Document and Management](#) section in this chapter. In addition, the following characteristics apply:

- The active document is identified using the ActiveDocument property.

- Retrieve all open documents using the Documents property.

4.1.4 Shared Parameter Management

Revit uses one shared parameter file at a time. The `Application.OpenSharedParameterFile()` method enables access to the shared parameter file whose path is set in the Options object. For more details, see the [Shared Parameter](#) chapter.

4.1.5 Ribbon Panel Utility

Use the Application object to add new ribbon panels and controls to Revit. You can add the following:

- Ribbon panels
- Push buttons
- Drop-down buttons
- Separators.

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

4.1.6 Events

The Revit Platform API exposes document and dialog box events such as document open and save. Subscribing to these events notifies the application when the events are enabled and acts accordingly. For more details, see the [Access to Revit Event](#) section in the [Add-In Integration](#) chapter.

4.1.7 Create

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

4.2 Document Functions

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

4.2.1 Settings Property

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

4.2.2 Place and Locations

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

4.2.3 Type Collections

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

4.2.4 View Management

A project document can have multiple views. The `ActiveView` property returns a View object representing the active view. You can traverse all elements in the project to retrieve other views. For more details, see the [Views](#) chapter.

4.2.5 Element Retrieval

The Document object stores elements in the project. Some properties enable you to retrieve elements from the Revit database.

- Retrieve all elements using the Elements property. This property provides access to all elements in the Document. It returns an iterator that traverses the entire element database and enables iteration for all elements.
- Retrieve selected elements using the Selection property. This property returns an object representing the active selection containing the selected project elements. It also provides an add-in UI interaction function Selection.PickOne to pick elements.
- Retrieve certain elements by ElementId or UniqueId using the Element property.

For more details, see the [Elements Essentials](#) chapter.

4.2.6 File Management

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

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

For more details, see the [Document and File management](#) section in this chapter.

4.2.7 Element Management

Revit maintains all Element objects in a project.

- The Create property returns an Object Factory used to create new project element instances in the Revit Platform API, such as FamilyInstance or Group.
- Use the Delete() method to delete an element in the project.
- Deleted elements are not displayed in any views and are removed from the Document and any dependent elements.
- References to deleted elements are invalid and cause an exception.
- For more details, see the [Editing Element](#) chapter.
- Document performs common operations on elements in the document such as
 - Move
 - Rotate
 - Mirror
 - Array.
- For more details, see the [Editing Element](#) chapter.

4.2.8 Transactions

Every external command is inside a transaction. A transaction combines several operations into one atomic operation. If an external command is successful, that is if it returns Result.Succeeded, the transaction will commit all the changes and close itself. The operation is completed. If, on the other hand, the external command is not successful, the transaction will abort, which in effect will cancel all operations within. For more details, see the [Transaction](#) chapter.

4.2.8.1 Events

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

4.2.9 Others

Document also provides other functions:

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

4.3 Document and File Management

Document and file management make it easier to create and find your documents and files so that you can work in them. Using the Revit Platform API, you can load family symbols and family instances.

4.3.1 Document Retrieval

The Application class maintains all documents. As previously mentioned, you can open more than one document in a session. When you open more than one document, the active, or the top view, identifies the document you are using. The active document is retrieved using the Application class property, `ActiveDocument`.

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

4.3.2 Document File Information

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

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

4.3.3 Open a Document

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

Method	Event
<code>Document OpenDocumentFile(string filename)</code>	<code>OnDocumentOpened</code>

Table 3: Open Document in API

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

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

4.3.4 Create a Document

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

Method	Event
<code>Document NewProjectDocument(string templateFileName);</code>	<u>OnDocumentNewed</u>
<code>Document NewFamilyDocument(string templateFileName);</code>	OnDocumentNewed
<code>Document NewProjectTemplateDocument(string templateFilename);</code>	OnDocumentNewed

Table 4: Create Document in the API

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

4.3.5 Save and Close a Document

The Document class provides methods to save or close instances.

Method	Event
<code>Save()</code>	OnSave
<code>SaveAs()</code>	OnSaveAs
<code>Close()</code>	OnClose

Table 5: Save and Close Document in API

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

4.3.6 Load Family and Load Family Symbol

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

For more details, see [Loading Families](#).

4.4 Settings

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

UI command	Associated API	Reference
Project Information	Document.ProjectInformation	See the following note
Project Parameters	Document.ParameterBindings (Only for Shared Parameter)	See Shared Parameter

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

Table 6: Settings in API and UI**Note:**

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

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

Table 7: ProjectInformation

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

- Fill Patterns - Retrieve all Fill Patterns in the current document using Document.Settings.FillPatterns.
- Object Styles - Use Settings.Categories to retrieve all information in Category objects except for Line Style. For more details, see the [Categories](#) section in the [Elements Essentials](#) chapter and [Material](#) chapter.
- Phases - Revit maintains the element lifetime by phases, which are distinct time periods in the project lifecycle. All phases in a document are retrieved using the Document.Phases property. The property returns an array containing Phase class instances. However, Revit does not expose functions from the Phase class.

- Options - The Options command configures project global settings. You can retrieve an Options.Application instance using the Application.Options property. Currently, the Options.Application class only supports access to library paths and shared parameters file.
- Area and Volume Calculations – The Document.Settings.VolumeCalculationSetting allows you to enable or disable volume calculations, and to change the room boundary location.

4.5 Units

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

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

Table 8: 7 Base Units in Revit Unit System

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

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

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

Table 9: Project Unit Properties

5 Elements Essentials

An Element corresponds to a single building or drawing component, such as a door, a wall, or a dimension. In addition, an Element can be abstract, like a door type, a view, or a material definition. If a building is regarded as a system, the components are Elements.

5.1 Element Classification

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

5.1.1 Model Elements

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

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

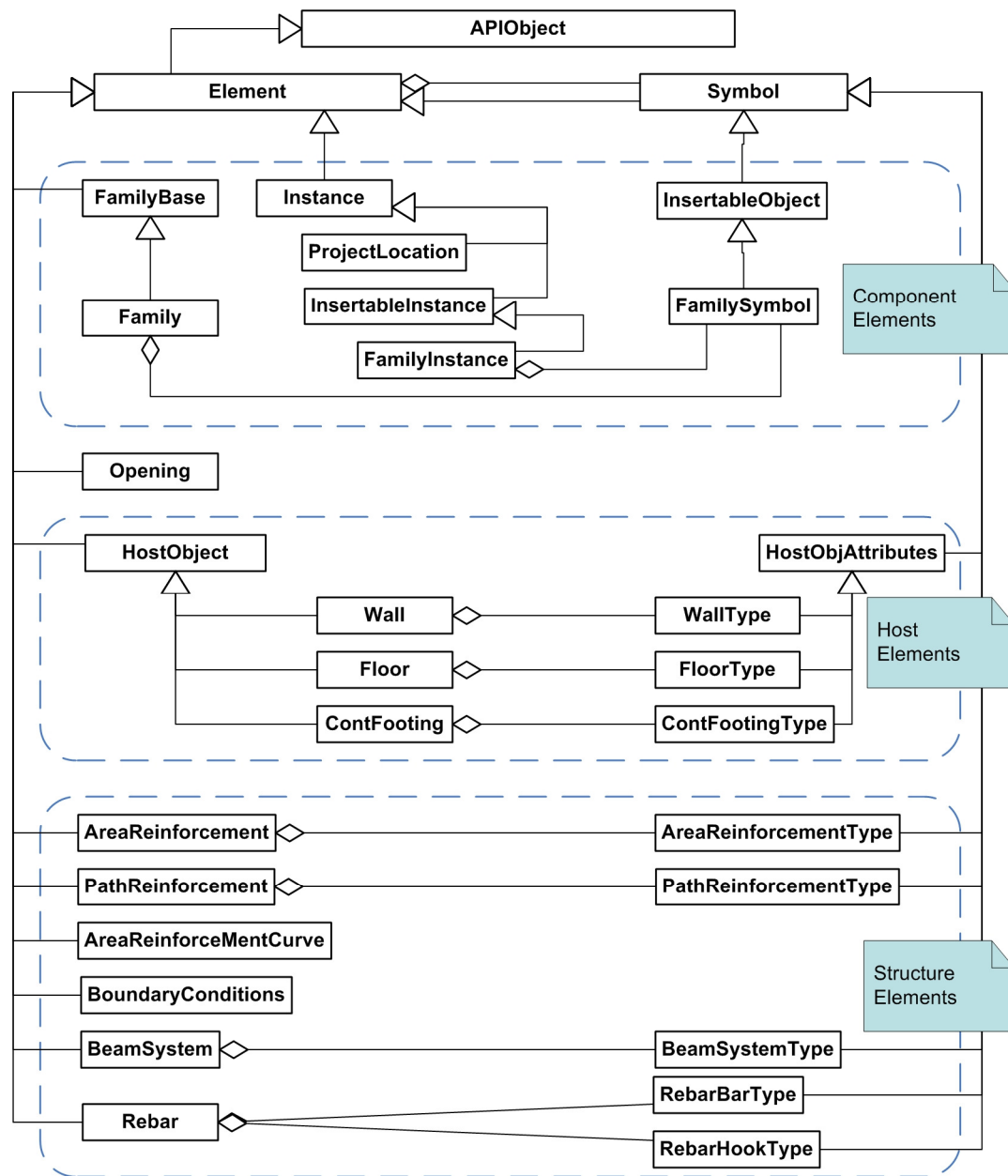


Figure 17: Model Elements diagram

5.1.2 View Elements

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

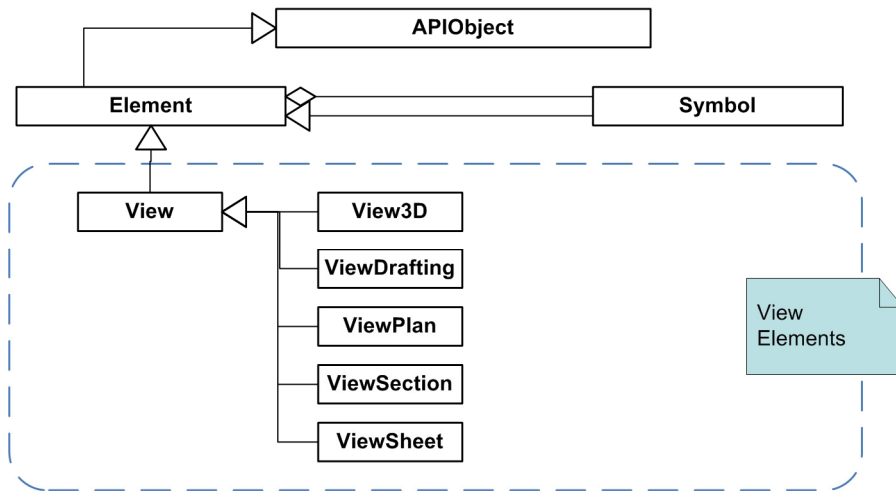


Figure 18: View Elements diagram

5.1.3 Group Elements

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

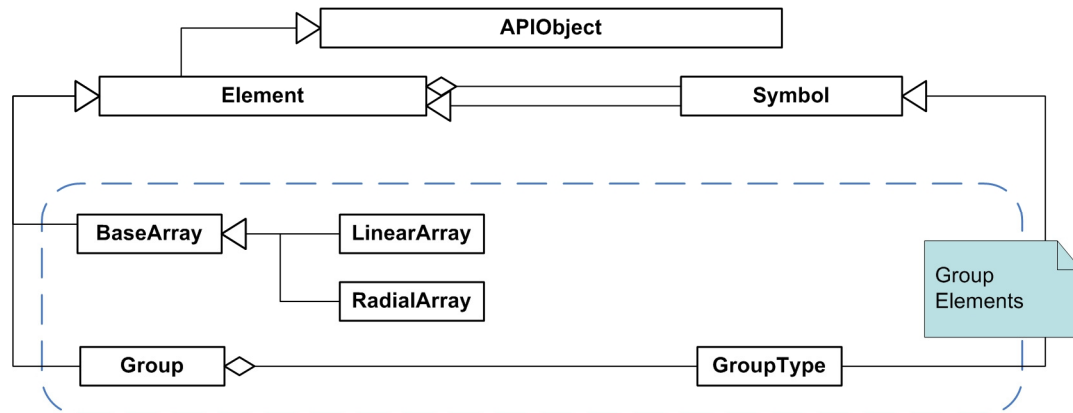


Figure 19: Group Elements UML diagram

5.1.4 Annotation and Datum Elements

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

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

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

- Datum Elements represent non-physical items that are used to establish project context. These elements can exist in views. Datum Elements are further divided into the following:
- Common Datum Elements - Common Datum Elements represent non-physical visible items used to store data for modeling.
- Datum FamilyInstance - Datum FamilyInstance represents non-physical visible items loaded into your project or created from family templates.

- **Note:** For more information about Common Datum Elements and Datum FamilyInstance, see the [Datum and Information Elements](#) chapter; for ModelCurve related contents, see the [Sketching](#) chapter.
- Structural Datum Elements - Structural Datum Elements represent non-physical visible items used to store data for structure modeling. For more information about Structural Datum Elements, see the [Revit Structure](#) chapter.

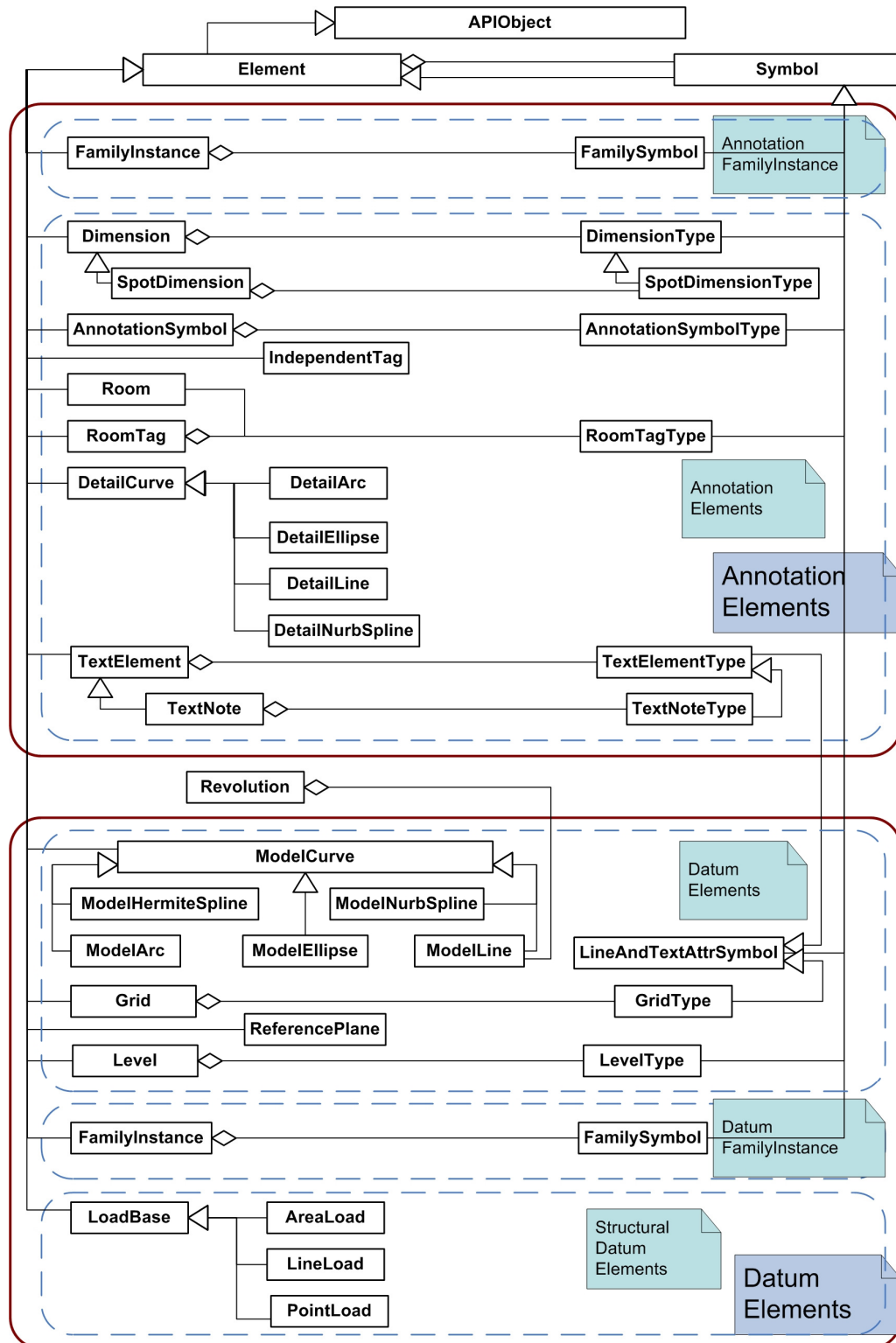


Figure 20: Annotation Elements diagram

5.1.5 Sketch Elements

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

- SketchPlane
- Sketch
- Path3D
- GenericForm.

For Sketch details, see the [Sketching](#) chapter.

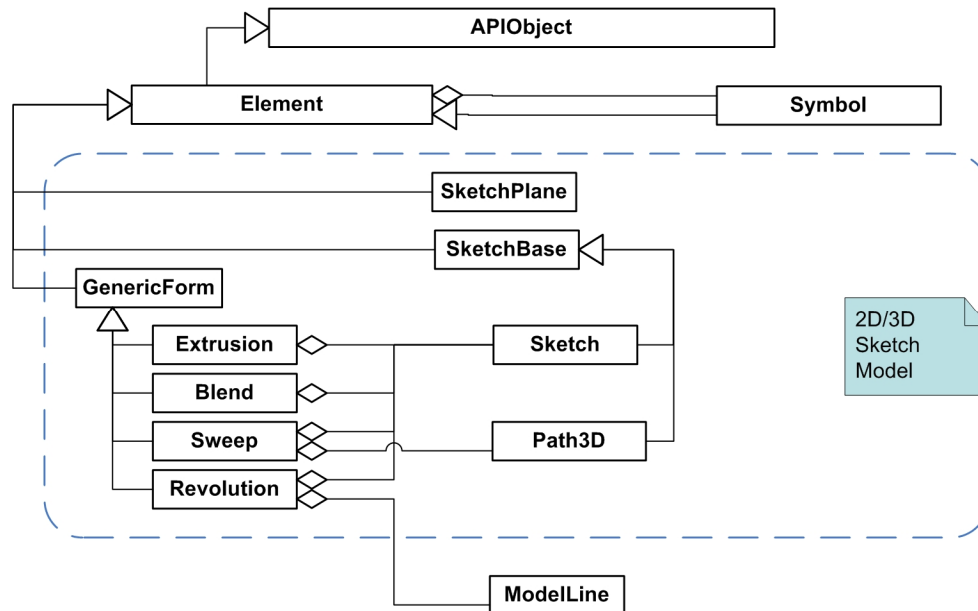


Figure 21: Sketch Elements UML diagram

5.1.6 Information Elements

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

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

For more information about Datum Elements, see the [Datum and Information Elements](#) chapter.

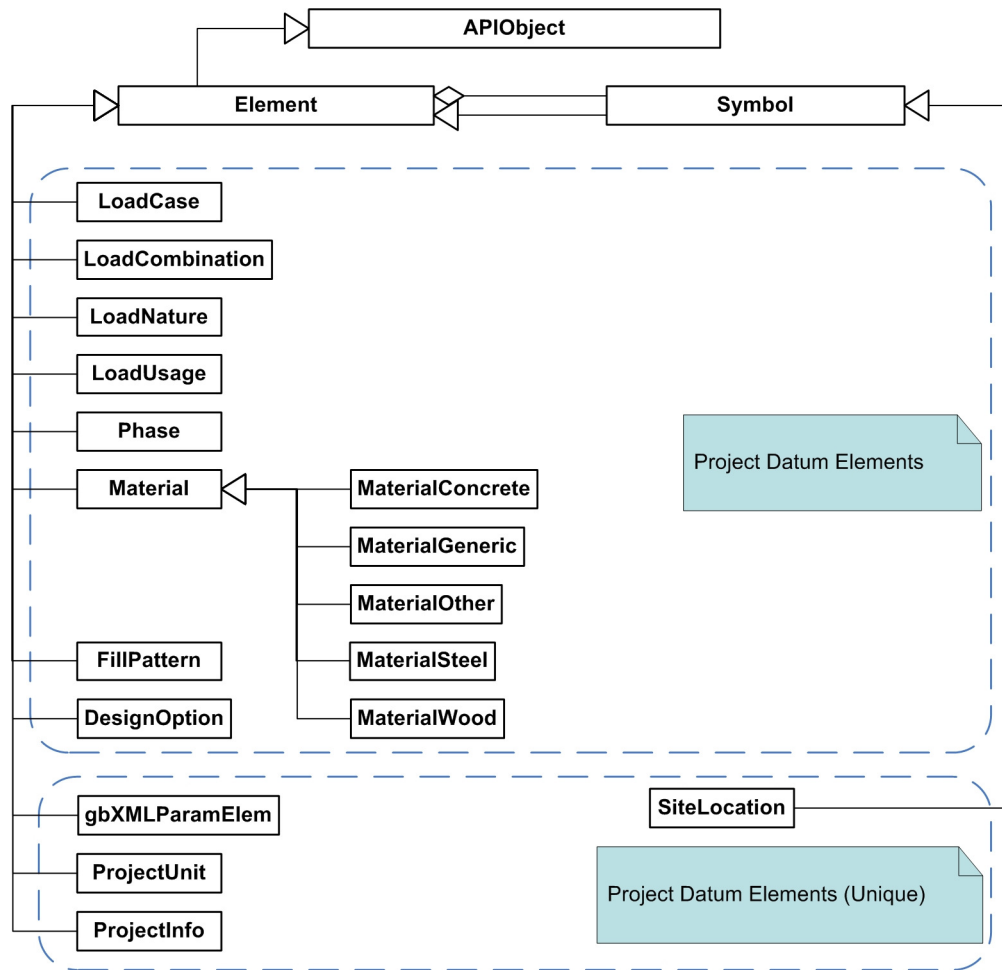


Figure 22: Information Elements diagram

5.2 Other Classifications

Elements are also classified by the following:

- Category
- Family
- Symbol
- Instance

There are some relationships between the classifications. For example:

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

5.2.1 Category

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

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

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

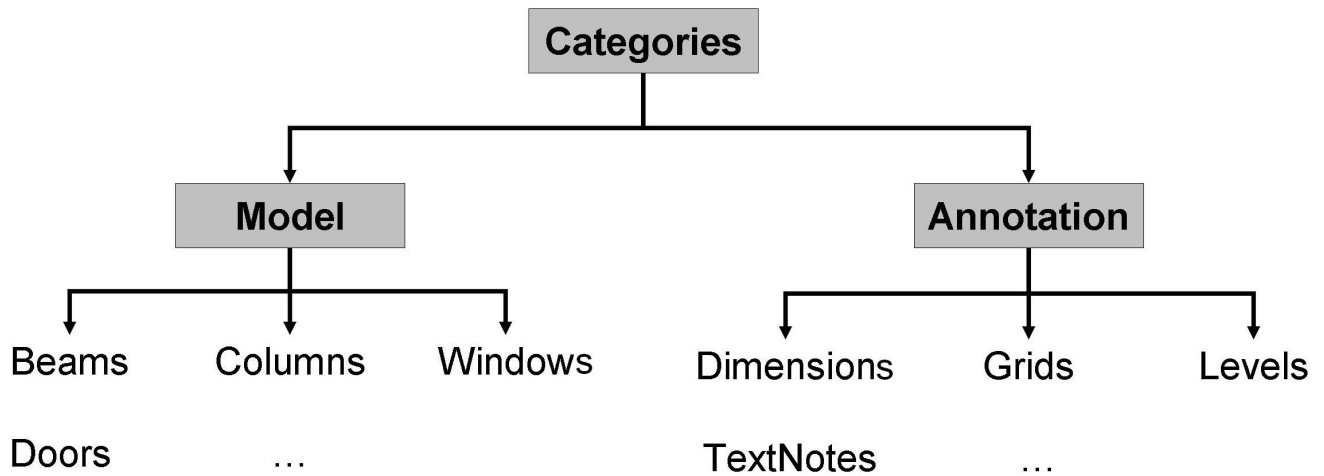


Figure 37: Categories

Note: The following guidelines apply to categories:

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

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

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

Code Region 5-1: Getting categories from document settings

```

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

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

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

```

```
// get Floor category according to OST_Floors and show its name
Category floorCategory = groups.GetItem(BuiltInCategory.OST_Floors);
prompt += floorCategory.Name;

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

Category is used in the following manner:

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

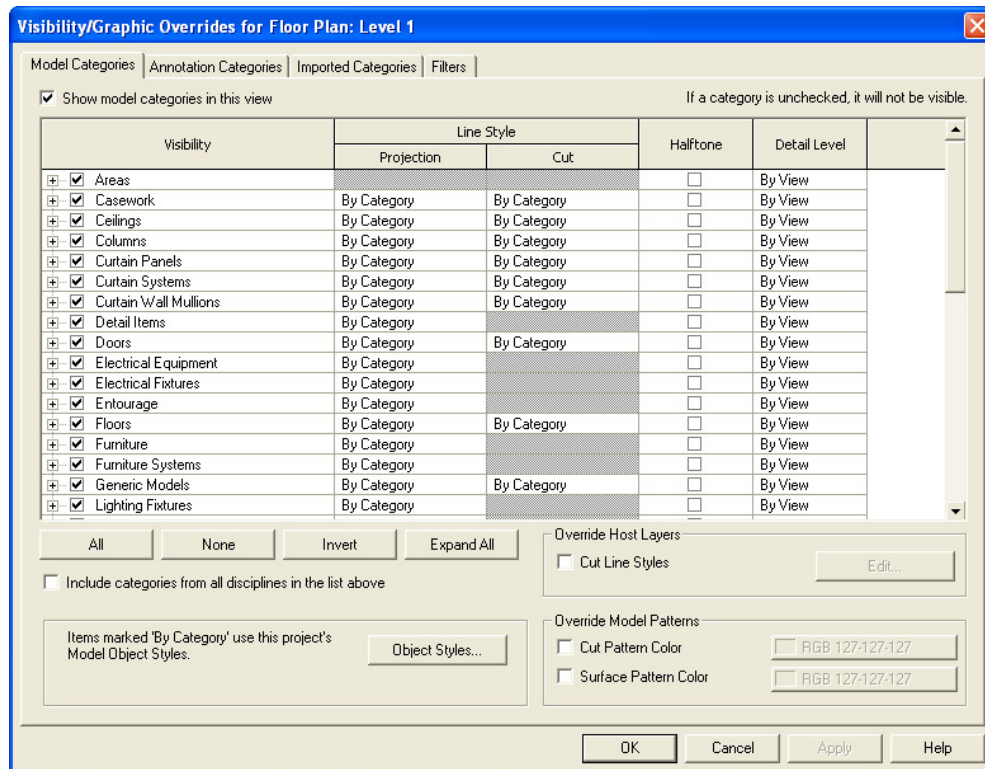


Figure 23: Visibility by Category

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

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

Code Region 5-2: Getting element category

```

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

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

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

```

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

5.2.2 Family

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

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

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

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

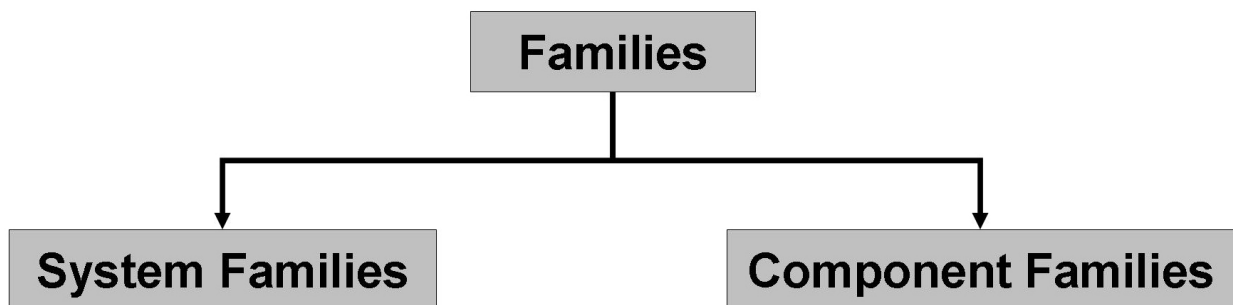


Figure 24: Families

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

5.2.2.1 Family in the Revit Platform API

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

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

- `Family`

- FamilySymbol
- FamilyInstance.

Each object plays a significant role in the Family structure.

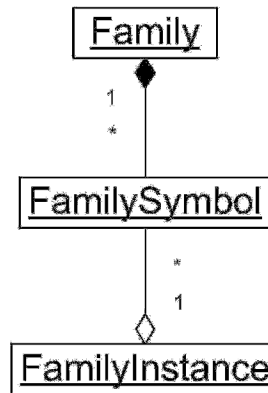


Figure 25: Family, FamilySymbol and FamilyInstance UML diagram

The Family object has the following characteristics:

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

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

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

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

Consequently, the following rules apply:

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

For more detailed information, see the [Family Instances](#) chapter.

5.2.3 Symbol

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

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

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

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

another type. Like a Family, a FamilySymbol is also a template. The FamilySymbol object is derived from the Symbol object and the Element object.

5.2.4 Instance

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

Note: For FamilyInstance, it is better to use the Symbol property instead of the ObjectType property to get the corresponding FamilySymbol. It is convenient and safe since you do not need to do a type conversion.

5.3 APIObject

Most reference types in the Revit Platform API, including Elements, are inherited from the APIObject class.

5.3.1 APIObject.IsReadOnly Property

The IsReadOnly property indicates whether the object is writable. If the property is set to true, an exception is raised when a method is called that requires write capabilities.

For example, Categories is set to IsReadOnly so that you cannot insert or remove an item. The Materials IsReadOnly setting is false so you can add and remove materials in the current document.

Code Region 5-3: Accessing APIObject.IsReadOnly

```
Document project = application.ActiveDocument;
Settings settings = project.Settings;
MessageBox.Show("Categories: " + settings.Categories.IsReadOnly.ToString());
MessageBox.Show("Materials: " + settings.Materials.IsReadOnly.ToString());
```

5.3.2 APIObject.ToString() Method

The method APIObject.ToString(), returns the object type derived from the APIObject.

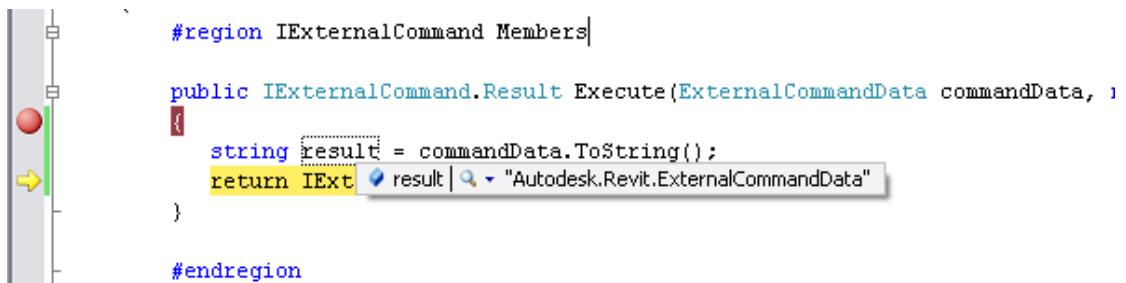


Figure 26: APIObject.ToString ()

The ToString() method inherited from APIObject works the same way as the APIObject.ToString() method. However, you cannot use the ToString() method to get object information such as the Element Name.

5.3.3 The Equals() Method

APIObject inherits the Equals() method from Object, and it is not overridden to correctly compare Revit objects from different sources. Therefore, you need to use different methods to compare Elements and other objects derived from APIObject. Compare Element and Category objects using

their `ElementId.Value`. Some classes provide specific compare methods. For example, `XYZ`, `UV`, and `Transform` use `AlmostEqual()` to compare two objects.

Code Region 5-4: Comparing Elements

```
private void CheckElementId(Autodesk.Revit.Document document, Autodesk.Revit.Element element)
{
    // Get the id of the element
    ElementId selectedId = element.Id;
    int idInteger = selectedId.Value;

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

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

    // And get the element using the original Id (they should be the same)
    Autodesk.Revit.Element second = document.get_Element(ref selectedId);

    String prompt = "Get an element twice. They should be the same element.";
    // Use the Equal method to check two elements. It is not a good way.
    bool isSame = first.Equals(second);
    prompt += "\nUsing Equal method on elements, the result is: " + isSame;
    isSame = first.Id.Equals(second.Id);
    prompt += "\nUsing Equal method on element's Id properties, the result is: " + isSame;

    MessageBox.Show(prompt);
}
```

5.4 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.

5.4.1 Getting an Element by ID

In general, an element is retrieved from the document `Elements` collection object using `ElementId` or iteration.

5.4.2 Iterating the Elements Collection

You can get the collection of all current project objects from the active document using the `Application.ActiveDocument.Elements` property. However, this isn't usually useful, since this property contains many abstract and invisible objects, and can take a very long time to iterate through. There are several overrides to this property that are optimized for performance, and operate on a specified type or filtered sub-set of elements:

Name	Description
<code>Elements(Type)</code>	Provides access to all elements of specified type within the Document.

Elements(Filter)	Provides access to all elements which satisfy specified filter within the Document.
Elements(Type, ICollection(Of Element))	Collects all elements which satisfy specified type within the Document.
Elements(Filter, ICollection(Of Element))	Collects all elements which satisfy specified filter within the Document.

For example, here's how you could get elements by type to build a list of all Wall elements in a document:

Code Region 5-5: Filtering elements

```
ElementIterator iterator = document.get_Elements(typeof(Wall));
String prompt = "The walls in the current document are:\n";
iterator.Reset();
while (iterator.MoveNext())
{
    Element e = iterator.Current as Element;
    prompt += e.Name + "\n";
}
MessageBox.Show(prompt, "Revit");
```

You can define a sub-set of Elements to iterate by specifying a filter or filters. You can filter on any combination of the following Element attributes:

- Category
- Family
- Instance usage
- Material
- Parameter
- Structural type
- Symbol
- Type
- Wall usage

You combine filters using LogicAndFilter, LogicOrFilter, and LogicNotFilter to implement AND, OR, and NOT relationships respectively. Here's an example of how you could iterate through all of the door FamilyInstances in a drawing (from the Door Swing SDK sample):

Code Region 5-6: Creating a logical filter

```
// get Filter Creation object
Autodesk.Revit.Creation.Filter filterCreator = document.Application.Create.Filter;

// Create a type filter for FamilyInstance, including derived types
TypeFilter familyInstanceFilter = filterCreator.NewTypeFilter(typeof(FamilyInstance), true);

// Create a category filter for Doors
CategoryFilter doorsCategoryfilter =
```

```

filterCreator.NewCategoryFilter(BuiltInCategory.Ost_Doors);

// Create a logic And filter for all Door FamilyInstances
Filter doorInstancesFilter = filterCreator.NewLogicAndFilter(familyInstanceFilter,
                                                            doorsCategoryfilter);

// Apply the filter to the elements in the active document
ElementIterator iter = document.get_Elements(doorInstancesFilter);

```

5.4.3 Selection

Rather than getting 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 `Application.ActiveDocument.Selection.Elements` property. The selected objects are in an `ElementSet` in Revit. From this `ElementSet`, all selected `Elements` are retrieved.

The selected `ElementSet` can be changed. To modify the `Selection.Elements`:

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

The following example illustrates how to change the selected `Elements`.

Code Region 5-7: Changing selected elements

```

// Get selected elements form current document.
Autodesk.Revit.SelElementSet collection = document.Selection.Elements;

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

//Create a new SelElementSet
SelElementSet newSelectedElementSet = document.Application.Create.NewSelElementSet();

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

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

// Give the user some information.
if (0 != newSelectedElementSet.Size)
{
    MessageBox.Show(document.Selection.Elements.Size.ToString() + " Walls are selected!");
}

```

```

else
{
    MessageBox.Show("No Walls have been selected!");
}

```

If you want to add one or more user-specified Elements to the set, use the `PickOne()` and `WindowSelect()` methods in the `Selection` class. This allows the user to select one or more Elements using the cursor and then returns control to your application.

- The `PickOne()` method adds one more Element to the set if the Element selected differs from the Elements already in the set.
- The `WindowSelect()` method adds the selected Elements to the set if possible.
- The `StatusBarTip` property shows a message in the status bar when your application prompts the user to pick elements.

Code Region 5-8: Adding selected elements with `PickOne()` and `WindowSelect()`

```

Selection choices = document.Selection;
// Pick one object from Revit.
bool hasPickOne = choices.PickOne();
if (true == hasPickOne)
{
    MessageBox.Show("One element added to Selection.", "Revit");
}

int selectionCount = choices.Elements.Size;
// Choose objects from Revit.
bool hasPickSome = choices.WindowSelect();
if (true == hasPickSome)
{
    int newSelectionCount = choices.Elements.Size;
    string prompt = string.Format("{0} elements added to Selection.",
                                   newSelectionCount - selectionCount);
    MessageBox.Show(prompt, "Revit");
}

```

The selected Elements and all Elements do not use the same collection class. You must write two separate functions depending on whether all Elements or selected Elements are needed.

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

For information about selecting an Element by its `ElementID`, see [ElementID](#) below.

5.4.4 Accessing Specific Elements from Document

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

Element	Access in property of Document
ProjectInfo	Document.ProjectInformation
ProjectUnit	Document.ProjectUnit
ProjectLocation	Document.ProjectLocations Document.ActiveProjectLocation
SiteLocation	Document.SiteLocation
Phase	Document.Phases
FillPattern	Document.Settings.FillPatterns
Material	Document.Settings.Materials
FamilySymbol relative to the deck profile of the layer within a structure	Document.DeckProfiles
FamilySymbol in the Title Block category	Document.TitleBlocks
All Element types	Document.AnnotationSymbolTypes / BeamSystemTypes / ContFootingTypes / DimensionTypes / FloorTypes / GridTypes / LevelTypes / RebarBarTypes / RebarHookTypes / RoomTagTypes / SpotDimensionTypes / WallTypes / TextNoteTypes

Table 10: Retrieving Elements from document properties

5.5 General Properties

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

5.5.1 ElementId

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

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

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

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

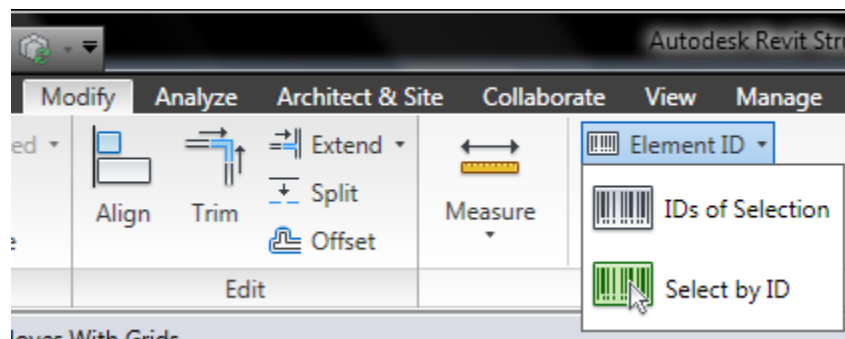


Figure 27: ElementId

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

Code Region 5-9: Setting ElementID

```
// Get the id of the element
ElementId selectedId = element.Id;
int idInteger = selectedId.Value;

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

ElementId has the following uses:

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

Code Region 5-10: Using ElementID

```
// Get the id of the element
ElementId selectedId = element.Id;
int idInteger = selectedId.Value;

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

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

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

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

5.5.2 UniqueId

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

Code Region 5-11: UniqueID

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

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

5.5.3 Location

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

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

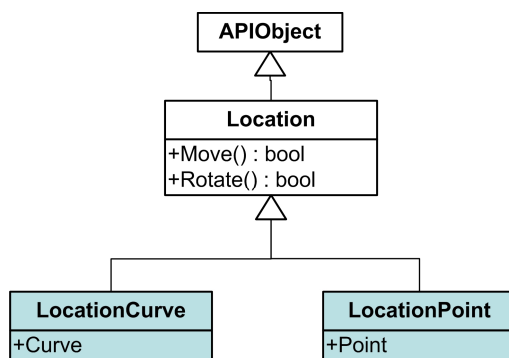


Figure 28: Location diagram

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

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

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

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

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

Table 11: Elements Location Information

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

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

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

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

For more information about LocationCurve and LocationPoint, see the [Editing Elements](#) chapter [Move](#) section.

5.5.4 Level

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

Code Region 5-12: Assigning Level

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

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

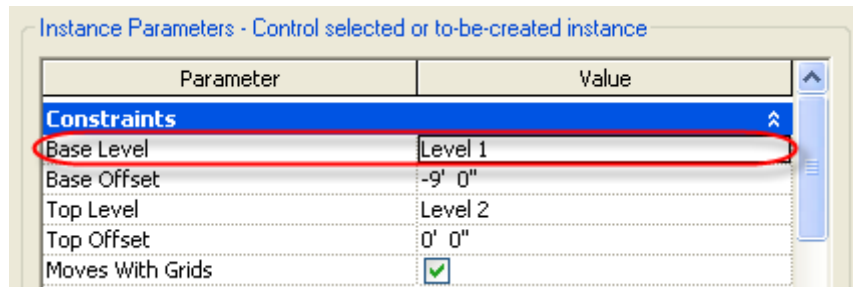


Figure 29: Column Base Level parameter

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

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

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

5.5.5 Parameter

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

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 30: Room parameters

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

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

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

For more details, see the [Parameter](#) Chapter.

6 Parameters

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

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

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

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

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

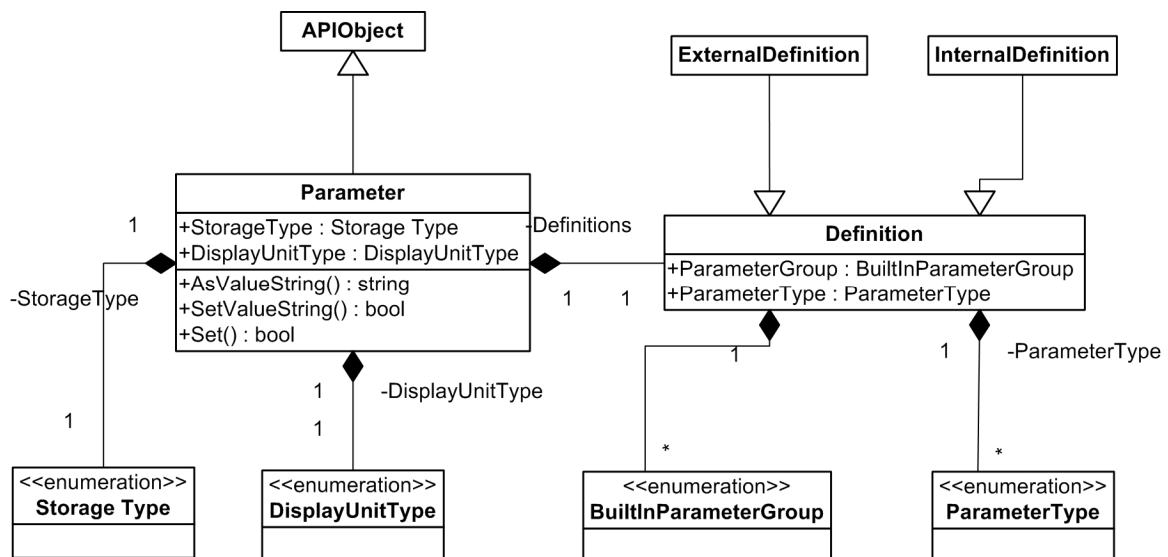


Figure 31: Parameter-related classes diagram

6.1 Walkthrough: Get Selected Element Parameters

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

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

Code Region 6-1: Getting selected element parameters

```
void GetElementParameterInformation(Document document, Element element)
```

```

{
    // Format the prompt information string
    String prompt = "Show parameters in selected Element:";

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

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

String GetParameterInformation(Parameter para, Document document)
{
    string defName = para.Definition.Name + @"\t";
    // Use different method to get parameter data according to the storage type
    switch (para.StorageType)
    {
        case StorageType.Double:
            //convert the number into Metric
            defName += " : " + para.AsValueString();
            break;
        case StorageType.ElementId:
            //find out the name of the element
            ElementId id = para.AsElementId();
            if (id.Value >= 0)
            {
                defName += " : " + document.get_Element(ref id).Name;
            }
            else
            {
                defName += " : " + id.Value.ToString();
            }
            break;
        case StorageType.Integer:
            if (ParameterType.YesNo == para.Definition.ParameterType)
            {
                if (para.AsInteger() == 0)
                {
                    defName += " : " + "False";
                }
                else
                {
                    defName += " : " + "True";
                }
            }
            else
            {
                defName += " : " + para.AsInteger().ToString();
            }
            break;
        case StorageType.String:
            defName += " : " + para.AsString();
            break;
        default:
            defName = "Unexposed parameter.";
            break;
    }

    return defName;
}

```

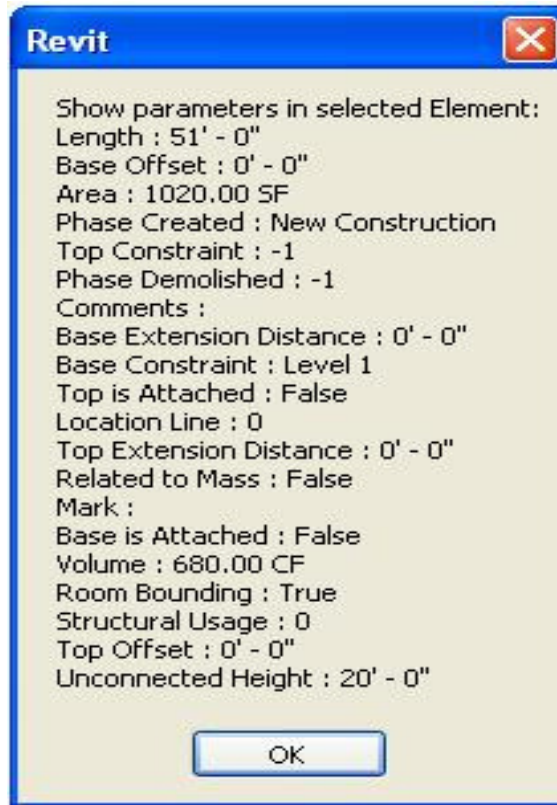


Figure 32: Get wall parameters result

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.

6.2 Definition

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

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

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

Code Region 6-2: Finding a parameter based on definition type

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

6.2.1 ParameterType

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

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

For more details about ParameterType.Material, see the [Material](#) chapter.

6.2.2 ParameterGroup

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

6.3 BuiltInParameter

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

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

Code Region 6-3: Getting a parameter based on BuiltInParameter

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

    return parameter;
}
```

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

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

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

6.4 StorageType

StorageType describes the type of parameter values stored internally.

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

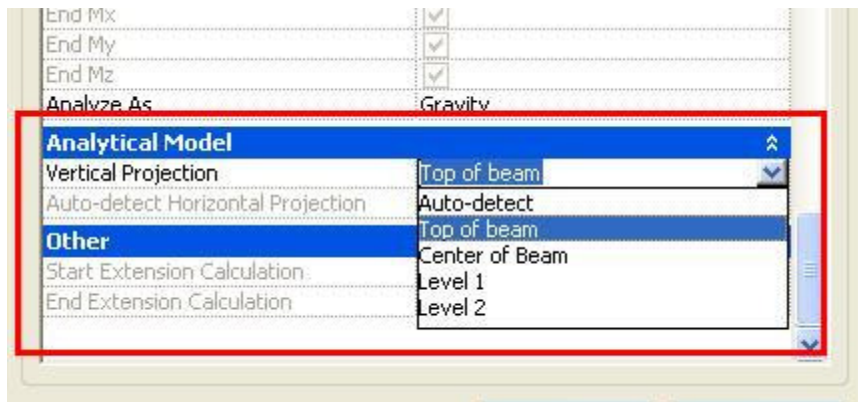
The StorageType is an enumerated type that lists all internal parameter data storage types supported by Revit:

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

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

Table 12: Storage Type

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

**Figure 33: Storage type sample**

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

Code Region 6-4: Checking a parameter's StorageType

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

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

    return result;
}
```

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

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

6.5 AsValueString() and SetValueString()

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

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

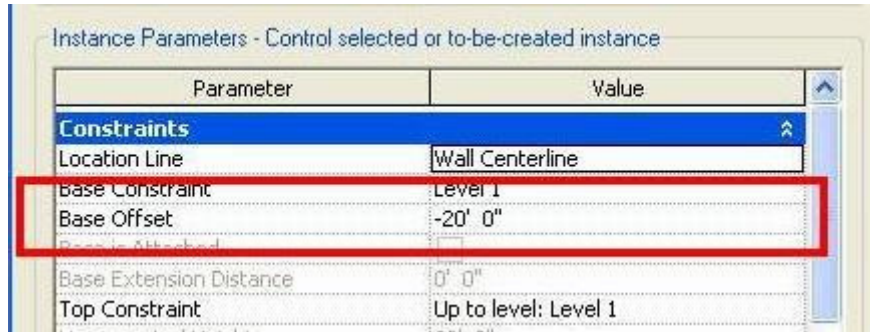


Figure 34: AsValueString and SetValueString sample

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

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

Code Region 6-5: Using Parameter.SetValueString()

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

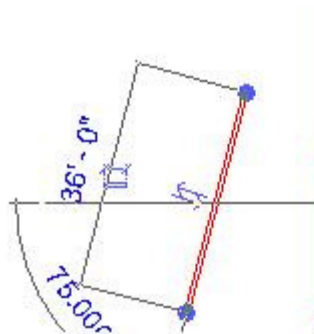
6.6 Parameter Relationships

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

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

Additionally, some parameters are always read-only.

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



Parameter	Value
Room Bounding	☒
Related to Mass	☐
Structural	
Structural Usage	Non-bearing
Dimensions	
Length	36' 0"
Area	720.00 SF
Volume	480.00 CF
Identity Data	

Figure 35: 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 6-6: Parameter relationship example

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

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

6.7 Adding Parameters to Elements

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

For more details, refer to the [Shared Parameter](#) chapter.

7 Collections

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

- Array
- Set
- Map

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

The collections inheritance hierarchy is as follows:

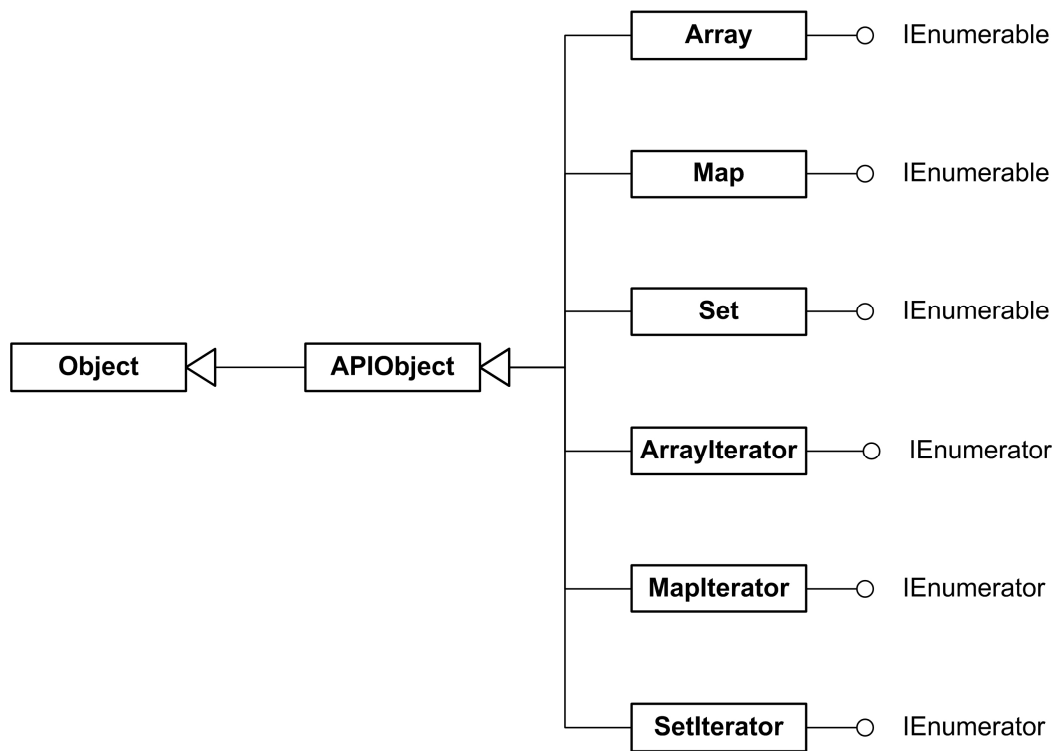


Figure 36: Collection diagram

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

7.1 Interface

The following sections discuss interface-related collection types.

7.1.1 IEnumerable

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

7.1.2 IEnumerator

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

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

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

Use IEnumerator as follows:

- Initially, the enumerator is positioned in front of the first element in the collection. However, it is a good idea to always call Reset() when you first obtain the enumerator.
 - The Reset() method moves the enumerator back to the original position. At this position, calling the Current property throws an exception.
 - Call the MoveNext() method to advance the enumerator to the collection's first element before reading the current iterator value.
- The Current property returns the same object until either the MoveNext() method or Reset() method is called. The MoveNext() method sets the current iterator to the next element.
- If MoveNext passes the end of the collection, the enumerator is positioned after the last element in the collection and MoveNext returns false.
 - When the enumerator is in this position, subsequent calls to the MoveNext also return false.
 - If the last call to the MoveNext returns false, calling the Current property throws an exception.
 - To set the current iterator to the first element in the collection again, call the Reset() method followed by MoveNext().
- An enumerator remains valid as long as the collection remains unchanged.
 - If changes are made to the collection, such as adding, modifying, or deleting elements, the enumerator is invalidated and the next call to the MoveNext() or the Reset() method throws an InvalidOperationException.
 - If the collection is modified between the MoveNext and the current iterator, the Current property returns to the specified element, even if the enumerator is already invalidated.

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

7.2 Collections and Iterators

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

Code Region 7-1: Using ElementSet and ElementIterator

```
ElementSet elems = document.Selection.Elements;

string info = "Selected elements:\n";
foreach (Autodesk.Revit.Element elem in elems)
```

```

{
    info += elem.Name + "\n";
}

MessageBox.Show(info, "Revit");

info = "Levels in document:\n";
ElementIterator elemItr = document.get_Elements(typeof(Level));
elemItr.Reset();
while (elemItr.MoveNext())
{
    Autodesk.Revit.Element elem = elemItr.Current as Autodesk.Revit.Element;
    // you needn't check null for elem
    info += elem.Name + "\n";
}

MessageBox.Show(info, "Revit");

```

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

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

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

Table 13: Collections Methods and Properties

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

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

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

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

Table 14: Method and Property Iterators

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

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

Code Region 7-2: Using collections

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

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

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

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

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

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

    XYZ startPoint = new XYZ(0, 0, 0); // the start point of the line
```

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

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

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

MessageBox.Show("After Insert: " + lineArray.Size + " in lineArray.");

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

MessageBox.Show("After Clear: " + lineArray.Size + " in lineArray.");
```

8 Editing Elements

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

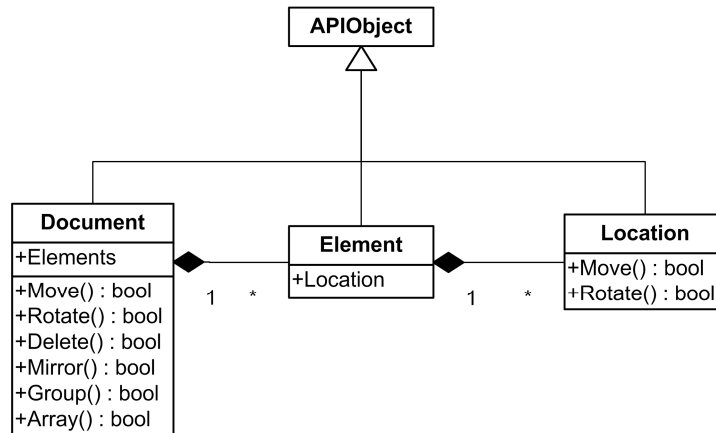


Figure 37: Editing Elements diagram

8.1 Moving Elements

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

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

Table 15: Move Members

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

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

Code Region 8-1: Using Move()

```

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

```

XYZ oldPlace = columnLocation.Point;

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

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

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

MessageBox.Show(info);
}

```

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

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

Code Region 8-2: Moving using Location

```

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

```

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

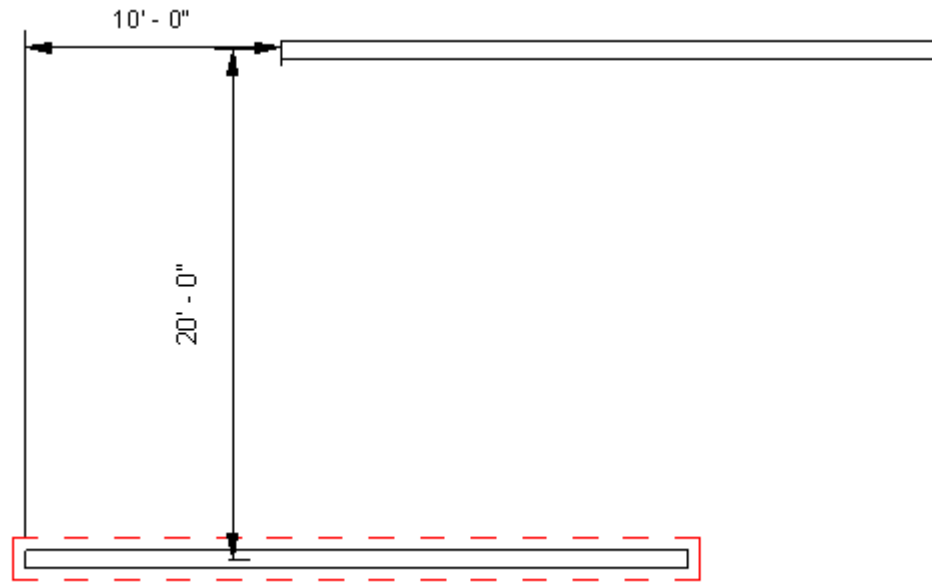


Figure 38: Move a wall using the LocationCurve

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 8-3: Moving using Curve

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

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

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

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

Code Region 8-4: 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;
}
}

```

8.2 Rotating elements

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

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

Table 16: Rotate Members

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

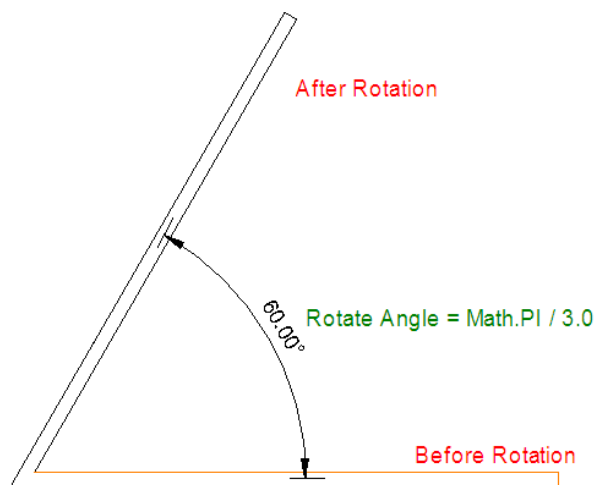


Figure 39: Counterclockwise rotation

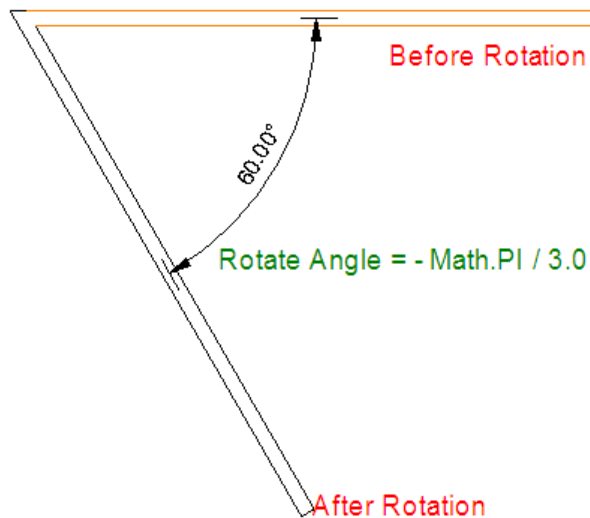


Figure 40: Clockwise rotation

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

Code Region 8-5: Using Rotate()

```
public bool RotateColumn(Autodesk.Revit.Document document, Autodesk.Revit.Element element)
{
    XYZ point1 = new XYZ(10, 20, 0);
    XYZ point2 = new XYZ(10, 20, 30);
    // The axis should be a bound line.
    Line axis = document.Application.Create.NewLineBound(point1, point2);
    bool successful = document.Rotate(element, axis, Math.PI / 3.0);

    return successful;
}
```

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

Code Region 8-6: Rotating based on location curve

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

```

        rotated = curve.Rotate(axis, Math.PI / 2.0);
    }

    return rotated;
}

```

Code Region 8-7: Rotating based on location point

```

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

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

    return rotated;
}

```

8.3 Mirror

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

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

Table 17: Mirror Members

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

Note: This code only works in Revit Structure.

Code Region 8-8: Mirroring a column

```

public void MirrorColumn(Autodesk.Revit.Document document, FamilyInstance column)
{
    AnalyticalModelFrame analytical = column.AnalyticalModel as AnalyticalModelFrame;
    Reference lineReference = analytical.Curve.Reference;
}

```

```

document.Mirror(column, lineReference);

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

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

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

```

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

8.4 Group

The Revit Platform API uses the Creation.Document.NewGroup() method to select an element or multiple elements or groups and then combines them. With each instance of a group that you place, there is associativity 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 8-9: Creating a Group

```

Group group = null;

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

```

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

Code Region 8-10: Naming a Group

```

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

```

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

- If no detail Element is passed, a Model Group is created.
- If all Elements are detail elements, the 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.

8.5 Array

The Revit Platform API uses the overloaded Document.Array() methods to array one or more elements in the project. These methods create a linear or radial array of one or more selected components. Linear arrays represent an array created along a line from one point, while radial arrays represent an array created along an arc.

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

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

Table 18 : Document Array Members

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

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

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

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

8.6 Delete

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

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

Table 19: Delete Members

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

Code Region 8-11: Deleting an element based on object

```
private void DeleteElement(Autodesk.Revit.Document document, Element element)
{
    // Delete a selected element.
    ElementIdSet deletedIdSet = document.Delete(element);

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

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

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

Code Region 8-12: Deleting an element based on ElementId

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

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

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

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

Note: When an element is deleted, any child elements associated with that element are also deleted, as indicated in the samples above.

The API also provides two ways to delete several elements.

Code Region 8-13: Deleting an ElementSet

```
// Delete all the selected elements via the set of elements
ElementSet elementSet = document.Selection.Elements;
ElementIdSet deletedIdSet = document.Delete(elementSet);

if (0 == deletedIdSet.Size)
{
    throw new Exception("Deleting the selected elements in revit failed.");
}

MessageBox.Show("The selected element has been removed.", "Revit");
```

Code Region 8-14: Deleting an ElementSet based on Id

```
// Delete all the selected elements via the set of element ids.
ElementIdSet idSelection = new ElementIdSet();
foreach (Autodesk.Revit.Element elem in document.Selection.Elements)
{
    ElementId id = elem.Id;
    idSelection.Insert(ref id);
}
ElementIdSet deletedIdSet = document.Delete(idSelection);
```

```

if (0 == deletedIdSet.Size)
{
    throw new Exception("Deleting the selected elements in revit failed.");
}

MessageBox.Show("The selected element has been removed.", "Revit");

```

Note: After you delete the elements, any references to the deleted elements become invalid and throw an exception if they are accessed.

Some elements have special deletion methods. For example, the `Materials.Remove()` method is specifically for deleting Material objects. For more information, see the [Material](#) chapter [Material Management](#) section.

8.7 SuspendUpdating

`SuspendUpdating` is a class that temporarily delays automatic updating and consistency operations in Revit. For example, if you create an instance of this object, it suspends certain updating and consistency operations to increase large-scale change performance using the API.

Updating continues when the `SuspendUpdating` object is explicitly disposed. Currently, the `SuspendUpdating` class works primarily with object transformation operations such as move and rotate. For a complete list of operations which this class suspends, see the Revit API Help file.

Code Region 8-15: Using SuspendUpdating

```

public void MoveElements(Autodesk.Revit.Document document)
{
    Selection choices = document.Selection;
    ElementSet collection = choices.Elements;
    // Using statement defines a scope at the end of which the suspendUpdating
    // object is disposed.
    using (SuspendUpdating aSuspendUpdating = new SuspendUpdating(document))
    {
        // Suspend these operations.
        foreach (FamilyInstance element in collection)
        {
            MoveRotateElement(document, element);
        }
    }
    // Move() operations only update when aSuspendUpdating and
    // bSuspendUpdating (in MoveRotateElement()) are disposed
}

```

`SuspendUpdating` objects can be nested. The `Nested` property returns true if the object is already nested. During nesting, the final update happens only when all `SuspendUpdating` objects are disposed. For example, if the following function is called by the previous example, the `bSuspendUpdating` object `Nested` property returns true. In this situation, the `Move()` operation update only happens after both `bSuspendUpdating` and `aSuspendUpdating` objects are disposed.

Code Region 8-16: Nesting SuspendUpdating

```
private bool MoveRotateElement(Document document, Autodesk.Revit.Element elem)
{
    // Operations suspended until SuspendUpdating object is disposed
    using (SuspendUpdating bSuspendUpdating = new SuspendUpdating(document))
    {
        // Move Element
        XYZ translation3D = new XYZ(0.0, 0.0, 0.0);
        bool successfull = document.Move(elem, translation3D);

        // Rotate Element
        XYZ point1 = new XYZ(10, 20, 0);
        XYZ point2 = new XYZ(10, 20, 30);
        Line axis = document.Application.Create.NewLineUnbound(point1, point2);
        bool successful2 = document.Rotate(elem, axis, Math.PI / 3.0);

        return (successfull && successful2);
    }
}
```

9 Walls, Floors, Roofs and Openings

This chapter discusses Elements and the corresponding Symbols representing built-in place construction:

- HostObject - The first two sections focus on HostObject and corresponding HostObjAttributes subclasses shown in the following diagram:

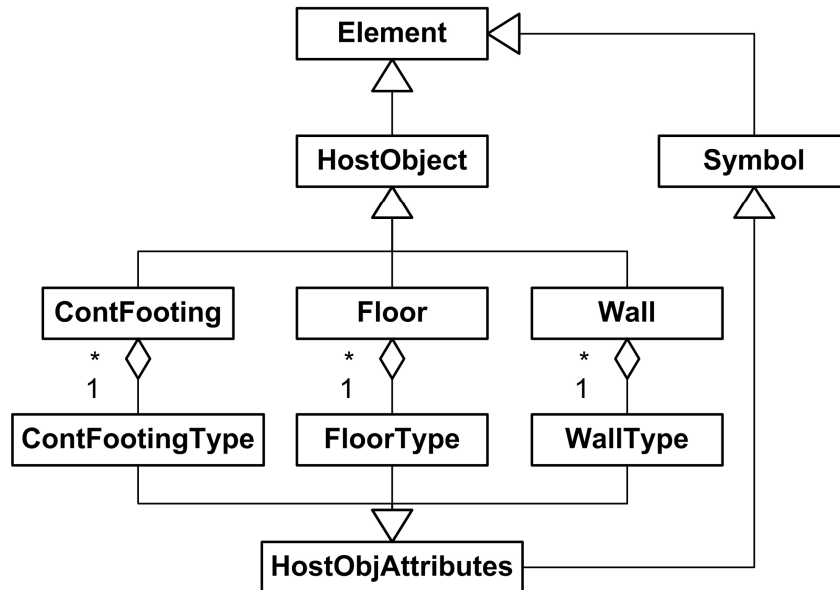


Figure 41: HostObject and HostObjAttributes diagram

- Foundation - Different foundations in the API are represented as different classes, including Floor, ContFooting, and FamilyInstance. The Floor and Foundation section compares them in the API.
- CompoundStructure - This section describes the HostObject.CompoundStructure property and provides tips to access Material.

In addition to host Elements, the Opening class is introduced at the end of this chapter.

9.1 Walls

There are four kinds of Walls represented by the WallType.WallKind enumeration:

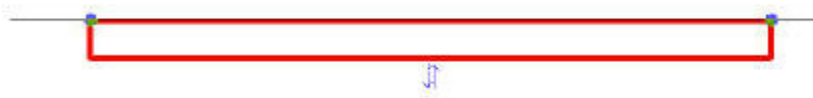
- Stacked
- Curtain
- Basic
- Unknown

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 42: Original wall****Figure 43: Wall after flip**

Location Line Value	Description
0	Wall Centerline
1	Core Centerline
2	Finish Face: Exterior
3	Finish Face: Interior
4	Core Face: Exterior
5	Core Face: Interior

Table 20: Wall Location Line

There are five override methods in the Document class to create a Wall:

Name	Description
NewWall(Curve, WallType, Level, Double, Double, Boolean, Boolean)	Creates a new rectangular profile wall within the project.
NewWall(CurveArray, Boolean)	Creates a non rectangular profile wall within the project using the default wall style.
NewWall(Curve, Level, Boolean)	Creates a new rectangular profile wall within the project on the specified level using the default wall style.
NewWall(CurveArray, WallType, Level, Boolean)	Creates a non rectangular profile wall within the project.
NewWall(CurveArray, WallType, Level, Boolean, XYZ)	Creates a non rectangular profile wall within the project with a specific normal.

Table 21: NewWall() Overrides

The WallType Wall Function (WALL_ATTR_EXTERIOR) parameter influences the created wall instance Room Bounding and Structural Usage parameter. The WALL_ATTR_EXTERIOR value is an integer:

Wall Function	Interior	Exterior	Foundation	Retaining	Soffit

Wall Function	Interior	Exterior	Foundation	Retaining	Soffit
Value	0	1	2	3	4

Table 22: Wall Function

There are two override methods in the Document class to batch create multiple walls:

Name	Description
NewWalls(List(Of RectangularWallCreationData))	Creates rectangular walls within the project.
NewWalls(List(Of ProfiledWallCreationData))	Creates profile walls within the project.

Table 23: NewWalls() Overrides

The following rules apply to Walls created by the API:

- If the input structural parameter is true or the Wall Function (WALL_ATTR_EXTERIOR) parameter is Foundation, the Wall StructuralUsage parameter is Bearing; otherwise it is NonBearing.
- The created Wall Room Bounding (WALL_ATTR_ROOM_BOUNDING) parameter is false if the Wall Function (WALL_ATTR_EXTERIOR) parameter is Retaining.

For more information about structure-related functions such as the AnalyticalModel property, refer to the [Revit Structure](#) chapter.

9.2 Floors and Foundations

Floor and Foundation-related API items include:

Object	Element Type	Symbol Type	Element Creation	Other
Floor	Floor	FloorType	NewFloor()/NewSlab()	StructuralUsage = InstanceUsage.Slab FloorType.IsFoundationSlab = false
Slab	Floor	FloorType	NewSlab()	StructuralUsage = InstanceUsage.Slab FloorType.IsFoundationSlab = false
Wall Foundation	ContFooting	ContFootingType	No	Category = OST_StructuralFoundation
Isolated Foundation	FamilyInstance	FamilySymbol	NewFamilyInstance()	Category = OST_StructuralFoundation
Foundation Slab	Floor	FloorType	NewFloor()	Category = OST_StructuralFoundation StructuralUsage = InstanceUsage.SlabFoundation FloorType.IsFoundationSlab = true

Table 24: Floors and Foundations in the API

The following rules apply to Floor:

- Elements created from the Foundation Design bar have the same category, OST_StructuralFoundation, but correspond to different Classes.
- The FloorType IsFoundationSlab property sets the FloorType category to OST_StructuralFoundation or not.

When you retrieve FloorType to create a Floor or Foundation Slab with NewFloor, use the following methods:

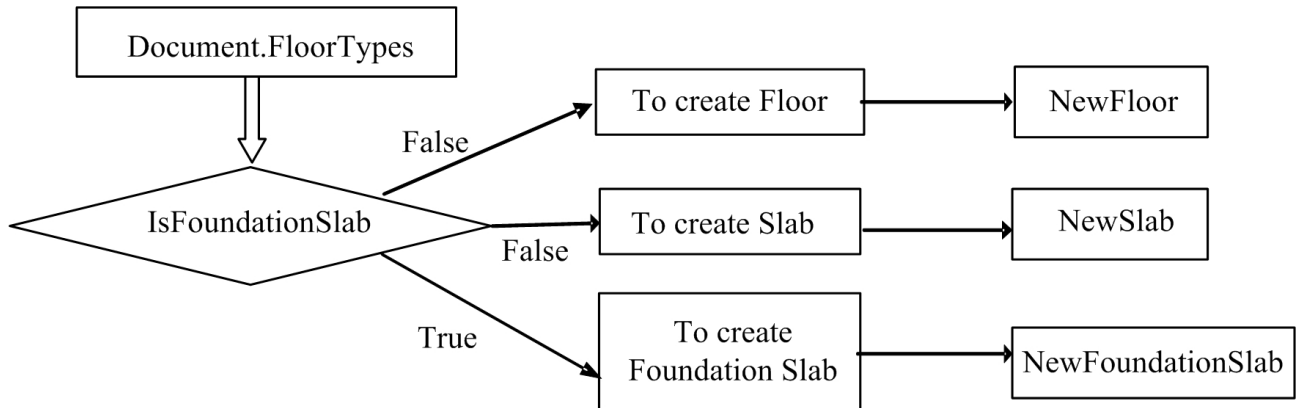


Figure 44: Create foundation and floor/slab

NewSlab() is an override NewFloor() method. Currently, the API does not provide access to the Floor Slope Arrow in the Floor class. However, in Revit Structure, you can create a sloped slab with NewSlab():

Code Region 9-1: NewSlab()

```
public Floor NewSlab(CurveArray profile, Level level, Line slopedArrow, double angle,
bool isImperial, bool isStructural);
```

The Slope Arrow is created using the `slopedArrow` parameter.

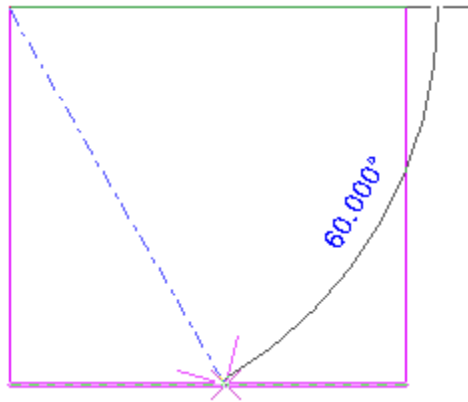


Figure 45: `slopedArrow` parameter in `NewSlab`

`Floor.FloorType` is preferred to `Floor.ObjectType` because it has a setter and does not need to be downcast to `FloorType`. For more information about structure-related functions such as `SpanDirectionSymbols` and `SpanDirectionAngle`, refer to the [Revit Structure](#) chapter.

When editing an Isolated Foundation in Revit, you can perform the following actions:

- You can pick a host, such as a floor. However, the `FamilyInstance` object `Host` property always returns null.
- When deleting the host floor, the Foundation is not deleted with it.
- The Foundation host is available from the `Host (INSTANCE_FREE_HOST_PARAM)` parameter.
- Use another related `Offset (INSTANCE_FREE_HOST_OFFSET_PARAM)` parameter to control the foundation offset from the host Element.

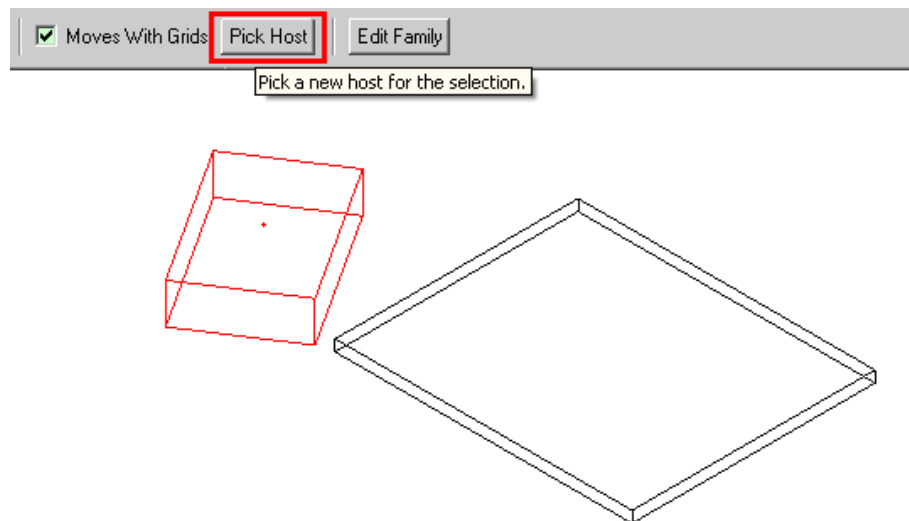


Figure 46: Pick Host for `FoundationSlab (FamilyInstance)`

Continuous footings are represented by the `ContFooting` class in the API. The API provides limited access to both `ContFooting` and `ContFootingType` except when using `AnalyticalModel` (refer to the [AnalyticalModel](#) section in the [Revit Structure](#) chapter). For example, the attached wall is not available in Revit Architecture. In Revit Structure, the relationship between the `Wall` class and the `ContFooting` class is shown using `SupportData` in the structural `Wall` class `AnalyticalModelWall`. For more details, refer to the [SupportData](#) section in the [Revit Structure](#) chapter.

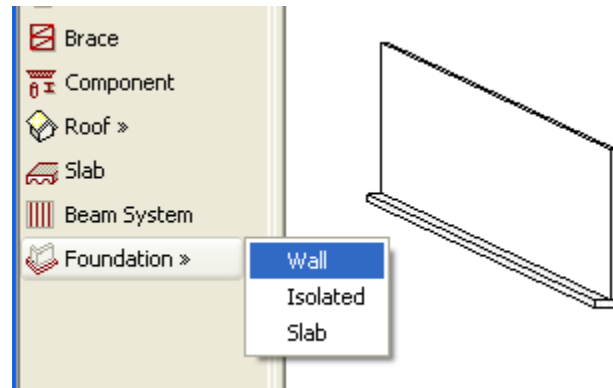


Figure 47: Wall ContFooting

9.2.1 Modifying Slabs

You can modify the form of slab-based elements using the `SlabShapeEditor` class. This class allows you to:

- Manipulate one or more of the points or edges on a selected slab-based element
- Add points on the element to change the element's geometry
- Add linear edges and split the existing face of a slab into smaller sub-regions
- Remove the shape modifier and reset the element geometry back to the unmodified shape.

Here's an example of reverting a selected modified floor back to its original shape:

Code Region 9-2: Reverting a slab's shape

```
private void ResetSlabShapes(Autodesk.Revit.Document document)
{
    Selection choices = document.Selection;
    ElementSet collection = choices.Elements;
    foreach (Autodesk.Revit.Element elem in collection)
    {
        Floor floor = elem as Floor;
        if (floor != null)
        {
            SlabShapeEditor slabShapeEditor = floor.SlabShapeEditor;
            slabShapeEditor.ResetSlabShape();
        }
    }
}
```

For more detailed examples of using the `SlabShapeEditor` and related classes, see the `SlabShapeEditing` sample application included in the Revit SDK.

9.3 Roofs

Roofs in the Revit Platform API all derive from the `RoofBase` object. There are two classes:

- `FootPrintRoof` – represents a roof made from a building footprint
- `ExtrusionRoof` – represents roof made from an extruded profile

Both have a `RoofType` property that gets or sets the type of roof. This examples shows how you can create a footprint roof based on some selected walls:

Code Region 9-3: Creating a footprint roof

```
// Before invoking this sample, select some walls to add a roof over.
// Make sure there is a level named "Roof" in the document.

// find the Roof level
Level level = null;
ElementIterator elemItor = document.get_Elements(typeof(Level));
elemItor.Reset();
while (elemItor.MoveNext())
{
    Level currentLevel = elemItor.Current as Level;
    if (currentLevel.Name == "Roof")
    {
        level = currentLevel;
    }
}

RoofType rooftype = null;
// select the first rooftype
foreach (RoofType rt in document.RoofTypes)
{
    rooftype = rt;
    break;
}

// Get the handle of the application
Autodesk.Revit.Application application = document.Application;

// Define the footprint for the roof based on user selection
CurveArray footprint = application.Create.NewCurveArray();

if (document.Selection.Elements.Size != 0)
{
    foreach (Autodesk.Revit.Element element in document.Selection.Elements)
    {
        Wall wall = element as Wall;
        if (wall != null)
        {
            LocationCurve wallCurve = wall.Location as LocationCurve;
            footprint.Append(wallCurve.Curve);
            continue;
        }

        ModelCurve modelCurve = element as ModelCurve;
        if (modelCurve != null)
        {
            footprint.Append(modelCurve.GeometryCurve);
        }
    }
}
else
{
    throw new Exception("You should select a curve loop, or a wall loop, or loops combination \nof walls and curves to create a footprint roof.");
}
```


For an example of how to create an ExtrusionRoof, see the NewRoof sample application included with the Revit API SDK.

9.3.1 Gutter and Fascia

Gutter and Fascia elements are derived from the HostedSweep class, which represents a roof. They can be created, deleted or modified via the API. To create these elements, use one of the Document.Create.NewFascia() or Document.Create.NewGutter() overrides. For an example of how to create new gutters and fascia, see the NewHostedSweep application included in the SDK samples. Below is a code snippet showing you can modify a gutter element's properties.

Code Region 9-4: Modifying a gutter

```
public void ModifyGutter(Autodesk.Revit.Document document)
{
    ElementSet collection = document.Selection.Elements;

    foreach (Autodesk.Revit.Element elem in collection)
    {
        if (elem is Gutter)
        {
            Gutter gutter = elem as Gutter;
            // convert degrees to rads:
            gutter.Angle = 45.00 * Math.PI / 180;
            MessageBox.Show("Changed gutter angle", "Revit");
        }
    }
}
```

9.4 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.

9.5 Other Elements

The following Elements are not HostObjects (and don't have a specific class), but are special cases that can host other objects.

9.5.1 Stair and Ramp

Stair, Ramp, and the associated symbols do not have specific classes in the API.

- Stair and its associated symbol are represented as Element and Symbol in the OST_Stairs category.
- Ramp and its symbol are represented as Element and Symbol in the OST_Ramps category.

9.5.2 Ceiling

Ceiling does not have a specific class, so you can't create it using the API. The Ceiling object is an Element in the OST_Ceilings category.

- The Ceiling symbol is HostObjAttributes using the OST_Ceilings category with its CompoundStructure available.

9.6 CompoundStructure

Some host elements, such as Walls, Floors, Ceilings, and Roofs, include parallel layers. The parallel layers are available from the HostObjAttributes.CompoundStructure property:

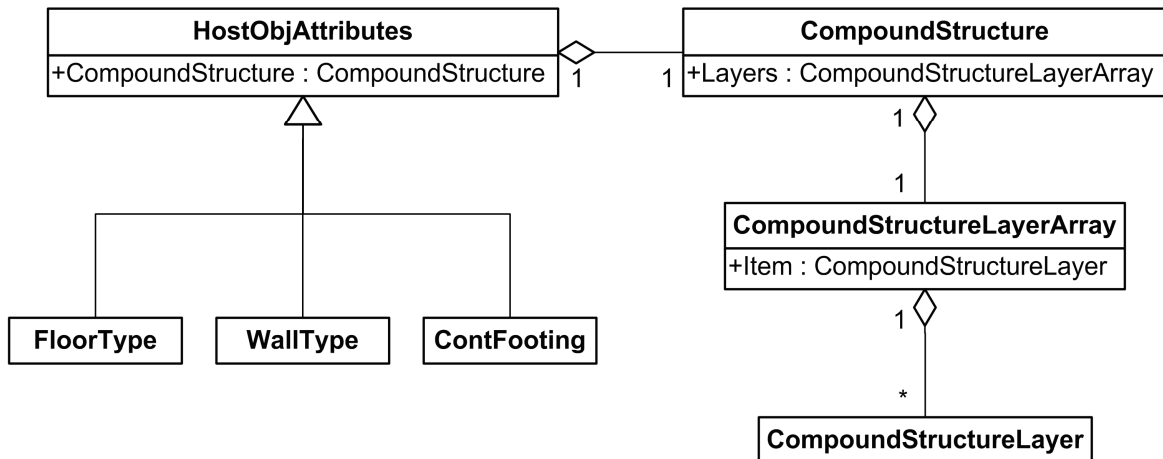


Figure 48: CompoundStructure diagram

Note: The following items are important when using CompoundStructure:

- The total thickness of the element is the sum of each CompoundStructureLayer's thickness. You cannot change the element's total thickness directly but you can change it via changing the CompoundStructureLayer thickness.
- CompoundStructure is a property of HostObjAttributes. Changing the CompoundStructureLayer changes every element instance in the current document.
- The CompoundStructureLayerArray ReadOnly property retrieved from the element CompoundStructure is always true. You cannot insert, remove, or reorder CompoundStructureLayers within the array.
- The CompoundStructureLayer DeckProfile, DeckUsage, and Variable properties only work with Slab in Revit Structure. For more details, refer to the [Revit Structure](#) chapter.

9.6.1 Material

Each CompoundStructureLayer in HostObjAttributes is typically displayed with some type of material. If CompoundStructureLayer.Material returns null, it does not mean the layer does not have a Material; it means the Material is Category-related. For more details, refer to the [Material](#) chapter. Getting the CompoundStructureLayer Material is illustrated in the following sample code:

Code Region 9-5: Getting the CompoundStructureLayer Material

```

public void GetWallLayerMaterial(Autodesk.Revit.Document document, Wall wall)
{
    // get WallType of wall
    WallType aWallType = wall.WallType;
    // Only Basic Wall has compoundStructure
    if (WallType.WallKind.Basic == aWallType.Kind)
    {
        // Get CompoundStructure
    }
}
  
```

```

CompoundStructure comStruct = aWallType.CompoundStructure;
Categories allCategories = document.Settings.Categories;

// Get the category OST_Walls default Material;
// use if that layer's default Material is <By Category>
Category wallCategory = allCategories.get_Item(BuiltInCategory.OST_Walls);
Autodesk.Revit.Elements.Material wallMaterial = wallCategory.Material;

foreach (CompoundStructureLayer structLayer in comStruct.Layers)
{
    Autodesk.Revit.Elements.Material layerMaterial = structLayer.Material;
    // If CompoundStructureLayer's Material is specified, use default
    // Material of its Category
    if (null == layerMaterial)
    {
        switch (structLayer.Function)
        {
            case CompoundStructureLayerFunction.Finish1:
                layerMaterial =
allCategories.get_Item(BuiltInCategory.OST_WallsFinish1).Material;
                break;
            case CompoundStructureLayerFunction.Finish2:
                layerMaterial =
allCategories.get_Item(BuiltInCategory.OST_WallsFinish2).Material;
                break;
            case CompoundStructureLayerFunction.MembraneLayer:
                layerMaterial =
allCategories.get_Item(BuiltInCategory.OST_WallsMembrane).Material;
                break;
            case CompoundStructureLayerFunction.Structure:
                layerMaterial =
allCategories.get_Item(BuiltInCategory.OST_WallsStructure).Material;
                break;
            case CompoundStructureLayerFunction.Substrate:
                layerMaterial =
allCategories.get_Item(BuiltInCategory.OST_WallsSubstrate).Material;
                break;
            case CompoundStructureLayerFunction.ThermalOrAir:
                layerMaterial =
allCategories.get_Item(BuiltInCategory.OST_WallsInsulation).Material;
                break;
            default:
                // It is impossible to reach here
                break;
        }
        if (null == layerMaterial)
        {
            // CompoundStructureLayer's default Material is its SubCategory
            layerMaterial = wallMaterial;
        }
        MessageBox.Show("Layer Material: " + layerMaterial, "Revit");
    }
}
}

```

9.7 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.

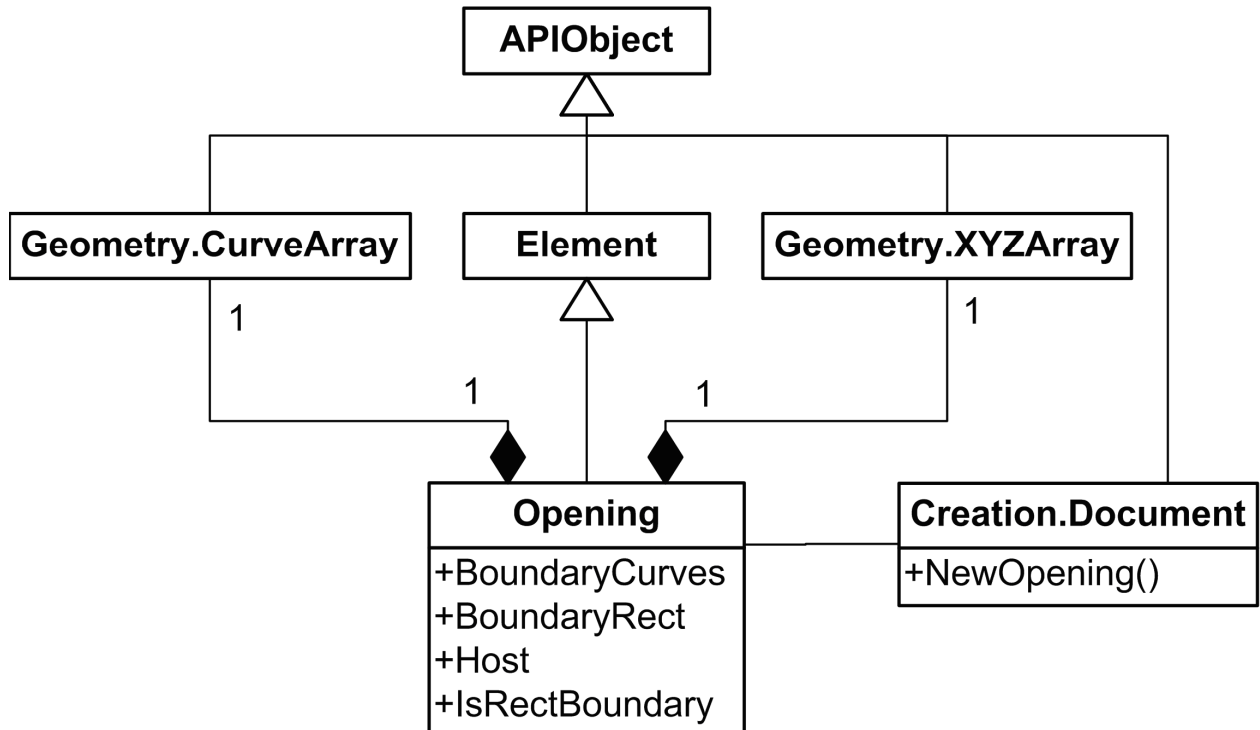


Figure 49: Opening diagram

9.7.1 General Properties

This section explains how to use the Opening properties.

- **IsRectBoundary** - Identifies whether the opening has a rectangular boundary.
 - If true, it means the Opening has a rectangular boundary and you can get an XYZArray object from the Opening BoundaryRect property. Otherwise, the property returns null.
 - If false, you can get a CurveArray object from the BoundaryCurves property.
- **BoundaryCurves** - If the opening boundary is not a rectangle, this property retrieves geometry information; otherwise it returns null. The property returns a CurveArray object containing the curves that represent the Opening object boundary.

For more details about Curve, refer to the [Geometry](#) chapter.

- **BoundaryRect** - If the opening boundary is a rectangle, you can get the geometry information using this property; otherwise it returns null.
 - The property returns an XYZArray object containing the XYZ coordinates.
 - The XYZArray usually contains the rectangle boundary minimum (lower left) and the maximum (upper right) coordinates.
- **Host** - The host property retrieves the Opening host element. The host element is the element cut by the Opening object.

Note: If the Opening object's category is Shaft Openings, the Opening host is null.

The following example illustrates how to retrieve the existing Opening properties.

Code Region 9-6: Retrieving existing opening properties

```
private void Getinfo_Opening(Opening opening)
{
    string message = "Opening:";
    //get the host element of this opening
    message += "\nThe id of the opening's host element is : " + opening.Host.Id.Value;

    //get the information whether the opening has a rect boundary
    //If the opening has a rect boundary, get the geom info from BoundaryRect property.
    //Otherwise we should get the geometry information from BoundaryCurves property
    if (opening.IsRectBoundary)
    {
        message += "\nThe opening has a rectangular boundary.";
        //array contains two XYZ objects: the max and min coords of boundary
        XYZArray boundaryRect = opening.BoundaryRect;

        //get the coordinate value of the min coordinate point
        XYZ point = opening.BoundaryRect.get_Item(0);
        message += "\nMin coordinate point: (" + point.X + ", "
            + point.Y + ", " + point.Z + ")";

        //get the coordinate value of the Max coordinate point
        point = opening.BoundaryRect.get_Item(1);
        message += "\nMax coordinate point: (" + point.X + ", "
            + point.Y + ", " + point.Z + ")";
    }
    else
    {
        message += "\nThe opening doesn't have a rectangular boundary.";
        // Get curve number
        int curves = opening.BoundaryCurves.Size;
        message += "\nNumber of curves is : " + curves;
        for (int i = 0; i < curves; i++)
        {
            Autodesk.Revit.Geometry.Curve curve = opening.BoundaryCurves.get_Item(i);
            // Get curve start and end points
            message += "\nCurve start point: " + XYZToString(curve.get_EndPoint(0));
            message += "; Curve end point: " + XYZToString(curve.get_EndPoint(1));
        }
    }
    MessageBox.Show(message, "Revit");
}

// output the point's three coordinates
string XYZToString(XYZ point)
{
    return "(" + point.X + ", " + point.Y + ", " + point.Z + ")";
}
```

9.7.2 Create Opening

In the Revit Platform API, use the `Document.NewOpening()` method to create an opening in your project. There are four method overloads you can use to create openings in different host elements:

Code Region 9-7: NewOpening()

```
//Create a new Opening in a beam, brace and column.
public Opening NewOpening(Element famInstElement, CurveArray profile, eRefFace iFace);

//Create a new Opening in a roof, floor and ceiling.
public Opening NewOpening(Element hostElement, CurveArray profile, bool bPerpendicularFace);

//Create a new Opening Element.
public Opening NewOpening(Level bottomLevel, Level topLevel, CurveArray profile);

//Create an opening in a straight wall or arc wall.
public Opening NewOpening(Wall, XYZ pntStart, XYZ pntEnd);
```

- Create an Opening in a Beam, Brace, or Column - Use to create an opening in a family instance. The `iFace` parameter indicates the face on which the opening is placed.
- Create a Roof, Floor, or Ceiling Opening - Use to create an opening in a roof, floor, or ceiling.
- The `bPerpendicularFace` parameter indicates whether the opening is perpendicular to the face or vertical.
- If the parameter is true, the opening is perpendicular to the host element face. See the following picture:

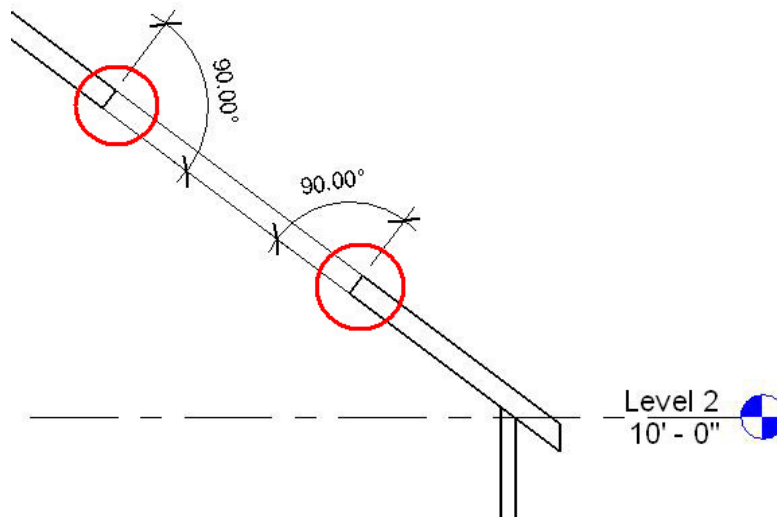


Figure 50: Opening cut perpendicular to the host element face

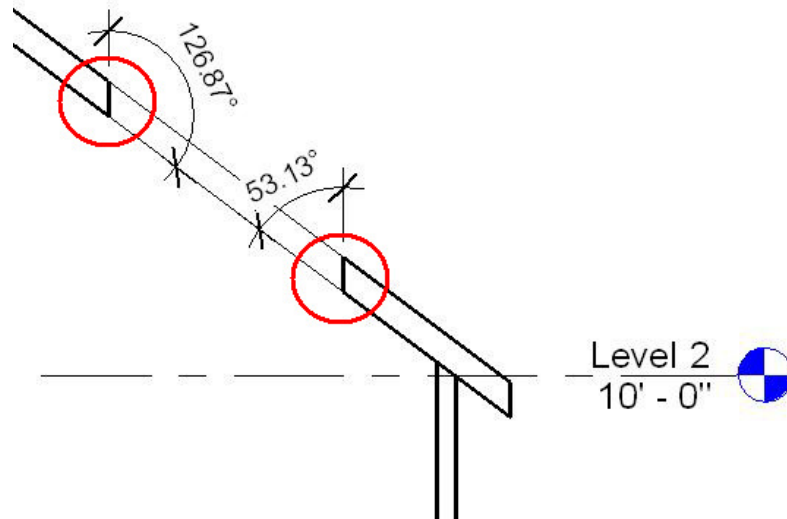


Figure 51: Opening cut vertically to the host element

- 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.

10 Family Instances

In this chapter, you will learn about the following:

- The relationship between family and family instance
- Family and family instance features
- How to load or create family and family instance features.

10.1 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.

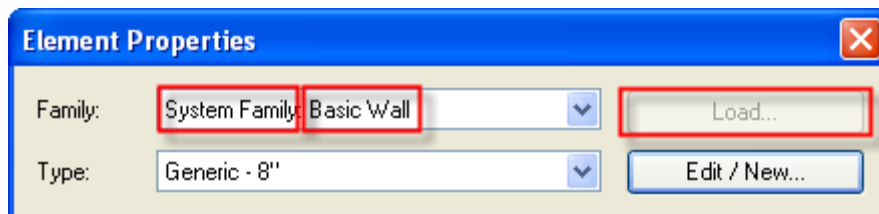


Figure 52: System Family

- 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.

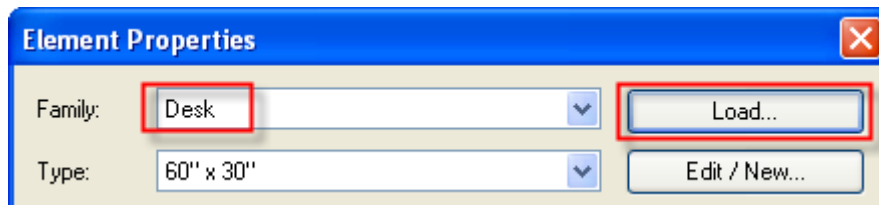


Figure 53: Component Family

- There are some exceptions, for example: Mass and in-place member. The Family and Type fields are blank.

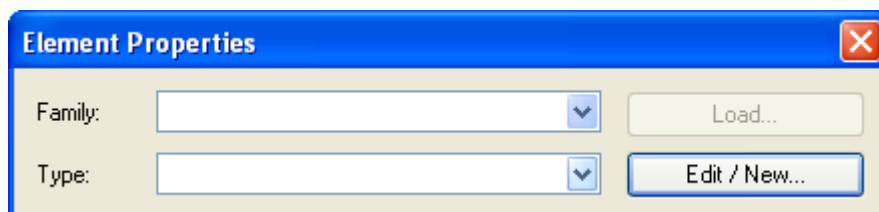


Figure 54: Mass or in-place member example

Family Instances

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

- Family
- FamilySymbol
- FamilyInstance

Each object plays a significant role in the family structure.

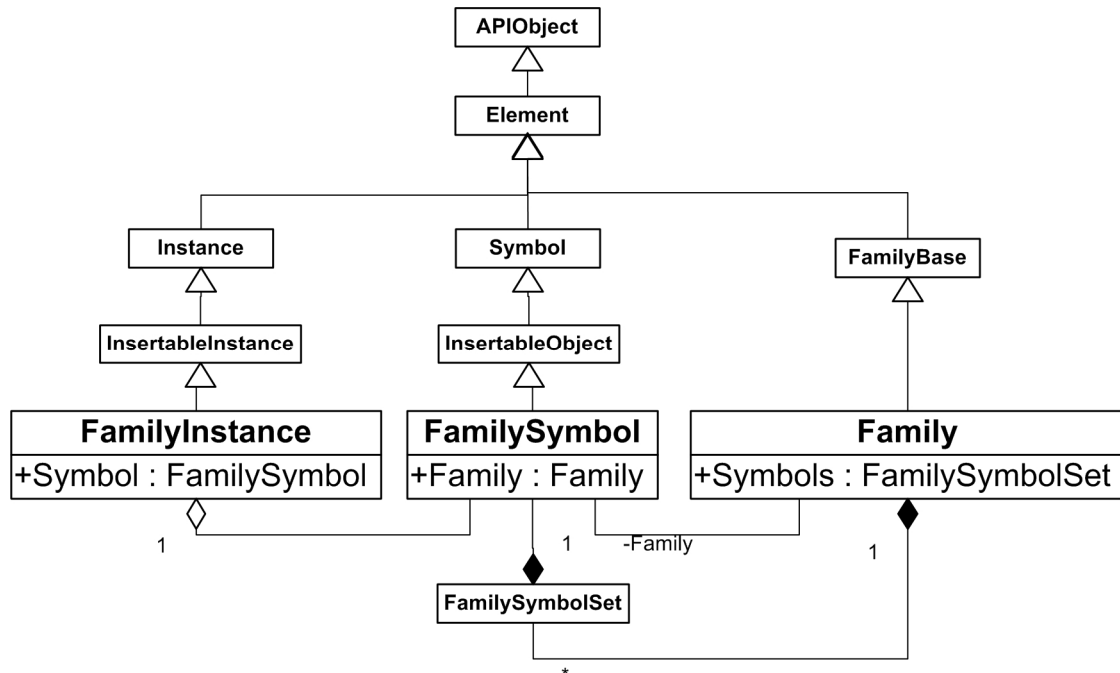


Figure 55: Family-related class diagram

The Family object represents an entire family such as a Single-Flush doors. For example, the Single-Flush door Family corresponds to the Single-Flush.rfa file. The Family object contains several FamilySymbols that are used to get all family symbols to facilitate swapping instances from one symbol to another.

The FamilySymbol object represents a specific set of family settings corresponding to a Type in the Revit UI, such as 34"X80".

The FamilyInstance object represents an actual Type (FamilySymbol) instance in the Revit project. For example, in the following picture, the FamilyInstance is a single door in the project.

- Each FamilyInstance has one FamilySymbol. The door is an instance of a 34"X80".
- Each FamilySymbol belongs to one Family. The 34"X80" symbol belongs to a Single-Flush family.
- Each Family contains one or more FamilySymbols. The Single-Flush family contains a 34"X80" symbol, a 34"X84" symbol, a 36"X84" and so on.

Note that while most component elements are exposed through the API classes FamilySymbol and FamilyInstance, some have been wrapped with specific API classes. For example, AnnotationSymbolType wraps FamilySymbol and AnnotationSymbol wraps FamilyInstance.

10.2 Family

The Family class represents an entire Revit family. It contains the FamilySymbols used by FamilyInstances.

10.2.1 Loading Families

The Document class contains the LoadFamily() and LoadFamilySymbol() methods.

- LoadFamily() loads an entire family and all of its types or symbols into the project.
- LoadFamilySymbol() loads only the specified family symbol from a family file into the project.

Note: To improve the performance of your application and reduce memory usage, if possible load specific FamilySymbols instead of entire Family objects.

- The family file path is retrieved using the Options.Application object LibraryPaths property.
- The Options.Application object is retrieved using the Application object Options property.
- In LoadFamilySymbol(), the input argument Name is the same string value returned by the FamilySymbol object Name property.

For more information, refer to the [Code Samples](#) in this chapter.

10.2.2 Categories

The FamilyBase.FamilyCategory property indicates the category of the Family such as Columns, Furniture, Structural Framing, or Windows.

10.3 FamilyInstances

Examples of categories of FamilyInstance objects in Revit are Beams, Braces, Columns, Furniture, Massing, and so on. The FamilyInstance object provides more detailed properties so that the family instance type and appearance in the project can be changed.

10.3.1 Location-Related Properties

Location-related properties show the physical and geometric characteristics of FamilyInstance objects, such as orientation, rotation and location.

10.3.1.1 Orientation

The face orientation or hand orientation can be changed for some FamilyInstance objects. For example, a door can face the outside or the inside of a room or wall and it can be placed with the handle on the left side or the right side. The following table compares door, window, and desk family instances.

Boolean Property	Door	Window (Fixed: 36" w x 72" h)	Desk
CanFlipFacing	True	True	False
CanFlipHand	True	False	False

Table 25: Compare Family Instances

If CanFlipFacing or CanFlipHand is true, you can call the flipFacing() or flipHand() methods respectively. These methods can change the facing orientation or hand orientation respectively. Otherwise, the methods do nothing and return False.

When changing orientation, remember that some types of windows can change both hand orientation and facing orientation, such as a Casement 3x3 with Trim family.

There are four different facing orientation and hand orientation combinations for doors. See the following picture for the combinations and the corresponding Boolean values are in the following table.

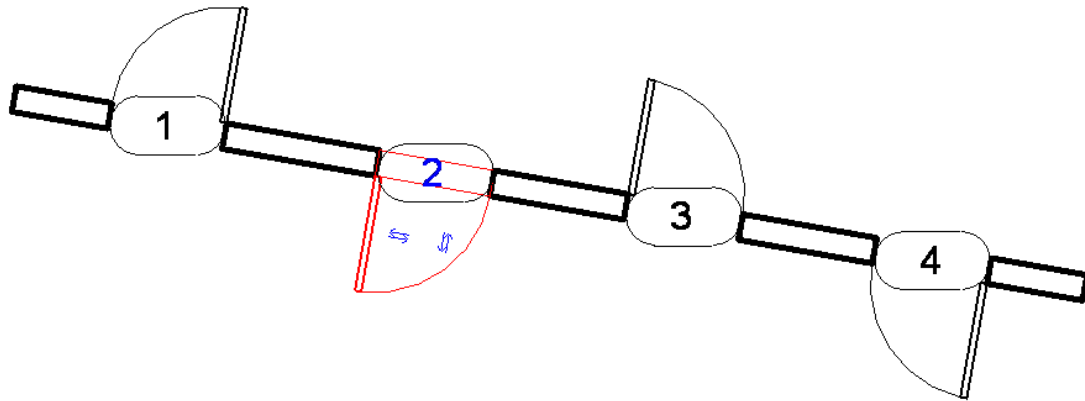


Figure 56: Doors with different Facing and Hand Orientations

Boolean Property	Door 1	Door 2	Door 3	Door 4
FacingFlipped	False	True	False	True
HandFlipped	False	True	True	False

Table 26: Different Instances of the Same Type

10.3.1.2 Rotation - Mirrored

The Mirrored property indicates whether the FamilyInstance object has been mirrored.

Boolean Property	Door 1	Door 2	Door 3	Door 4
Mirrored	False	False	True	True

Table 27: Door Mirrored Property

In the previous door example, the Mirrored property for Door 1 and Door 2 is False, while for both Door 3 and Door 4 it is True. This is because when you create a door in the Revit project, the default result is either Door 1 or Door 2. To create a door like Door 3 or Door 4, you must flip the Door 1 and Door 2 hand orientation respectively. The flip operation is like a mirror transformation, which is why the Door 3 and Door 4 Mirrored property is True.

For more information about using the Mirror() method in Revit, refer to the [Editing Elements](#) chapter.

10.3.1.3 Rotation – CanRotate and rotate()

The family instance Boolean CanRotate property is used to test whether the family instance can be rotated 180 degrees. This depends on the family to which the instance belongs. For example, in the following picture, the CanRotate properties for Window 1 (Casement 3x3 with Trim: 36"x72") and Door 1 (Double-Glass 2: 72"x82") are true, while Window 2 (Fixed: 36"w x 72"h) is false.

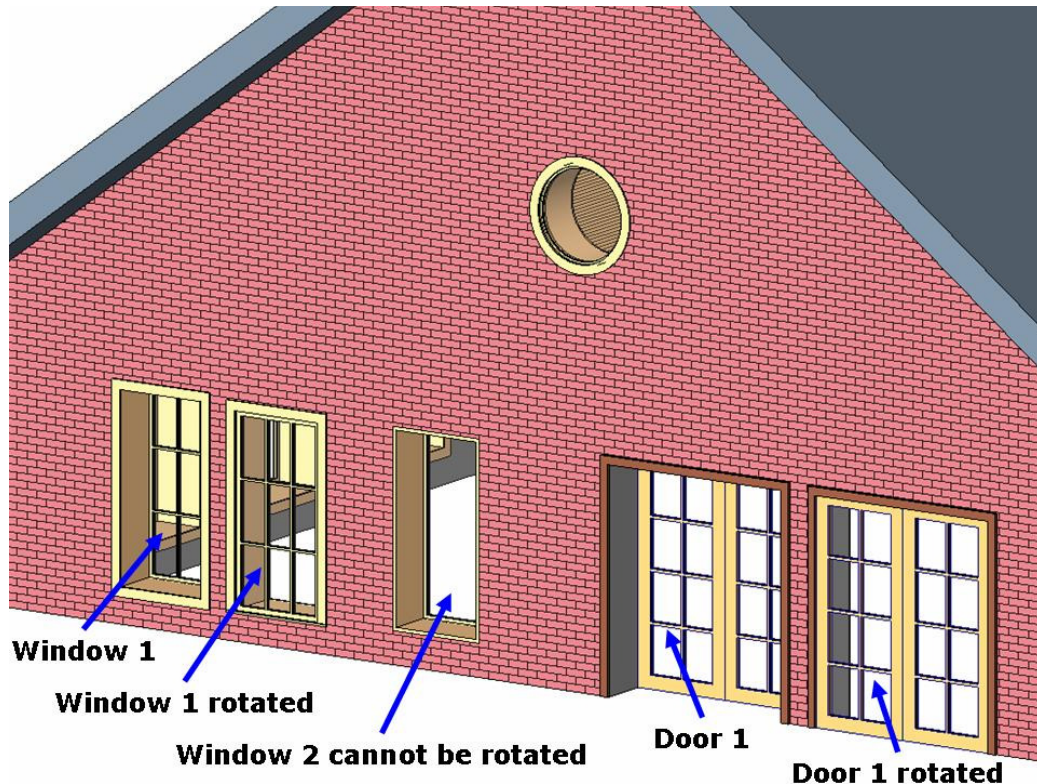


Figure 57: Changes after rotate()

If CanRotate is true, you can call the family instance rotate() method, which flips the family instance by 180 degrees. Otherwise, the method does nothing and returns False. The previous picture also shows the Window 1 and Door 1 states after executing the rotate() method.

Recall from the [Rotating elements](#) section earlier in this document, that family instances (and other elements) can be rotated a user-specified angle using Document.Rotate().

10.3.1.4 Location

The Location property determines the physical location of an instance in a project. An instance can have a point location or a line location.

The following characteristics apply to Location:

- A point location is a LocationPoint class object - A footing, a door, or a table has a point location
- A line location is a LocationCurve class object - A beam has a line location.
- They are both subclasses of the Location class.

For more information about Location, refer to the [Editing Elements](#) chapter.

10.3.2 Host and Component

Host and Component are both FamilyInstance properties. Component can be further divided into Subcomponent and Supercomponent.

10.3.2.1 Host

A FamilyInstance object has a Host property that returns its hosting element.

Some FamilyInstance objects do not have host elements, such as Tables and other furniture, so the Host property returns nothing because no host elements are created. However, other objects, such as doors and windows, must have host elements. In this case the Host property returns a wall Element in which the window or the door is located. See the following picture.

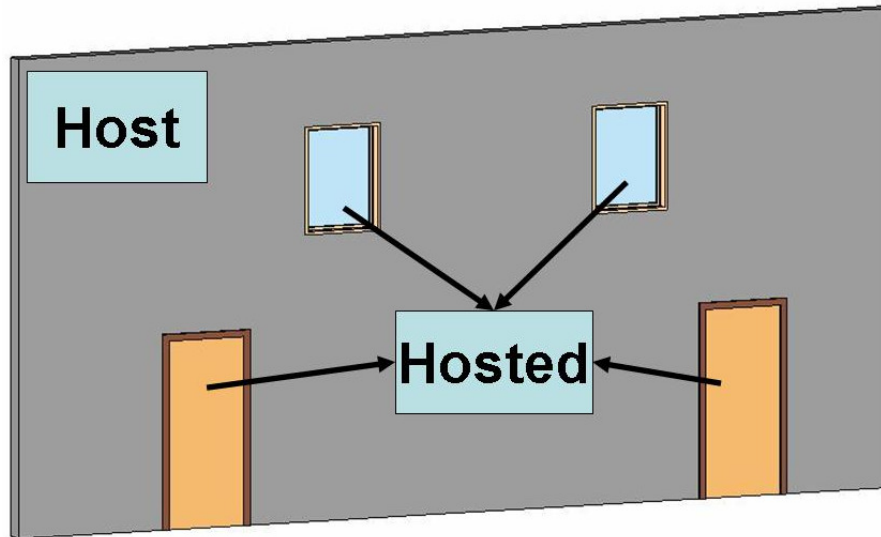


Figure 58: Doors and windows hosted in a wall

10.3.3 Subcomponent and Supercomponent

The `FamilyInstance.SubComponents` property returns the set of family instances loaded into that family. When an instance of 'Table-Dining Round w Chairs.rfa' is placed in a project, the set of chairs are returned by the `SubComponent` property.

The `SuperComponent` property returns the family instance's parent component. In 'Table-Dining Round w Chairs.rfa', the family instance supercomponent for each nested chair is the instance of 'Table-Dining Round w Chairs.rfa'.

Code Region 10-1: Getting SubComponents and SuperComponent from FamilyInstance

```
public void GetSubAndSuperComponents(FamilyInstance familyInstance)
{
    ElementSet subElemSet = familyInstance.SubComponents;
    if (subElemSet != null)
    {
        string subElems = "";
        foreach (Autodesk.Revit.Element ee in subElemSet)
        {
            FamilyInstance f = ee as FamilyInstance;
            subElems = subElems + f.Name + "\n";
        }
        MessageBox.Show("Subcomponent count = " + subElemSet.Size + "\n" + subElems);
    }
    FamilyInstance super = familyInstance.SuperComponent as FamilyInstance;
    if (super != null)
    {
        MessageBox.Show("SUPER component: " + super.Name);
    }
}
```

10.3.4 Other Properties

The properties in this section are specific to Revit Architecture and Revit Structure. They are covered thoroughly in their respective chapters.

10.3.4.1 Room Information

FamilyInstance properties include Room, FromRoom, and ToRoom. For more information about Room, refer to the [Revit Architecture](#) chapter.

10.3.4.2 Space Information

FamilyInstance has a Space property for identifying the space that holds an instance in MEP.

10.3.4.3 Revit Structure Related Analytical Model

The AnalyticalModel property retrieves the family instance structural analytical model. Different analytical models are returned based on the instance type.

For example, if the instance is a beam, a frame analytical model is returned. If the instance is a footing, a location analytical model is returned. For more information about AnalyticalModel refer to the [Revit Structure](#) chapter.

10.3.5 Creating FamilyInstance Objects

Typically a FamilyInstance object is created using one of the eight overload methods of Autodesk.Revit.Creation.Document called NewFamilyInstance(). The choice of which overload to use depends not only on the category of the instance, but also other characteristics of the placement like whether it should be hosted, placed relative to a reference level, or placed directly on a particular face. The details are included in Table 28 - 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 29 - Options for creating instances with other methods below). If the instance is not created, an exception is thrown. The type/symbol used must be loaded into the project before the method is called.

Category	NewFamilyInstance() parameters	Comments
Air Terminals Boundary Conditions Casework Communication Devices Data Devices Electrical Equipment Electrical Fixtures Entourage Fire Alarm Devices Furniture	XYZ, FamilySymbol, StructuralType	Creates the instance in an arbitrary location without reference to a level or host element.
	XYZ, FamilySymbol, Element, StructuralType	If it is to be hosted on a wall, floor or ceiling
	XYZ, FamilySymbol, XYZ, Element, StructuralType	If it is to be hosted on a wall, floor, or ceiling, and needs to be oriented in a non-default direction

Family Instances

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	XYZ, FamilySymbol, Element, Level, StructuralType	If it is to be hosted on a wall, floor or ceiling and associated to a reference level
	XYZ, FamilySymbol, Level, StructuralType	If it is to be associated to a reference level
	Face, XYZ, XYZ, FamilySymbol	If it is face-based and needs to be oriented in a non-default direction
	Face, Line, FamilySymbol	If it is face-based and linear
Columns Structural Columns	XYZ, FamilySymbol, Level, StructuralType	Creates the column so that its base lies on the reference level. The column will extend to the next available level in the model, or will extend the default column height if there are no suitable levels above the reference level.
Doors Windows	XYZ, FamilySymbol, Element, StructuralType	Doors and windows must be hosted by a wall. Use this method if they can be placed with the default orientation.
	XYZ, FamilySymbol, XYZ, Element, StructuralType	If the created instance needs to be oriented in a non-default direction
	XYZ, FamilySymbol, Element, Level, StructuralType	If the instance needs to be associated to a reference level
Structural Framing (Beams, Braces)	Curve, FamilySymbol, Level, StructuralType	Creates a level based brace or beam given its curve. This is the recommended method to create Beams and Braces
	XYZ, FamilySymbol, StructuralType	Creates instance in an arbitrary location ¹
	XYZ, FamilySymbol, Element, Level, StructuralType	If it is hosted on an element (floor etc.) and associated to a reference level ¹

	XYZ, FamilySymbol, Level, StructuralType	If it is associated to a reference level ¹
	XYZ, FamilySymbol, Element, StructuralType	If it is hosted on an element (floor etc.) ¹

Table 28 - Options for creating instance with NewFamilyInstance()

¹ The structural instance will be of zero-length after creation. Extend it by setting its curve (FamilyInstance.Location as LocationCurve) using LocationCurve.Curve property.

You can simplify your code and improve performance by creating more than one family instance at a time using Document.NewFamilyInstances(). This method has a single parameter, which is a list of FamilyInstanceCreationData objects describing the family instances to create.

Code Region 10-2: Batch creating family instances

```

ElementSet BatchCreateColumns(Autodesk.Revit.Document document, Level level)
{
    List<FamilyInstanceCreationData> fiCreationDatas =
        new List<FamilyInstanceCreationData>();

    ElementSet elementSet = null;

    //Try to get a FamilySymbol
    FamilySymbol familySymbol = null;
    ElementIterator iter = document.get_Elements(typeof(FamilySymbol));
    iter.Reset();
    while (iter.MoveNext())
    {
        familySymbol = iter.Current as FamilySymbol;
        if (null != familySymbol && null != familySymbol.Category)
        {
            if ("Structural Columns" == familySymbol.Category.Name)
            {
                break;
            }
        }
    }

    if (null != familySymbol)
    {
        //Create 10 FamilyInstanceCreationData items for batch creation
        for (int i = 1; i < 11; i++)
        {
            XYZ location = new XYZ(i * 10, 100, 0);
            FamilyInstanceCreationData fiCreationData =
                new FamilyInstanceCreationData(location, familySymbol, level,
                    Autodesk.Revit.Structural.Enums.StructuralType.Column);
        }
    }
}

```


Family Instances

```

        if (null != fiCreationData)
        {
            fiCreationDatas.Add(fiCreationData);
        }
    }

    if (fiCreationDatas.Count > 0)
    {
        // Create Columns
        elementSet = document.Create.NewFamilyInstances(fiCreationDatas);
    }
    else
    {
        throw new Exception("Batch creation failed.");
    }
}
else
{
    throw new Exception("No column types found.");
}

return elementSet;
}

```

Instances of some family types are better created through methods other than Autodesk.Revit.Creation.Document.NewFamilyInstance(). These are listed in the table below.

Category	Creation method	Comments
Air Terminal Tags Area Load Tags Area Tags Casework Tags Ceiling Tags Communication Device Tags Curtain Panel Tags Data Device Tags Detail Item Tags Door Tags Duct Accessory Tags Duct Fitting Tags Duct Tags Electrical Equipment Tags Electrical Fixture Tags	NewTag(View, Element, Boolean, TagMode, TagOrientation, XYZ)	TagMode should be TM_ADDBY_CATEGORY and there should be a related tag family loaded when try to create a tag, otherwise exception will be thrown

Family Instances

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	NewTag(View, Element, Boolean, TagMode, TagOrientation, XYZ)	TagMode should be TM_ADDBY_MATERIAL and there should be a material tag family loaded, otherwise exception will be thrown
Multi-Category Tags	NewTag(View, Element, Boolean, TagMode, TagOrientation, XYZ)	TagMode should be TM_ADDBY_MULTICATEGORY, and there should be a multi-category tag family loaded, otherwise exception will be thrown
Generic Annotations	NewAnnotationSymbol(XYZ, AnnotationSymbolType, View)	
Title Blocks	NewViewSheet(FamilySymbol)	The titleblock will be added to the newly created sheet.

Table 29 - Options for creating instances with other methods

Families and family symbols are loaded using the Document.LoadFamily() or Document.LoadFamilySymbol() methods. Some families, such as Beams, have more than one endpoint and are inserted in the same way as a single point instance. Once the linear family instances are inserted, their endpoints can be changed using the Element.Location property. For more information, refer to the [Code Samples](#) section in this chapter.

10.4 Code Samples

Review the following code samples for more information about working with Family Instances. Please note that in the NewFamilyInstance() method, a StructuralType argument is required to specify the type of the family instance to be created. Here are some examples:

Type of Family Instance	Value of StructuralType
Doors, tables, etc.	NonStructural
Beams	Beam
Braces	Brace
Columns	Column
Footings	Footing

Table 30: The value of StructuralType argument in the NewFamilyInstance() method

10.4.1 Create Tables

The following function demonstrates how to load a family of Tables into a Revit project and create instances from all symbols in this family.

The LoadFamily() method returns false if the specified family was previously loaded. Therefore, in the following case, do not load the family, Table-Dining Round w Chairs.rfa, before this function is called. In this example, the tables are created at Level 1 by default.

Code Region 10-3: Creating tables

```
String fileName = @"C:\Documents and Settings\All Users\Application Data\Autodesk\RST
2010\Imperial Library\Furniture\Table-Dining Round w Chairs.rfa";

// try to load family
Family family = null;
if (!document.LoadFamily(fileName, out family))
{
    throw new Exception("Unable to load " + fileName);
}

// Loop through table symbols and add a new table for each
FamilySymbolSetIterator symbolItr = family.Symbols.ForwardIterator();
double x = 0.0, y = 0.0;
while (symbolItr.MoveNext())
{
    FamilySymbol symbol = symbolItr.Current as FamilySymbol;
    XYZ location = new XYZ(x, y, 10.0);
    // Do not use the overloaded NewFamilyInstance() method that contains
    // the Level argument, otherwise Revit cannot show the instances
    // correctly in 3D View, for the table is not level-based component.
    FamilyInstance instance = document.Create.NewFamilyInstance(location, symbol,
                                                                    StructuralType.NonStructural);

    x += 10.0;
}
```

The result of loading the Tables family and placing one instance of each FamilySymbol:

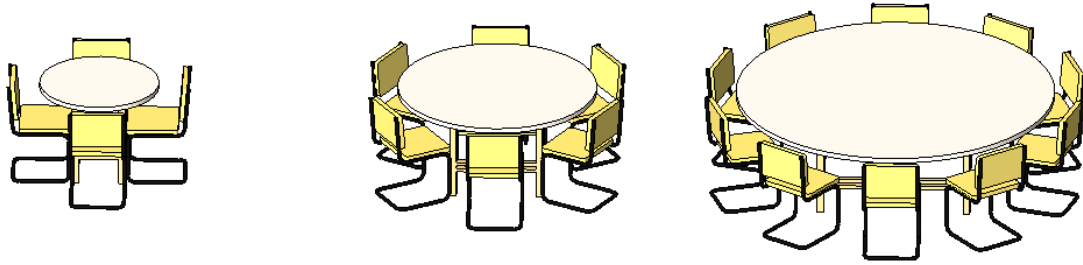


Figure 59: Load family and create tables in the Revit project

10.4.2 Create a Beam

In this sample, a family symbol is loaded instead of a family, because loading a single FamilySymbol is faster than loading a Family that contains many FamilySymbols.

Code Region 10-4: Creating a beam

```
// get the active view's level for beam creation
Level level = document.Application.ActiveDocument.ActiveView.Level;

// load a family symbol from file
FamilySymbol gotSymbol = null;
String fileName = @"C:\Documents and Settings\All Users\Application Data\Autodesk\RST
2010\Imperial Library\Structural\Framing\Steel\W-Wide Flange.rfa";
String name = "W10X54";

FamilyInstance instance = null;

if (document.LoadFamilySymbol(fileName, name, out gotSymbol))
{
    // look for a model line in the list of selected elements
    ElementSet sel = document.Selection.Elements;
    ModelLine modelLine = null;
    foreach (Autodesk.Revit.Element elem in sel)
    {
        if (elem is ModelLine)
        {
            modelLine = elem as ModelLine;
            break;
        }
    }
    if (null != modelLine)
    {
        // create a new beam
        instance = document.Create.NewFamilyInstance(modelLine.GeometryCurve,
                                                    gotSymbol, level, StructuralType.Beam);
    }
    else
    {
        throw new Exception("Please select a model line before invoking this command");
    }
}
```

```

}
else
{
    throw new Exception("Couldn't load " + fileName);
}

```

10.4.3 Create Doors

Create a long wall about 180' in length and select it before running this sample. The host object must support inserting instances; otherwise the `NewFamilyInstance()` method will fail. If a host element is not provided for an instance that must be created in a host, or the instance cannot be inserted into the specified host element, the method `NewFamilyInstance()` does nothing.

Code Region 10-5: Creating doors

```

void CreateDoorsInWall(Autodesk.Revit.Document document, Wall wall)
{
    String fileName = @"C:\Documents and Settings\All Users\Application Data\Autodesk\RST
2010\Imperial Library\Doors\Single-Decorative 2.rfa";

    Family family = null;
    if (!document.LoadFamily(fileName, out family))
    {
        throw new Exception("Unable to load " + fileName);
    }

    // get the active view's level for beam creation
    Level level = document.Application.ActiveDocument.ActiveView.Level;

    FamilySymbolSetIterator symbolItr = family.Symbols.ForwardIterator();
    double x = 0, y = 0, z = 0;
    while (symbolItr.MoveNext())
    {
        FamilySymbol symbol = symbolItr.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.

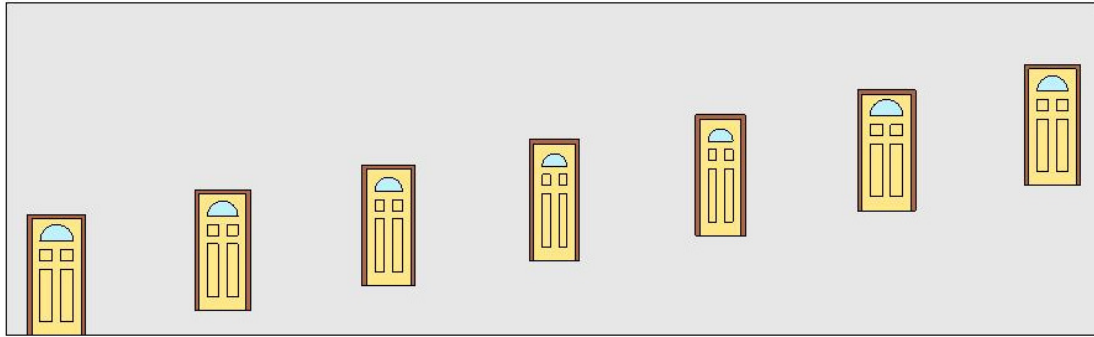


Figure 60: Insert doors into a wall

10.4.4 Create FamilyInstances Using Reference Directions

Use reference directions to insert an item in a specific direction.

Code Region 10-6: Creating FamilyInstances using reference directions

```
String fileName = @"C:\Documents and Settings\All Users\Application Data\Autodesk\RST
2010\Imperial Library\Furniture\Bed-Box.rfa";

Autodesk.Revit.Elements.Family family = null;
if (!document.LoadFamily(fileName, out family))
{
    throw new Exception("Couldn't load " + fileName);
}

Floor floor = null;

ElementIterator elemItr = document.get_Elements(typeof(Floor));
elemItr.Reset();
while (elemItr.MoveNext())
{
    Floor currentFloor = elemItr.Current as Floor;
    if (null != currentFloor)
    {
        floor = currentFloor;
        break;
    }
}

if (null != floor)
{
    FamilySymbolSetIterator symbolItr = family.Symbols.ForwardIterator();
    int x = 0, y = 0;
    int i = 0;
    while (symbolItr.MoveNext())
    {
        FamilySymbol symbol = symbolItr.Current as FamilySymbol;
        XYZ location = new XYZ(x, y, 0);
        XYZ direction = new XYZ();
```

Family Instances

```
switch (i % 3)
{
    case 0:
        direction = new XYZ(1, 1, 0);
        break;
    case 1:
        direction = new XYZ(0, 1, 1);
        break;
    case 2:
        direction = new XYZ(1, 0, 1);
        break;
}
FamilyInstance instance = document.Create.NewFamilyInstance(location, symbol,
                                                             direction, floor, StructuralType.NonStructural);

x += 10;
i++;
}
}
else
{
    throw new Exception("Please open a model with at least one floor element before invoking
this command.");
}
```

The result of the previous code appears in the following picture:

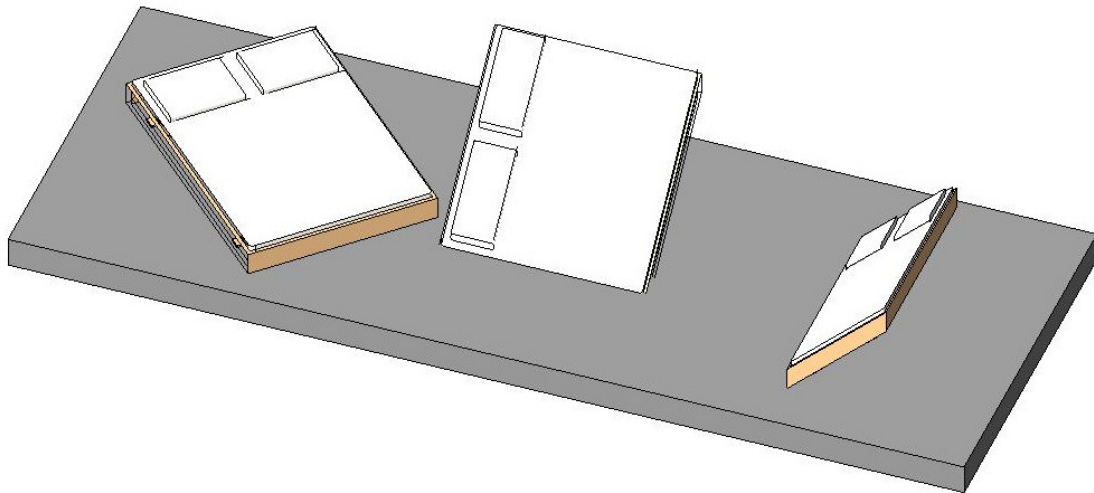


Figure 61: Create family instances using different reference directions

11 Family Creation

This chapter discusses families and how to:

- Create and modify Family documents
- Access family types and parameters

11.1 Family Documents

11.1.1 Family

The Family object represents an entire Revit family. A Family Document is a Document that represents a Family rather than a Revit project.

Using the Family Creation functionality of the Revit API, you can create and edit families and their types. This functionality is particularly useful when you have pre-existing data available from an external system that you want to convert to a Revit family library.

API access to system family editing is not available.

11.1.2 Categories

As noted in the previous chapter, the `FamilyBase.FamilyCategory` property indicates the category of the Family such as Columns, Furniture, Structural Framing, or Windows.

The following code can be used to determine the category of the family in an open Revit Family document.

Code Region 11-1: Category of open Revit Family Document

```
string categoryName = familyDoc.OwnerFamily.FamilyCategory.Name;
```

11.1.3 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 11-2: Creating a new Family document

```
// create a new family document using Generic Model.rft template
string templateFileName = @"C:\Documents and Settings\All Users\Application Data\Autodesk\RST
2010\Imperial Templates\Generic Model.rft";

Document familyDocument = application.NewFamilyDocument(templateFileName);
if (null == familyDocument)
{
    throw new Exception("Cannot open family document");
}
```

11.1.4 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 11-3: Getting nested Family symbols in a Family

```
public void GetLoadedSymbols(Autodesk.Revit.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";

            ElementIterator itrFamilySymbols =
                familyDoc.get_Elements(typeof(FamilySymbol));

            itrFamilySymbols.Reset();

            while (itrFamilySymbols.MoveNext())
            {
                FamilySymbol fs = itrFamilySymbols.Current as FamilySymbol;

                loadedFamilies += "\t" + fs.Name + "\n";
            }

            MessageBox.Show(loadedFamilies, "Revit");
        }
    }
}
```

11.2 Family Item Factory

The FamilyItemFactory class provides the ability to create elements in family documents. It is accessed through the Document.FamilyCreate property. FamilyItemFactory is derived from the ItemFactoryBase class which is a utility to create elements in both Revit project documents and family documents.

11.2.1 Creating Forms

The FamilyItemFactory provides the ability to create form elements in families, such as extrusions, revolutions, sweeps, and blends. See section 15.2 for more information on these 3D sketch forms.

The following example demonstrates how to create a new Extrusion element. It creates a simple rectangular profile and then moves the newly created Extrusion to a new location.

Code Region 11-4: Creating a new Extrusion

```

private Extrusion CreateExtrusion(Autodesk.Revit.Document document, SketchPlane sketchPlane)
{
    Extrusion rectExtrusion = null;
    // make sure we have a family document
    if (true == document.IsFamilyDocument)
    {
        // define the profile for the extrusion
        CurveArrArray curveArrArray = new CurveArrArray();
        CurveArray curveArray1 = new CurveArray();
        CurveArray curveArray2 = new CurveArray();
        CurveArray curveArray3 = new CurveArray();
        // create a rectangular profile
        XYZ p0 = XYZ.Zero;
        XYZ p1 = new XYZ(10, 0, 0);
        XYZ p2 = new XYZ(10, 10, 0);
        XYZ p3 = new XYZ(0, 10, 0);
        Line line1 = document.Application.Create.NewLineBound(p0, p1);
        Line line2 = document.Application.Create.NewLineBound(p1, p2);
        Line line3 = document.Application.Create.NewLineBound(p2, p3);
        Line line4 = document.Application.Create.NewLineBound(p3, p0);
        curveArray1.Append(line1);
        curveArray1.Append(line2);
        curveArray1.Append(line3);
        curveArray1.Append(line4);
        curveArrArray.Append(curveArray1);
        // create solid rectangular extrusion
        rectExtrusion = document.FamilyCreate.NewExtrusion(true, curveArrArray,
                                                            sketchPlane, 10);

        if (null != rectExtrusion)
        {
            // move extrusion to proper place
            XYZ transPoint1 = new XYZ(-16, 0, 0);
            document.Move(rectExtrusion, transPoint1);
        }
        else
        {
            throw new Exception("Create new Extrusion failed.");
        }
    }
    else
    {
        throw new Exception("Please open a Family document before invoking this command.");
    }
    return rectExtrusion;
}

```

The following sample shows how to create a new Sweep from a solid ovoid profile in a Family Document.

Code Region 11-5: Creating a new Sweep

```
private Sweep CreateSweep(Autodesk.Revit.Document document, SketchPlane sketchPlane)
{
    Sweep sweep = null;
    // make sure we have a family document
    if (true == document.IsFamilyDocument)
    {
        // Define a profile for the sweep
        CurveArrArray arrarr = new CurveArrArray();
        CurveArray arr = new CurveArray();
        // Create an ovoid profile
        XYZ pnt1 = new XYZ(0, 0, 0);
        XYZ pnt2 = new XYZ(2, 0, 0);
        XYZ pnt3 = new XYZ(1, 1, 0);
        arr.Append(document.Application.Create.NewArc(pnt2, 1.0d, 0.0d, 180.0d, XYZ.BasisX,
                                                       XYZ.BasisY));
        arr.Append(document.Application.Create.NewArc(pnt1, pnt3, pnt2));
        arrarr.Append(arr);
        SweepProfile profile = document.Application.Create.NewCurveLoopsProfile(arrarr);
        // Create a path for the sweep
        XYZ pnt4 = new XYZ(10, 0, 0);
        XYZ pnt5 = new XYZ(0, 10, 0);
        Curve curve = document.Application.Create.NewLineBound(pnt4, pnt5);
        CurveArray curves = new CurveArray();
        curves.Append(curve);
        // create a solid ovoid sweep
        sweep = document.FamilyCreate.NewSweep(true, curves, sketchPlane, profile, 0,
                                                ProfilePlaneLocation.Start);

        if (null != sweep)
        {
            // move to proper place
            XYZ transPoint1 = new XYZ(11, 0, 0);
            document.Move(sweep, transPoint1);
        }
        else
        {
            throw new Exception("Failed to create a new Sweep.");
        }
    }
    else
    {
        throw new Exception("Please open a Family document before invoking this command.");
    }
    return sweep;
}
```

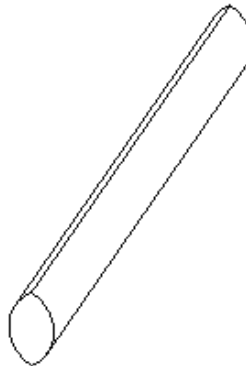


Figure 62: Ovoid sweep created by previous example

11.2.1.1 Assigning Subcategories to forms

After creating a new form in a family, you may want to change the subcategory for the form. For example, you may have a Door family and want to create multiple subcategories of doors and assign different subcategories to different door types in your family.

The following example shows how to create a new subcategory, assign it a material, and then assign the subcategory to a form.

Code Region 11-6: Assigning a subcategory

```
public void AssignSubCategory(Document document, GenericForm extrusion)
{
    // create a new subcategory
    Category cat = document.OwnerFamily.FamilyCategory;
    Category subCat = document.Settings.Categories.NewSubcategory(cat, "NewSubCat");

    // create a new material and assign it to the subcategory
    MaterialWood woodMaterial = document.Settings.Materials.AddWood("Wood Material");
    subCat.Material = woodMaterial;

    // assign the subcategory to the element
    extrusion.Subcategory = subCat;
}
```

11.2.2 Creating Annotations

New annotations such as Dimensions and ModelText and TextNote objects can also be created in families, as well as curve annotation elements such as SymbolicCurve, ModelCurve, and DetailCurve. See Chapter 14 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 11-7: Creating a Dimension

```

public Dimension CreateArcDimension(Document document, SketchPlane sketchPlane)
{
    Autodesk.Revit.Creation.Application appCreate = document.Application.Create;
    Line gLine1 = appCreate.NewLine(new XYZ(0, 2, 0), new XYZ(2, 2, 0), true);
    Line gLine2 = appCreate.NewLine(new XYZ(0, 2, 0), new XYZ(2, 4, 0), true);
    Arc arcToDim = appCreate.NewArc(new XYZ(1, 2, 0), new XYZ(-1, 2, 0), new XYZ(0, 3, 0));
    Arc arcOfDim = appCreate.NewArc(new XYZ(0, 3, 0), new XYZ(1, 2, 0), new XYZ(0.8, 2.8, 0));

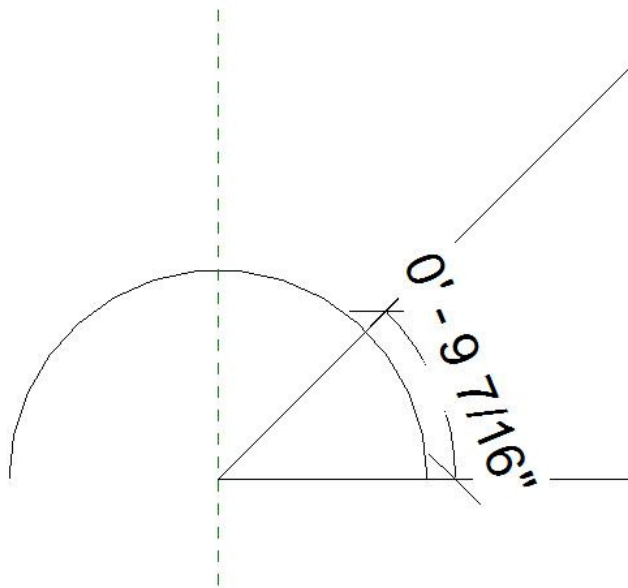
    Autodesk.Revit.Creation.FamilyItemFactory creationFamily = document.FamilyCreate;
    ModelCurve modelCurve1 = creationFamily.NewModelCurve(gLine1, sketchPlane);
    ModelCurve modelCurve2 = creationFamily.NewModelCurve(gLine2, sketchPlane);
    ModelCurve modelCurve3 = creationFamily.NewModelCurve(arcToDim, sketchPlane);
    //get their reference
    Reference ref1 = modelCurve1.GeometryCurve.Reference;
    Reference ref2 = modelCurve2.GeometryCurve.Reference;
    Reference arcRef = modelCurve3.GeometryCurve.Reference;

    Dimension newArcDim = creationFamily.NewArcLengthDimension(document.ActiveView, arcOfDim,
                                                                arcRef, ref1, ref2);

    if (newArcDim == null)
    {
        throw new Exception("Failed to create new arc length dimension.");
    }

    return newArcDim;
}

```

**Figure 63: Resulting arc length dimension**

Some types of dimensions can be labeled with a FamilyParameter. Dimensions that cannot be labeled will throw a System.NotSupportedException 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 11-8: Labeling a dimension

```
public Dimension CreateLinearDimension(Document document)
{
    // first create two lines
    XYZ pt1 = new XYZ(5, 5, 0);
    XYZ pt2 = new XYZ(5, 10, 0);
    Line line = document.Application.Create.NewLine(pt1, pt2, true);
    Plane plane = document.Application.Create.NewPlane(pt1.Cross(pt2), pt2);
    SketchPlane skplane = document.FamilyCreate.NewSketchPlane(plane);
    ModelCurve modelcurve1 = document.FamilyCreate.NewModelCurve(line, skplane);

    pt1 = new XYZ(10, 5, 0);
    pt2 = new XYZ(10, 10, 0);
    line = document.Application.Create.NewLine(pt1, pt2, true);
    plane = document.Application.Create.NewPlane(pt1.Cross(pt2), pt2);
    skplane = document.FamilyCreate.NewSketchPlane(plane);
    ModelCurve modelcurve2 = document.FamilyCreate.NewModelCurve(line, skplane);

    // now create a linear dimension between them
    ReferenceArray ra = new ReferenceArray();
    ra.Append(modelcurve1.GeometryCurve.Reference);
    ra.Append(modelcurve2.GeometryCurve.Reference);

    pt1 = new XYZ(5, 10, 0);
    pt2 = new XYZ(10, 10, 0);
    line = document.Application.Create.NewLine(pt1, pt2, true);

    Dimension dim = document.FamilyCreate.NewLinearDimension(document.ActiveView, line, ra);

    // create a label for the dimension called "width"
    FamilyParameter param = document.FamilyManager.AddParameter("width",
        Autodesk.Revit.Parameters.BuiltInParameterGroup.PG_CONSTRAINTS,
        Autodesk.Revit.Parameters.ParameterType.Length, false);

    dim.Label = param;

    return dim;
}
```

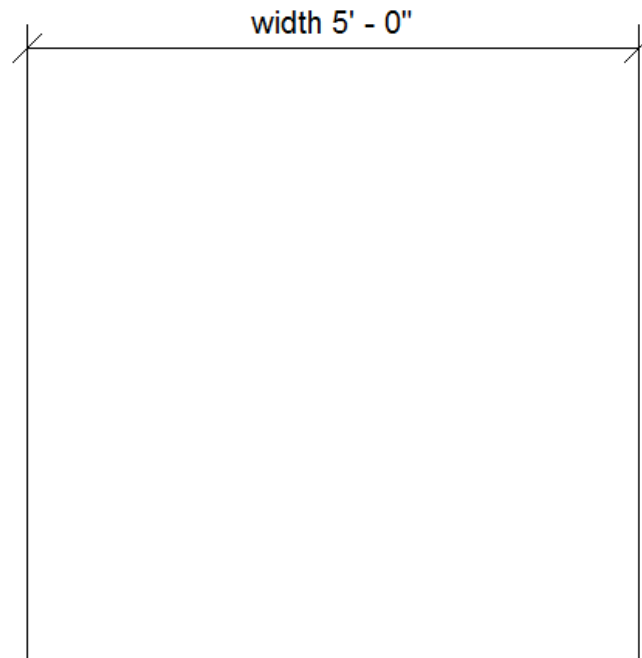


Figure 64: Labeled linear dimension

11.3 Family Element Visibility

The `FamilyElementVisibility` class can be used to control the visibility of family elements in the project document. For example, if you have a door family, you may only want the door swing to be visible in plan views in the project document in which doors are placed, not 3D views. By setting the visibility on the curves of the door swing, you can control their visibility. `FamilyElementVisibility` is applicable to the following family element classes which have the `SetVisibility()` function:

- `GenericForm`, which is the base class for form classes such as Sweep and Extrusion
- `SymbolicCurve`
- `ModelText`
- `CurveByPoints`
- `ModelCurve`
- `ReferencePoint`
- `ImportInstance`

In the example below, the resulting family document will display the text "Hello World" with a line under it. When the family is loaded into a Revit project document and an instance is placed, in plan view, only the line will be visible. In 3D view, both the line and text will be displayed, unless the Detail Level is set to Course, in which case the line will disappear.

Code Region 11-9: Setting family element visibility

```
public void CreateAndSetVisibility(Autodesk.Revit.Document familyDocument, SketchPlane
sketchPlane)
{
    // create a new ModelCurve in the family document
```



```

XYZ p0 = new XYZ(1, 1, 0);
XYZ p1 = new XYZ(5, 1, 0);
Line line1 = familyDocument.Application.Create.NewLineBound(p0, p1);

ModelCurve modelCurve1 = familyDocument.FamilyCreate.NewModelCurve(line1, sketchPlane);

// create a new ModelText along ModelCurve line
ModelText text = familyDocument.FamilyCreate.NewModelText("Hello World", null,
    sketchPlane, p0, Autodesk.Revit.Enums.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);
}

```

11.4 Family Manager

Family documents provide access to the FamilyManager property. The FamilyManager class provides access to family types and parameters. Using this class you can add and remove types, add and remove family and shared parameters, set the value for parameters in different family types, and define formulas to drive parameter values.

11.4.1 Getting Types in a Family

The FamilyManager can be used to iterate through the types in a family, as the following example demonstrates.

Code Region 11-10: Getting the types in a family

```

public void GetFamilyTypesInFamily(Document familyDoc)
{
    if (familyDoc.IsFamilyDocument == true)
    {
        FamilyManager familyManager = familyDoc.FamilyManager;

        // get types in family
        string types = "Family Types: ";
        FamilyTypeSet familyTypes = familyManager.Types;
        FamilyTypeSetIterator familyTypesItor = familyTypes.ForwardIterator();
        familyTypesItor.Reset();
        while (familyTypesItor.MoveNext())
        {
            FamilyType familyType = familyTypesItor.Current as FamilyType;
        }
    }
}

```

```

        types += "\n" + familyType.Name;
    }
    MessageBox.Show(types, "Revit");
}
}

```

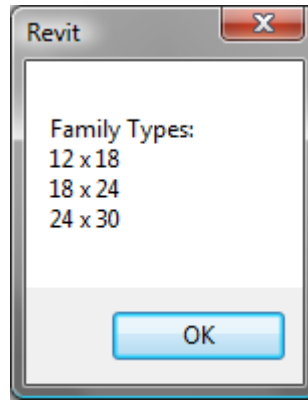


Figure 65: Family types in Concrete-Rectangular-Column family

11.4.2 Editing FamilyTypes

FamilyManager provides the ability to iterate through existing types in a family, and add and modify types and their parameters.

The following example shows how to add a new type, set its parameters and then assign the new type to a FamilyInstance. Type editing is done on the current type by using the Set() function. The current type is available from the CurrentType property. The CurrentType property can be used to set the current type before editing, or use the NewType() function which creates a new type and sets it to the current type for editing.

Note that once the new type is created and modified, Document.LoadFamily() is used to load the family back into the Revit project to make the new type available.

Code Region 11-11: Editing Family Types

```

public void EditFamilyTypes(Document document, FamilyInstance familyInstance)
{
    // example works best when familyInstance is a rectangular concrete element
    if (null != familyInstance.Symbol)
    {
        // Get family associated with this
        Family family = familyInstance.Symbol.Family;

        // Get Family document for family
        Document familyDoc = document.EditFamily(family);
        if (null != familyDoc)
        {
            FamilyManager familyManager = familyDoc.FamilyManager;

            // add a new type and edit its parameters
            FamilyType newFamilyType = familyManager.NewType("2X2");

```

```

// look for 'b' and 'h' parameters and set them to 2 feet
FamilyParameter familyParam = familyManager.get_Parameter("b");
if (null != familyParam)
{
    familyManager.Set(familyParam, 2.0);
}
familyParam = familyManager.get_Parameter("h");
if (null != familyParam)
{
    familyManager.Set(familyParam, 2.0);
}

// now update the Revit project with Family which has a new type
family = familyDoc.LoadFamily(document);
// find the new type and assign it to FamilyInstance
FamilySymbolSetIterator symbolsItor = family.Symbols.ForwardIterator();
symbolsItor.Reset();
while (symbolsItor.MoveNext())
{
    FamilySymbol familySymbol = symbolsItor.Current as FamilySymbol;
    if (familySymbol.Name == "2X2")
    {
        familyInstance.Symbol = familySymbol;
        break;
    }
}
}
}
}

```

12 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.

12.1 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 12-1: Creating a new `CurveByPoints`

```
ReferencePointArray rpa = new ReferencePointArray();

XYZ xyz = document.Application.Create.NewXYZ(0, 0, 0);
ReferencePoint rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

xyz = document.Application.Create.NewXYZ(0, 30, 10);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

xyz = document.Application.Create.NewXYZ(0, 60, 0);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

xyz = document.Application.Create.NewXYZ(0, 100, 30);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

xyz = document.Application.Create.NewXYZ(0, 150, 0);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

CurveByPoints curve = document.FamilyCreate.NewCurveByPoints(rpa);

PointOnEdge poe =
document.Application.Create.NewPointOnEdge(curve.GeometryCurve.Reference, 0.5);
rp = document.FamilyCreate.NewReferencePoint(poe);
```

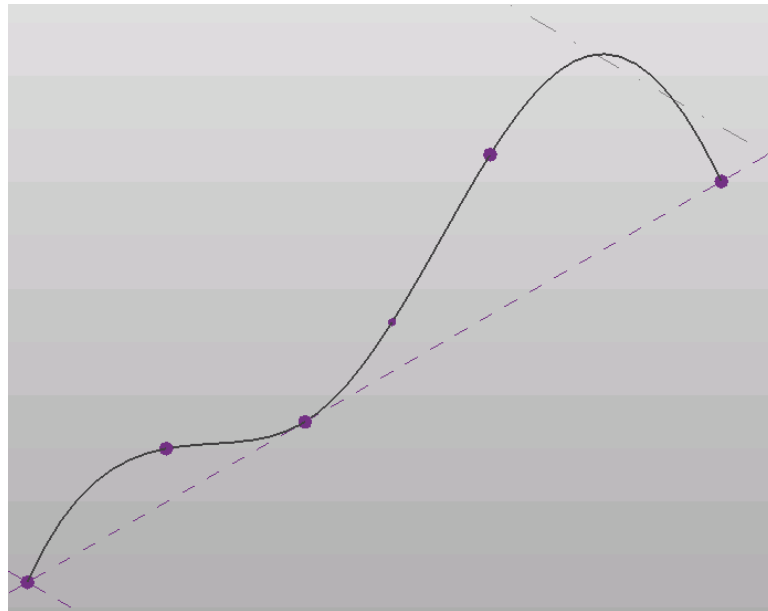


Figure 66: CurveByPoints curve

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 12-2: Using Reference Lines to create Form

```
private FormArray CreateRevolveForm(Document document)
{
    FormArray revolveForms = null;

    // Create one profile
    ReferenceArray ref_ar = new ReferenceArray();

    XYZ ptA = new XYZ(0, 0, 10);
    XYZ ptB = new XYZ(100, 0, 10);
```

```

Line line = document.Application.Create.NewLine(ptA, ptB, true);
ModelCurve modelcurve = MakeLine(document.Application, ptA, ptB);
ref_ar.Append(modelcurve.GeometryCurve.Reference);

ptA = new XYZ(100, 0, 10);
ptB = new XYZ(100, 100, 10);
modelcurve = MakeLine(document.Application, ptA, ptB);
ref_ar.Append(modelcurve.GeometryCurve.Reference);

ptA = new XYZ(100, 100, 10);
ptB = new XYZ(0, 0, 10);
modelcurve = MakeLine(document.Application, ptA, ptB);
ref_ar.Append(modelcurve.GeometryCurve.Reference);

// Create axis for revolve form
ptA = new XYZ(-5, 0, 10);
ptB = new XYZ(-5, 10, 10);
ModelCurve axis = MakeLine(document.Application, 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 MakeLine(Application app, XYZ ptA, XYZ ptB)
{
    Document doc = app.ActiveDocument;
    // Create plane by the points
    Line line = app.Create.NewLine(ptA, ptB, true);
    XYZ norm = ptA.Cross(ptB);
    if (norm.Length == 0) norm = XYZ.BasisZ;
    Plane plane = app.Create.NewPlane(norm, ptB);
    SketchPlane skplane = doc.FamilyCreate.NewSketchPlane(plane);
    // Create line here
    ModelCurve modelcurve = doc.FamilyCreate.NewModelCurve(line, skplane);
    return modelcurve;
}

```

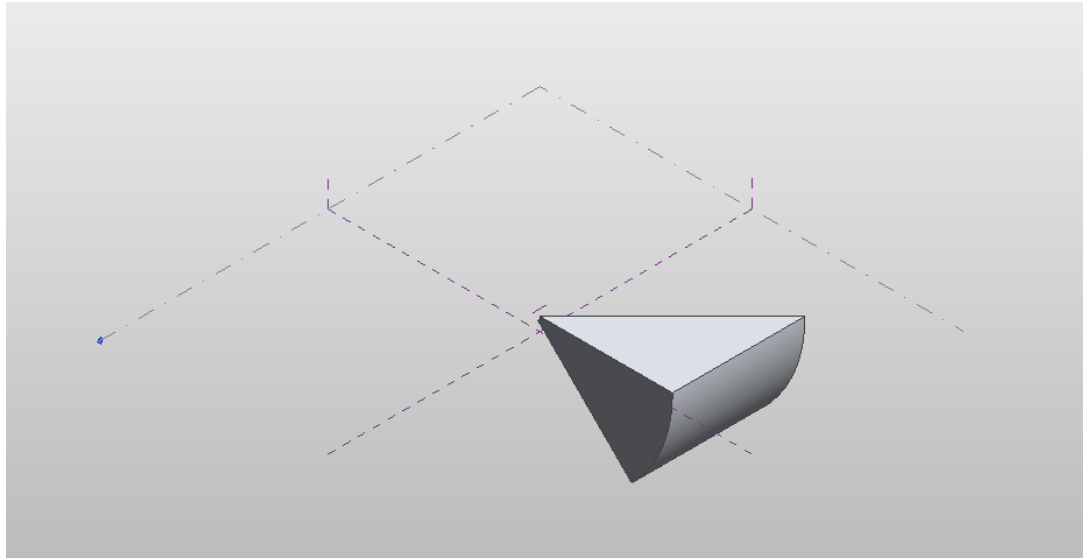


Figure 67: Resulting Revolve Form

12.2 Forms

12.2.1 Creating Forms

Similar to family creation, the conceptual design environment provides the ability to create new forms. The following types of forms can be created: extrusions, revolves, sweeps, swept blends, lofts, and surface forms. Rather than using the Blend, Extrusion, Revolution, Sweep, and SweptBlend classes used in Family creation, Mass families use the Form class for all types of forms.

An extrusion form is created from a closed curve loop that is planar. A revolve form is created from a profile and a line in the same plane as the profile which is the axis around which the shape is revolved to create a 3D form. A sweep form is created from a 2D profile that is swept along a planar path. A swept blend is created from multiple profiles, each one planar, that is swept along a single curve. A loft form is created from 2 or more profiles located on separate planes. A single surface form is created from a profile, similarly to an extrusion, but is given no height.

The following example creates a simple extruded form. Note that since the ModelCurves used to create the form are not converted to reference lines, they will be consumed by the resulting form.

Code Region 12-3: Creating an extrusion form

```
private Form CreateExtrusionForm(Autodesk.Revit.Document document)
{
    Form extrusionForm = null;

    // Create one profile
    ReferenceArray ref_ar = new ReferenceArray();

    XYZ ptA = new XYZ(10, 10, 0);
    XYZ ptB = new XYZ(90, 10, 0);
    ModelCurve modelcurve = MakeLine(document.Application, ptA, ptB);
    ref_ar.Append(modelcurve.GeometryCurve.Reference);

    ptA = new XYZ(90, 10, 0);
```

```

    ptB = new XYZ(10, 90, 0);
    modelcurve = MakeLine(document.Application, ptA, ptB);
    ref_ar.Append(modelcurve.GeometryCurve.Reference);

    ptA = new XYZ(10, 90, 0);
    ptB = new XYZ(10, 10, 0);
    modelcurve = MakeLine(document.Application, ptA, ptB);
    ref_ar.Append(modelcurve.GeometryCurve.Reference);

    // The extrusion form direction
    XYZ direction = new XYZ(0, 0, 50);

    extrusionForm = document.FamilyCreate.NewExtrusionForm(true, ref_ar, direction);

    return extrusionForm;
}

public ModelCurve MakeLine(Application app, XYZ ptA, XYZ ptB)
{
    Document doc = app.ActiveDocument;
    // Create plane by the points
    Line line = app.Create.NewLine(ptA, ptB, true);
    XYZ norm = ptA.Cross(ptB);
    if (norm.Length == 0) norm = XYZ.BasisZ;
    Plane plane = app.Create.NewPlane(norm, ptB);
    SketchPlane skplane = doc.FamilyCreate.NewSketchPlane(plane);
    // Create line here
    ModelCurve modelcurve = doc.FamilyCreate.NewModelCurve(line, skplane);
    return modelcurve;
}

```

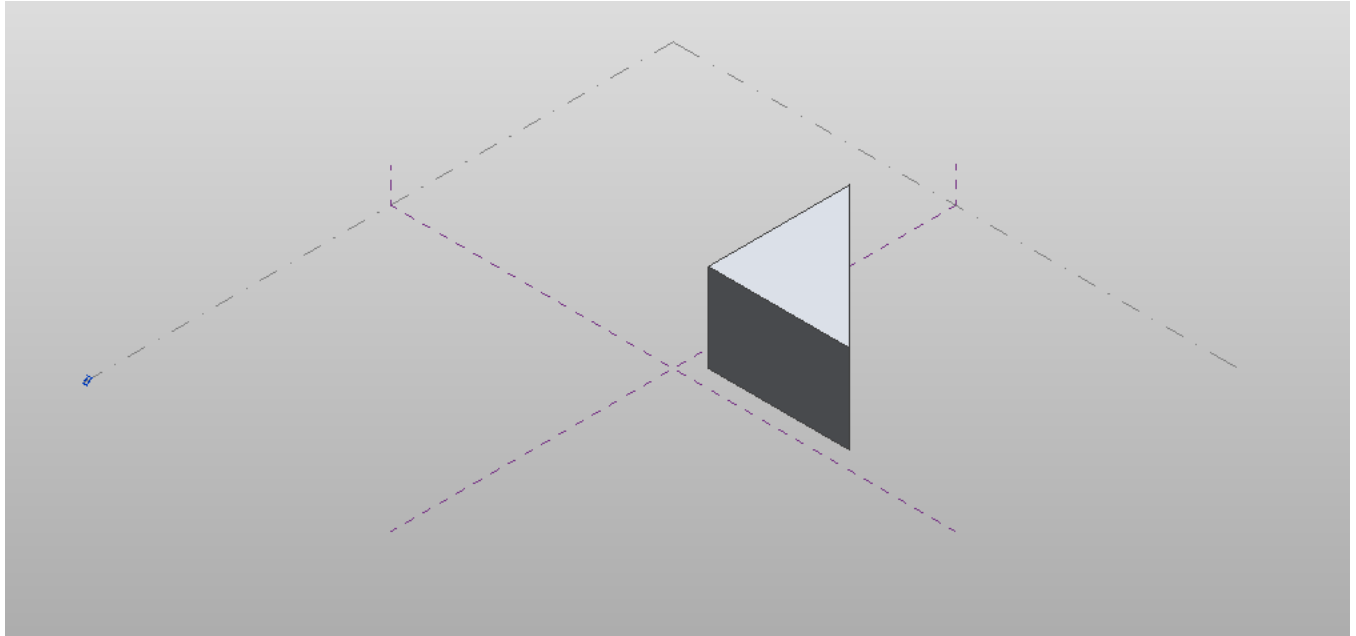



Figure 68: Resulting extrusion form

The following example shows how to create loft form using a series of CurveByPoints objects.

Code Region 12-4: Creating a loft form

```
private Form CreateLoftForm(Autodesk.Revit.Document document)
{
    Form loftForm = null;

    ReferencePointArray rpa = new ReferencePointArray();
    ReferenceArrayArray ref_ar_ar = new ReferenceArrayArray();
    ReferenceArray ref_ar = new ReferenceArray();
    ReferencePoint rp = null;
    XYZ xyz = null;

    // make first profile curve for loft
    xyz = document.Application.Create.NewXYZ(0, 0, 0);
    rp = document.FamilyCreate.NewReferencePoint(xyz);
    rpa.Append(rp);

    xyz = document.Application.Create.NewXYZ(0, 50, 10);
    rp = document.FamilyCreate.NewReferencePoint(xyz);
    rpa.Append(rp);

    xyz = document.Application.Create.NewXYZ(0, 100, 0);
    rp = document.FamilyCreate.NewReferencePoint(xyz);
    rpa.Append(rp);

    CurveByPoints cbp = document.FamilyCreate.NewCurveByPoints(rpa);
    ref_ar.Append(cbp.GeometryCurve.Reference);
    ref_ar_ar.Append(ref_ar);
    rpa.Clear();
}
```

```

ref_ar = new ReferenceArray();

// make second profile curve for loft
xyz = document.Application.Create.NewXYZ(50, 0, 0);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

xyz = document.Application.Create.NewXYZ(50, 50, 30);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

xyz = document.Application.Create.NewXYZ(50, 100, 0);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

cbp = document.FamilyCreate.NewCurveByPoints(rpa);
ref_ar.Append(cbp.GeometryCurve.Reference);
ref_ar_ar.Append(ref_ar);
rpa.Clear();
ref_ar = new ReferenceArray();

// make third profile curve for loft
xyz = document.Application.Create.NewXYZ(75, 0, 0);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

xyz = document.Application.Create.NewXYZ(75, 50, 5);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

xyz = document.Application.Create.NewXYZ(75, 100, 0);
rp = document.FamilyCreate.NewReferencePoint(xyz);
rpa.Append(rp);

cbp = document.FamilyCreate.NewCurveByPoints(rpa);
ref_ar.Append(cbp.GeometryCurve.Reference);
ref_ar_ar.Append(ref_ar);

loftForm = document.FamilyCreate.NewLoftForm(true, ref_ar_ar);

return loftForm;
}

```

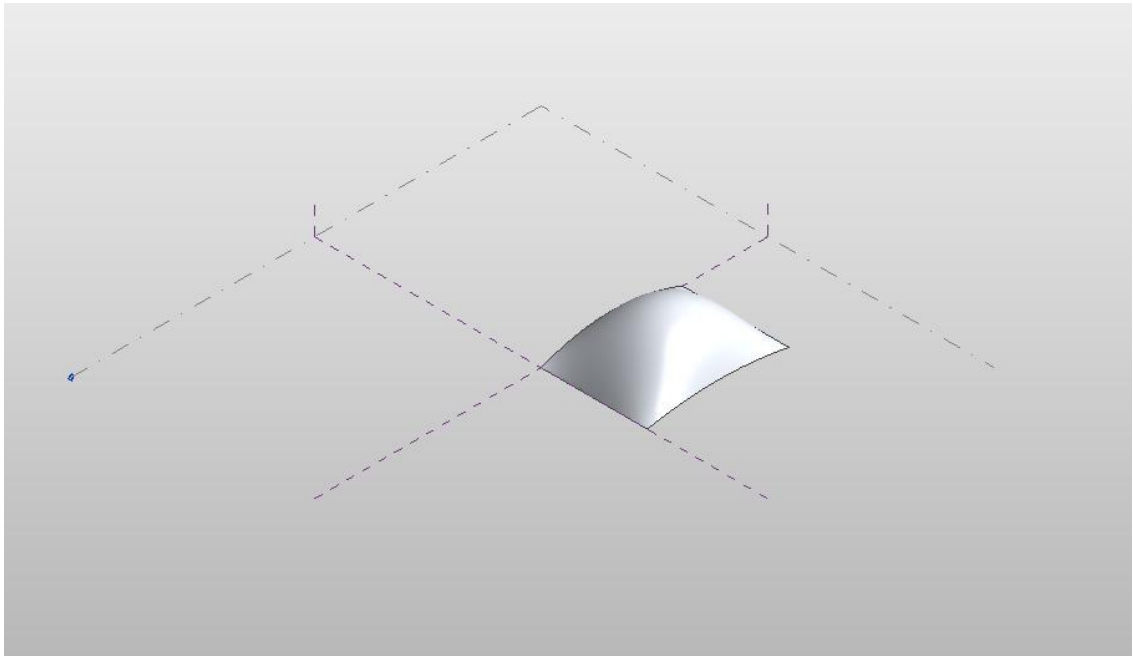


Figure 69: Resulting loft form

12.2.2 Form modification

Once created, forms can be modified by changing a sub element (i.e. a face, edge, curve or vertex) of the form, or an entire profile. The methods to modify a form include:

- AddEdge
- AddProfile
- DeleteProfile
- DeleteSubElement
- MoveProfile
- MoveSubElement
- RotateProfile
- RotateSubElement
- ScaleSubElement

Additionally, you can modify a form by adding an edge or a profile, which can then be modified using the methods listed above.

The following example moves the first profile curve of the given form by a specified offset. The corresponding figure shows the result of applying this code to the loft form from the previous example.

Code Region 12-5: 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);
}
}

```

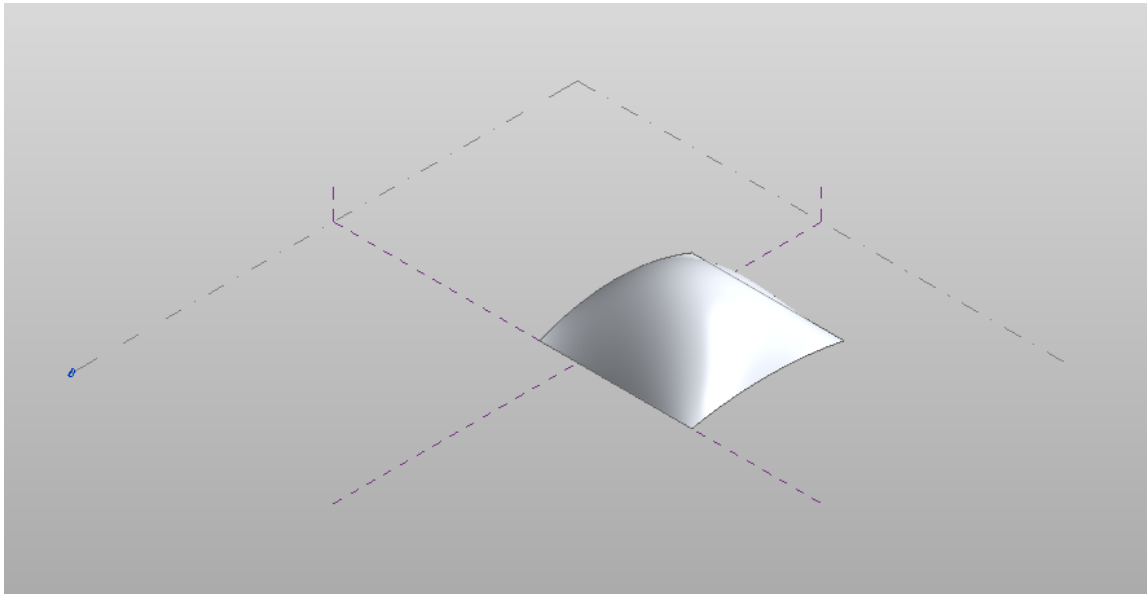


Figure 70: Modified loft form

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 12-6: Moving a sub element

```

public void MoveSubElement(Form form)
{
    if (form.ProfileCount > 0)
    {
        int profileIndex = 0; // get first profile
        ReferenceArray ra = form.get_CurveLoopReferencesOnProfile(profileIndex, 0);
        foreach (Reference r in ra)
        {
            ReferenceArray ra2 = form.GetControlPoints(r);
            foreach (Reference r2 in ra2)
            {
                Point vertex = r2.GeometryObject as Point;

                XYZ offset = new XYZ(0, 15, 0);
                form.MoveSubElement(r2, offset);
                break; // just move the first point
            }
        }
    }
}

```

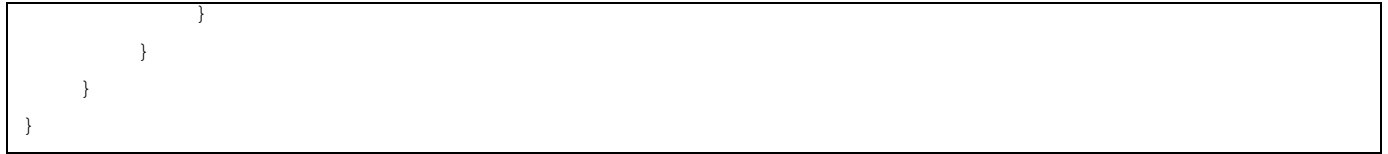


Figure 71: Modified extrusion form

12.3 Rationalizing a Surface

12.3.1 Dividing a surface

Faces of forms can be divided with UV grids. You can access the data for a divided surface using the `Form.GetDividedSurfaceData()` method (as is shown in a subsequent example) as well as create new divided surfaces on forms as shown below.

Code Region 12-7: Dividing a surface

```
public void DivideSurface(Document document, Form form)
{
    Application application = document.Application;
    Options opt = application.Create.NewGeometryOptions();
    opt.ComputeReferences = true;

    Autodesk.Revit.Geometry.Element geomElem = form.get_Geometry(opt);
    foreach (GeometryObject geomObj in geomElem.Objects)
    {
        Solid solid = geomObj as Solid;
        foreach (Face face in solid.Faces)
        {
            if (face.Reference != null)
            {
                DividedSurface ds = document.FamilyCreate.NewDividedSurface(face.Reference);
            }
        }
    }
}
```

```

// create a divided surface with fixed number of U and V grid lines
SpacingRule srU = ds.USpacingRule;
srU.SetLayoutFixedNumber(16, SpacingRuleJustification.Center, 0, 0);

SpacingRule srV = ds.VSpacingRule;
srV.SetLayoutFixedNumber(24, SpacingRuleJustification.Center, 0, 0);

break; // just divide one face of form
    }
}
}
}

```

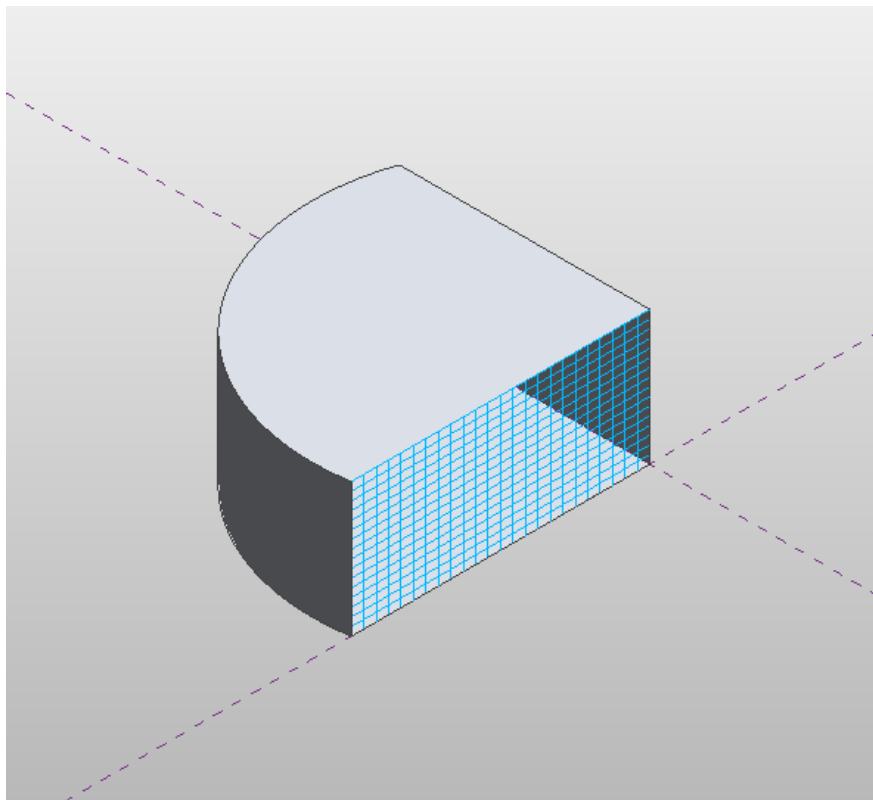


Figure 72: Face of form divided by UV grids

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.

12.3.2 Patterning a surface

A divided surface can be patterned. Any of the built-in tile patterns can be applied to a divided surface. A tile pattern is a `Symbol` that is assigned to the `ObjectType` property of a `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.

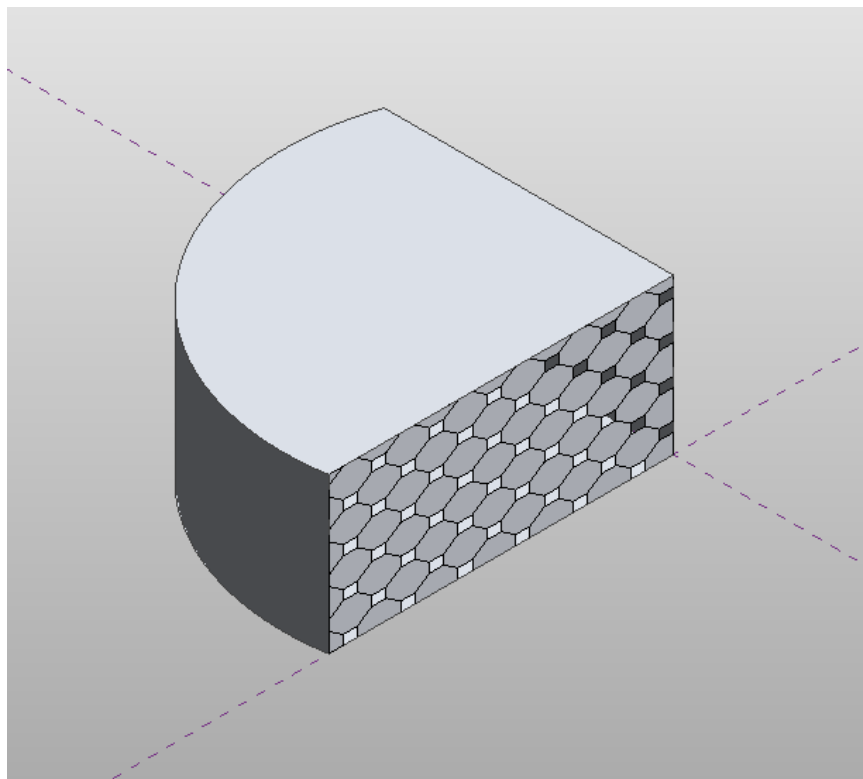
Code Region 12-8: Patterning a surface

```

public void TileSurface(Document document, Form form)
{
    // cover surface with OctagonRotate tile pattern
    TilePatterns tilePatterns = document.Settings.TilePatterns;
    DividedSurfaceData dsData = form.GetDividedSurfaceData();
    if (dsData != null)
    {
        foreach (Reference r in dsData.GetReferencesWithDividedSurfaces())
        {
            DividedSurface ds = dsData.GetDividedSurfaceForReference(r);

            ds.ObjectType = tilePatterns.GetTilePattern(TilePatternsBuiltIn.OctagonRotate);
        }
    }
}

```

**Figure 73: 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.ObjectType` property.

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 12-9: Editing a curtain panel family

```
Family family = document.OwnerFamily;
if (family.IsCurtainPanelFamily == true &&
    family.CurtainPanelTilePattern == Autodesk.Revit.Enums.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.Application app = document.Application;
    Filter filter = app.Create.Filter.NewTypeFilter(typeof(ReferencePoint));
    List<Autodesk.Revit.Element> list = new List<Autodesk.Revit.Element>();
    Int64 num = app.ActiveDocument.get_Elements(filter, list);
    int ctr = 0;
    ReferencePoint rp0 = null, rp1 = null, rp2 = null, rp3 = null;
    foreach (Autodesk.Revit.Element e in list)
    {
        ReferencePoint rp = e as ReferencePoint;
        switch (ctr)
        {
            case 0:
                rp0 = rp;
                break;
            case 1:
                rp1 = rp;
                break;
            case 2:
                rp2 = rp;
                break;
            case 3:
                rp3 = rp;
                break;
        }
        ctr++;
    }

    ReferencePointArray rpAr = new ReferencePointArray();
    rpAr.Append(rp0);
    rpAr.Append(rp2);
    CurveByPoints curve1 = document.FamilyCreate.NewCurveByPoints(rpAr);
    PointOnEdge poeA = app.Create.NewPointOnEdge(curve1.GeometryCurve.Reference, 0.25);
```



```

ReferencePoint rpA = document.FamilyCreate.NewReferencePoint(poeA);
PointOnEdge poeB = app.Create.NewPointOnEdge(curve1.GeometryCurve.Reference, 0.75);
ReferencePoint rpB = document.FamilyCreate.NewReferencePoint(poeB);

rpAr.Clear();
rpAr.Append(rp1);
rpAr.Append(rp3);
CurveByPoints curve2 = document.FamilyCreate.NewCurveByPoints(rpAr);
PointOnEdge poeC = app.Create.NewPointOnEdge(curve2.GeometryCurve.Reference, 0.25);
ReferencePoint rpC = document.FamilyCreate.NewReferencePoint(poeC);
PointOnEdge poeD = app.Create.NewPointOnEdge(curve2.GeometryCurve.Reference, 0.75);
ReferencePoint rpD = document.FamilyCreate.NewReferencePoint(poeD);
}
else
{
    throw new Exception("Please open a curtain family document before calling this
command.");
}

```

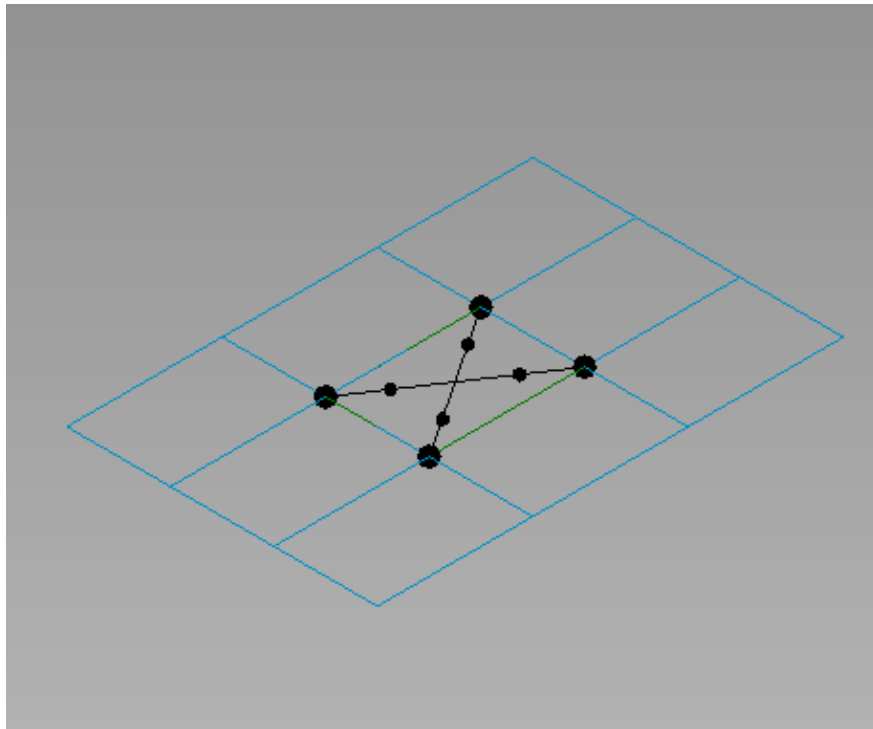


Figure 74: Curtain panel family

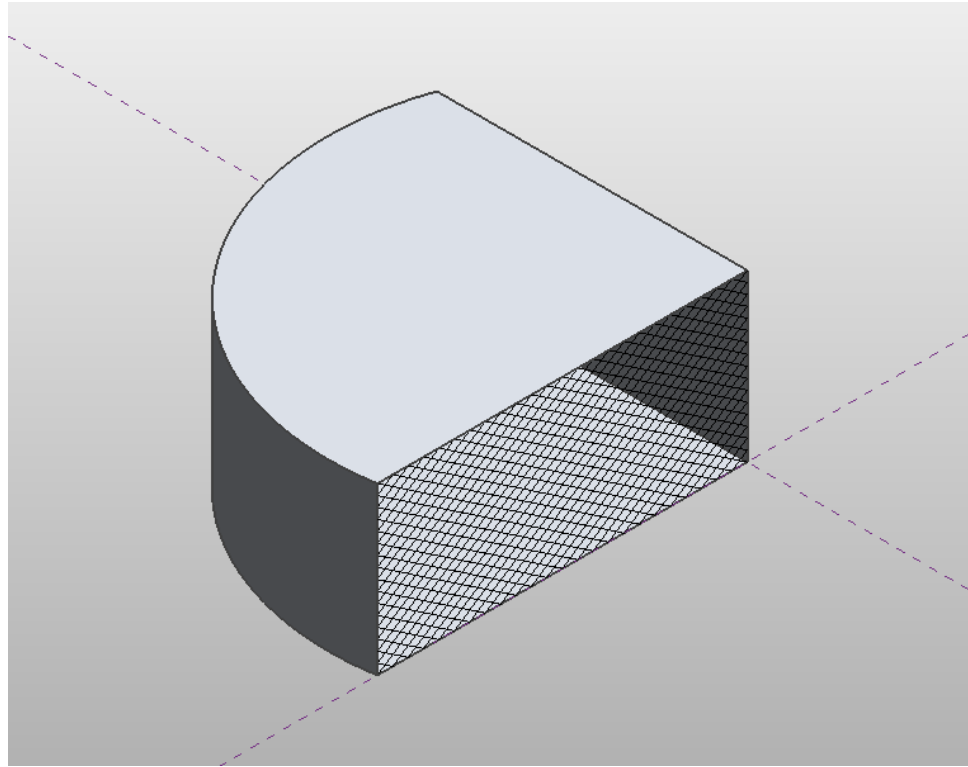


Figure 75: Curtain panel assigned to divided surface

13 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 gbXMLParamElements.

For more information about Revit Element classifications, refer to the [Elements Essentials](#) chapter

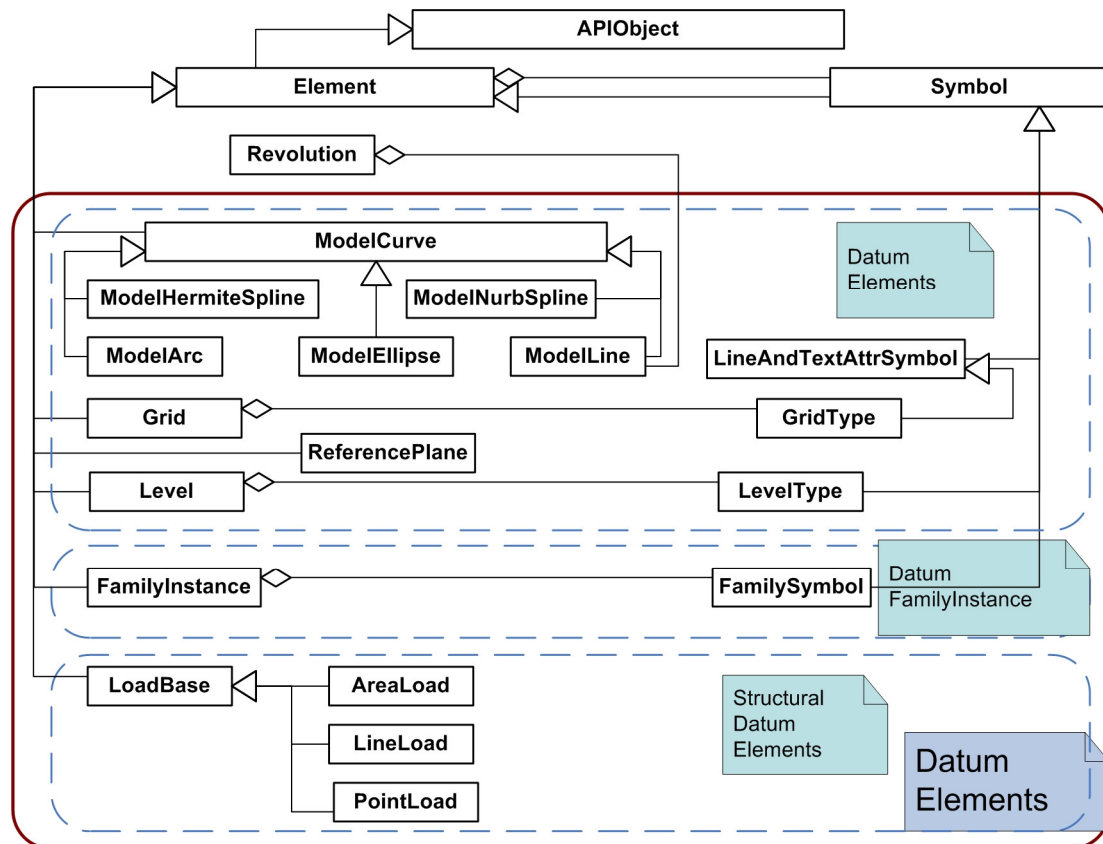


Figure 76: Datum Elements diagram

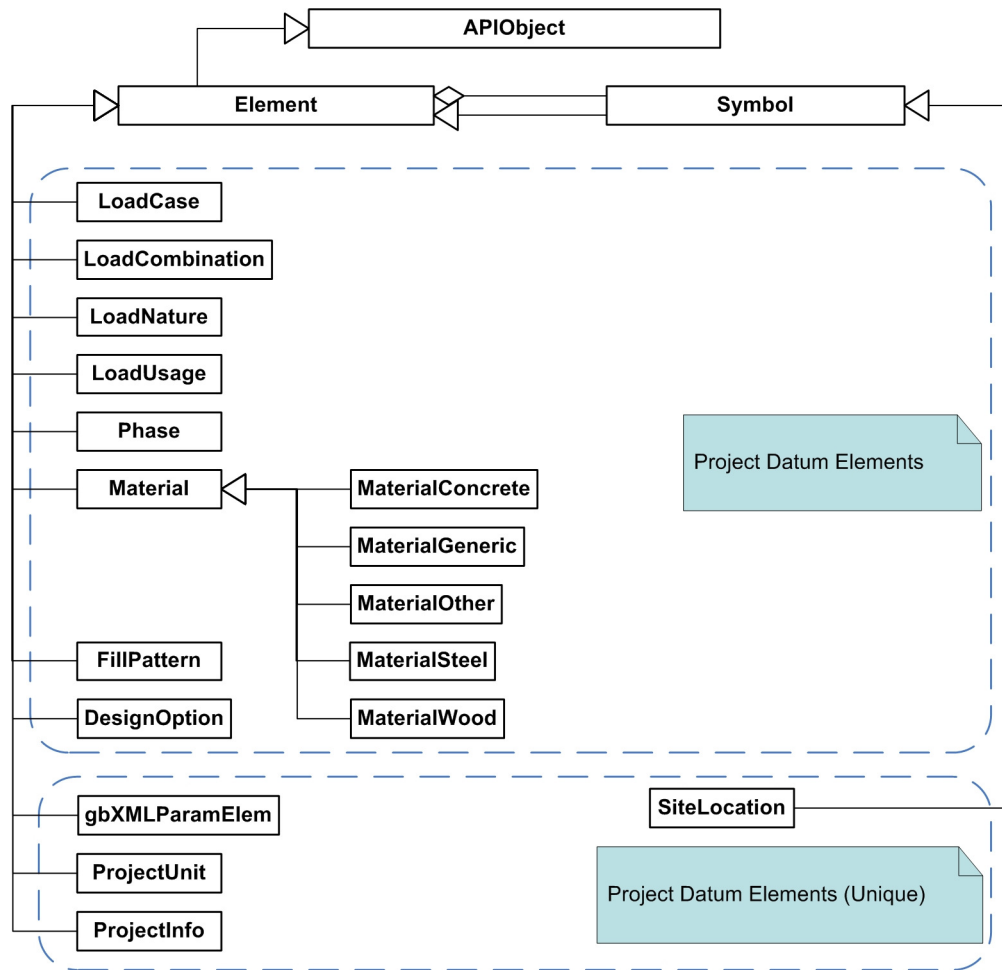


Figure 77: Information Elements diagram

Note: Many elements in the previous diagram are not introduced in this chapter. If you need more information, refer to the related chapter:

- For LoadBase, LoadCase, LoadCombination, LoadNature and LoadUsage, refer to the [Revit Structure](#) chapter
- For ModelCurve, refer to the [Sketching](#) chapter
- For Material and FillPattern, refer to the [Material](#) chapter
- For gbXMLParamElem, refer to the [Revit Architecture](#) chapter

13.1 Levels

A level is a finite horizontal plane that acts as a reference for level-hosted elements, such as walls, roofs, floors, and ceilings. In the Revit Platform API, the Level class is derived from the Element class. The inherited Name property is used to retrieve the user-visible level name beside the level bubble in the Revit UI. To retrieve all levels in a project, use the ElementIterator iterator to search for Level objects.

13.1.1 Elevation

The Level class has the following properties:

- The Elevation property (LEVEL_ELEV) is used to retrieve or change the elevation above or below ground level.

- The ProjectElevation property is used to retrieve the elevation relative to the project origin regardless of the Elevation Base parameter value.
- Elevation Base is a Level type parameter.
 - Its BuiltInParameter is LEVEL_RELATIVE_BASE_TYPE.
 - Its StorageType is Integer
 - 0 corresponds to Project and 1 corresponds to Shared.

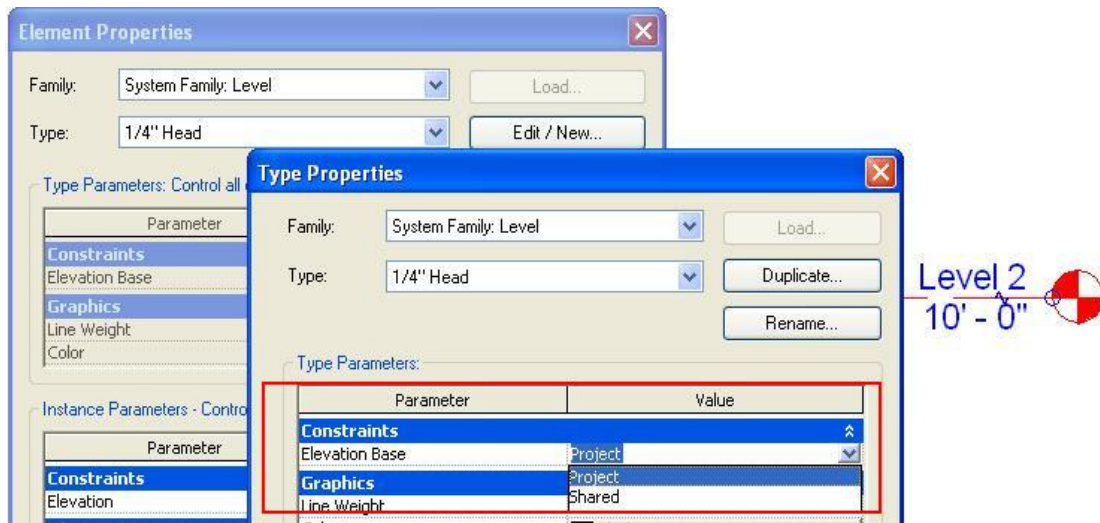


Figure 78: Level Type Elevation Base property

The following code sample illustrates how to retrieve all levels in a project using ElementIterator.

Code Region 13-1: Retrieving all Levels

```
private void Getinfo_Level(Document document)
{
    StringBuilder levelInformation = new StringBuilder();
    int levelNumber = 0;

    TypeFilter levelFilter = document.Application.Create.Filter.NewTypeFilter(typeof(Level));
    ElementIterator itrLevels = document.get_Elements(levelFilter);
    itrLevels.Reset();
    while (itrLevels.MoveNext())
    {
        Level level = itrLevels.Current 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
MessageBox.Show(levelInformation.ToString(), "Revit");
}

```

13.1.2 Creating a Level

Using the Level command, you can define a vertical height or story within a building and you can create a level for each existing story or other building references. Levels must be added in a section or elevation view. Additionally, you can create a new level using the Revit Platform API.

The following code sample illustrates how to create a new level.

Code Region 13-2: Creating a new Level

```

Level CreateLevel(Autodesk.Revit.Document document)
{
    // The elevation to apply to the new level
    double elevation = 20.0;

    // Begin to create a level
    Level level = document.Create.NewLevel(elevation);
    if (null == level)
    {
        throw new Exception("Create a new level failed.");
    }

    // Change the level name
    level.Name = "New level";

    return level;
}

```

Note: After creating a new level, Revit does not create the associated plan view for this level. If necessary, you can create it yourself. For more information about how to create a plan view, refer to the [Views](#) chapter.

13.2 Grids

Grids are represented by the Grid class which is derived from the Element class. It contains all grid properties and methods. The inherited Name property is used to retrieve the content of the grid line's bubble.

13.2.1 Curve

The Grid class Curve property gets the object that represents the grid line geometry.

- If the IsCurved property returns true, the Curve property will be an Arc class object.
- If the IsCurved property returns false, the Curve property will be a Line class object.

For more information, refer to the [Geometry](#) chapter.

The following code is a simple example using the Grid class. The result appears in a message box after invoking the command.

Code Region 13-3: 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.Geometry.Curve curve = grid.Curve;
    if (grid.IsCurved)
    {
        // if the curve is an arc, give center and radius information
        Autodesk.Revit.Geometry.Arc arc = curve as Autodesk.Revit.Geometry.Arc;
        message += "\nArc's radius: " + arc.Radius;
        message += "\nArc's center: (" + XYZToString(arc.Center);
    }
    else
    {
        // if the curve is a line, give length information
        Autodesk.Revit.Geometry.Line line = curve as Autodesk.Revit.Geometry.Line;
        message += "\nLine's Length: " + line.Length;
    }

    // Get curve start point
    message += "\nStart point: " + XYZToString(curve.get_EndPoint(0));
    // Get curve end point
    message += "; End point: " + XYZToString(curve.get_EndPoint(1));

    MessageBox.Show(message, "Revit", MessageBoxButtons.OK);
}

// output the point's three coordinates
string XYZToString(Autodesk.Revit.Geometry.XYZ point)
{
    return "(" + point.X + ", " + point.Y + ", " + point.Z + ")";
}
```

13.2.2 Creating a Grid

Two overloaded Document methods are available to create a new grid in the Revit Platform API. Using the following method with different parameters, you can create a curved or straight grid:

Code Region 13-4: NewGrid()

```
public Grid NewGrid( Arc arc );
public Grid NewGrid( Line line );
```

Note: The arc or the line used to create a grid must be in a horizontal plane.

The following code sample illustrates how to create a new grid with a line or an arc.

Code Region 13-5: Creating a grid with a line or an arc

```
void CreateGrid(Autodesk.Revit.Document document)
{
    // Create the geometry line which the grid locates
    XYZ start = new XYZ(0, 0, 0);
    XYZ end = new XYZ(30, 30, 0);
    Line geomLine = Line.get_Bound(start, end);

    // Create a grid using the geometry line
    Grid lineGrid = document.Create.NewGrid(geomLine);

    if (null == lineGrid)
    {
        throw new Exception("Create a new straight grid failed.");
    }

    // Modify the name of the created grid
    lineGrid.Name = "New Name1";

    // Create the geometry arc which the grid locates
    XYZ end0 = new XYZ(0, 0, 0);
    XYZ end1 = new XYZ(10, 40, 0);
    XYZ pointOnCurve = new XYZ(5, 7, 0);
    Arc geomArc = document.Application.Create.NewArc(end0, end1, pointOnCurve);

    // Create a grid using the geometry arc
    Grid arcGrid = document.Create.NewGrid(geomArc);

    if (null == arcGrid)
    {
        throw new Exception("Create a new curved grid failed.");
    }

    // Modify the name of the created grid
    arcGrid.Name = "New Name2";
}
```


Note: In Revit, the grids are named automatically in a numerical or alphabetical sequence when they are created.

You can also create several grids at once using the `Document.NewGrids()` method, which takes a `CurveArray` parameter.

13.3 Phase

Some architectural projects, such as renovations, proceed in phases. Phases have the following characteristics:

- Phases represent distinct time periods in a project lifecycle.
- The lifetime of an element within a building is controlled by phases.
- Each element has a construction phase but only the elements with a finite lifetime have a destruction phase.

All phases in a project can be retrieved from the `Document` object. A `Phase` object contains three pieces of useful information: Name, ID and `UniqueId`. The remaining properties always return null or an empty collection.

Each new modeling component added to a project has a `Phase Created` and a `Phase Demolished` property. The `Phase Created` property has the following characteristics:

- It identifies the phase in which the component was added.
- The default value is the same as the current view `Phase` value.
- Change the `Phase Created` parameter by selecting a new value from the drop-down list.

The `Phase Demolished` property has the following characteristics:

- It identifies in which phase the component is demolished.
- The default value is none.
- Demolishing a component with the demolition tool updates the property to the current `Phase` value in the view where you demolished the element.
- You can demolish a component by setting the `Phase Demolished` property to a different value.
- If you delete a phase using the Revit Platform API, all modeling components in the current phase still exist. The `Phase Created` parameter value for these components is changed to the next item in the drop-down list in the Properties dialog box.

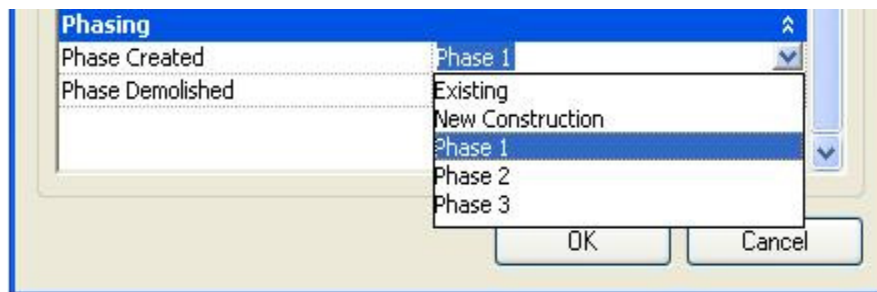


Figure 79: Phase-created component parameter value

The following code sample displays all supported phases in the current document. The phase names are displayed in a message box.

Code Region 13-6: Displaying all supported phases

```

void Getinfo_Phase(Document doc)
{
    // Get the phase array which contains all the phases.
    PhaseArray phases = doc.Phases;
    // Format the prompt 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.
    MessageBox.Show(prompt, "Revit", MessageBoxButtons.OK);
}

```

13.4 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 13-7: Using design options

```

void Getinfo_DesignOption(Document document)
{
    // Get the selected Elements in the Active Document
    ElementSet selection = document.Selection.Elements;

    foreach (Autodesk.Revit.Element element in selection)
    {
        //Use the DesignOption property of Element
        if (element.DesignOption != null)
        {
            MessageBox.Show(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.

14 Annotation Elements

This chapter introduces Revit Annotation Elements, including the following:

- Dimension
- DetailCurve
- IndependentTag
- TextNote
- AnnotationSymbol

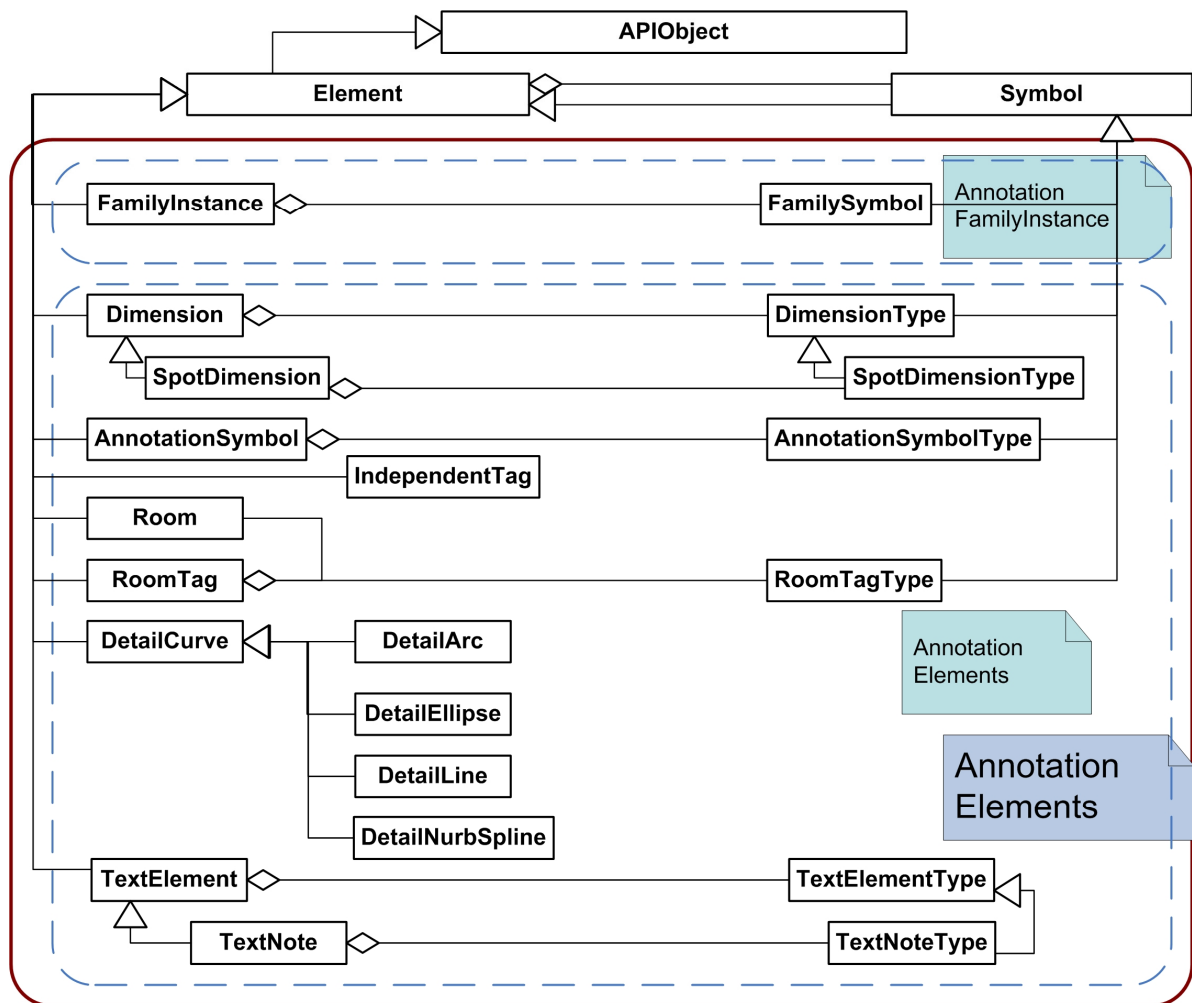


Figure 80: Annotation Elements diagram

- Dimensions are view-specific elements that display sizes and distances in a project.
- Detail curves are created for detailed drawings. They are visible only in the view in which they are drawn. Often they are drawn over the model view.
- Tags are an annotation used to identify elements in a drawing. Properties associated with a tag can appear in schedules.
- AnnotationSymbol has multiple leader options when loaded into a project.

For more information about Revit Element classification, refer to the [Elements Essentials](#) chapter.

14.1 Dimensions and Constraints

- The Dimension class represents permanent dimensions and dimension related constraint elements. Temporary dimensions created while editing an element in the UI are not accessible. Spot elevation and spot coordinate are represented by the SpotDimension class.

The following code sample illustrates, near the end, how to distinguish permanent dimensions from constraint elements.

Code Region 14-1: 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.Geometry.Curve curve = dimension.Curve;
    if (curve != null && curve.IsBound)
    {
        // Get curve start point
        message += "\nCurve start point:(" + curve.get_EndPoint(0).X + ", "
            + curve.get_EndPoint(0).Y + ", " + curve.get_EndPoint(0).Z + ")";

        // Get curve end point
        message += "; Curve end point:(" + curve.get_EndPoint(1).X + ", "
            + curve.get_EndPoint(1).Y + ", " + curve.get_EndPoint(1).Z + ")";
    }

    // Get Dimension type name
    message += "\nDimension type name is : " + dimension.DimensionType.Name;

    // Get Dimension view name
    message += "\nDimension view name is : " + dimension.View.ViewName;

    // Get Dimension reference count
    message += "\nDimension references count is " + dimension.References.Size;

    if ((int)BuiltInCategory.OST_Dimensions == dimension.Category.Id.Value)
    {
        message += "\nDimension is a permanent dimension.";
    }
    else if ((int)BuiltInCategory.OST_Constraints == dimension.Category.Id.Value)
    {
        message += "\nDimension is a constraint element.";
    }
    MessageBox.Show(message, "Revit", MessageBoxButtons.OK);
}
```

14.1.1 Dimensions

There are four kinds of permanent dimensions:

- Linear dimension
- Radial dimension
- Angular dimension
- Arc length dimension

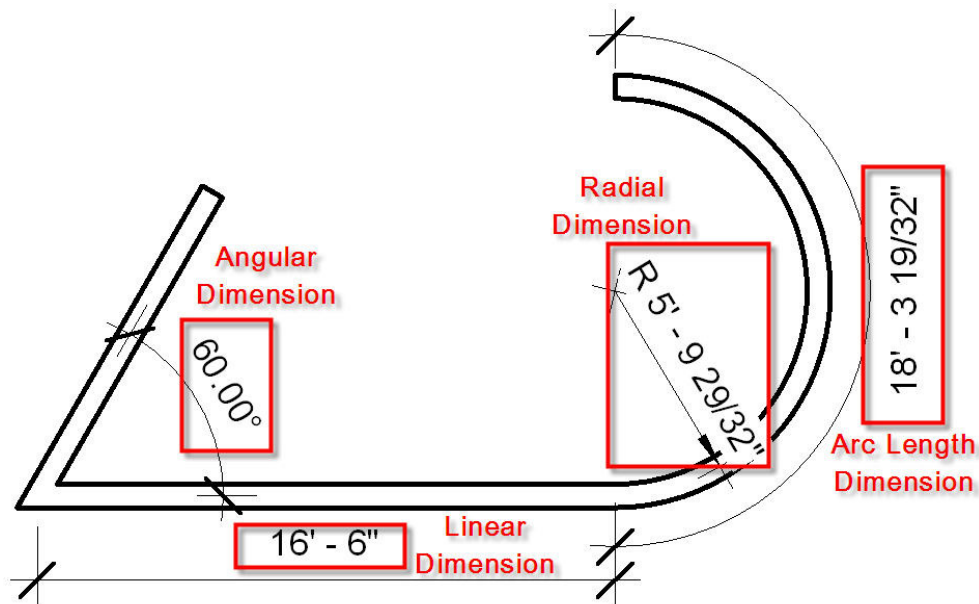


Figure 81: Permanent dimensions

The BuiltInCategory for all permanent dimensions is OST_Dimensions. There is not an easy way to distinguish the four dimensions using the API.

Except for radial dimension, every dimension has one dimension line. Dimension lines are available from the Dimension.Curve property which is always unbound. In other words, the dimension line does not have a start-point or end-point. Based on the previous picture:

- A Line object is returned for a linear dimension.
- An arc object is returned for a radial dimension or angular dimension.
- A radial dimension returns null.

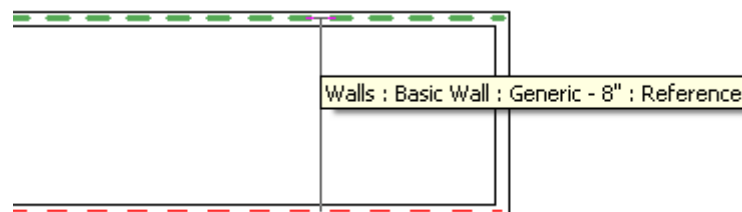


Figure 82: Dimension references

A dimension is created by selecting geometric references as the previous picture shows. Geometric references are represented as a Reference class in the API. The following dimension references are available from the References property. For more information about Reference, please see the [Geometry.Reference](#) section in the [Geometry](#) chapter.

- Radial dimension - One Reference object for the curve is returned
- Angular and arc length dimensions - Two Reference objects are returned.
- Linear dimensions - Two or more Reference objects are returned. In the following picture, the linear dimension has five Reference objects.

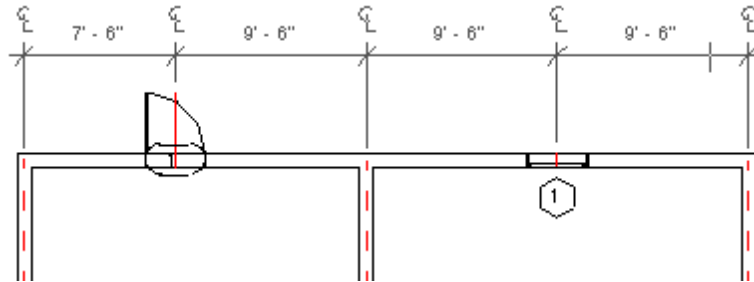


Figure 83: Linear dimension references

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.

14.1.2 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.

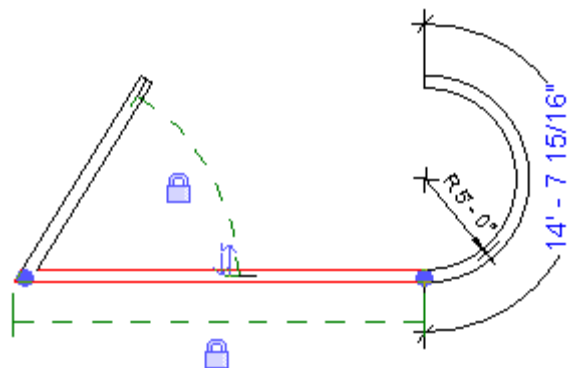


Figure 84: Linear and Radial dimension constraints

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.

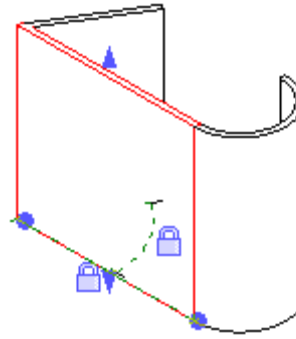


Figure 85: Linear and Radial dimension constraints in 3D view

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.

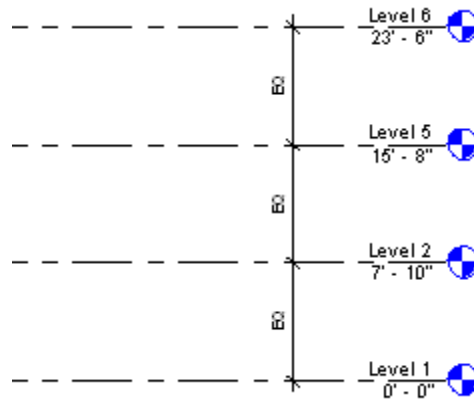


Figure 86: Equality constraints

Note: Not all constraint elements are represented by the Dimension class but all belong to a Constraints (OST_Constraints) category such as alignment constraint.

14.1.3 Spot Dimensions

Spot coordinates and spot elevations are represented by the SpotDimension class and are distinguished by category. Like the permanent dimension, spot dimensions are view-specific. The type and category for each spot dimension are listed in the following table:

Type	Category
Spot Coordinates	OST_SpotCoordinates
Spot Elevations	OST_SpotElevations

Table 31: Spot dimension Type and Category

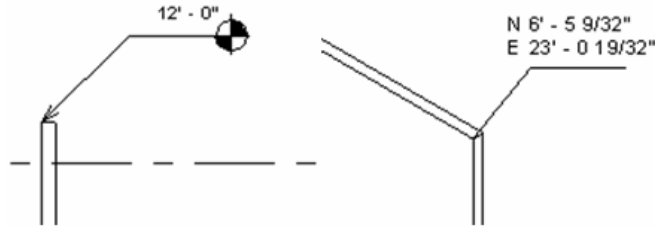


Figure 87: SpotCoordinates and SpotElevations

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.

14.1.4 Comparison

The following table compares different kinds of dimensions and constraints in the API:

Dimension or Constraint		API Class	BuiltInCategory	Curve	Reference	View	Location
Permanent Dimension	linear dimension	Dimension	OST_Dimensions	A Line	>=2	Specific view	null
				Null	1		
	radial dimension			An Arc	2		
				An Arc	2		
	angular dimension		OST_Constraints	A Line	2		
				An Arc	2		
arc length dimension	A Line			>=3			
Dimension Constraint	linear dimension constraint						
	angular dimension						
Equality Constraint							
Spot Dimension	Spot Coordinates	SpotDimension	OST_SpotCoordinates	Null	1	Specific view	LocationPoint
	Spot Elevations		OST_SpotElevations				

Table 32: Dimension Category Comparison

14.1.5 Create and Delete

The NewDimension() method is available in the Creation.Document class. This method can create a linear dimension only.

Code Region 14-2: 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 14-3: Duplicating a dimension with NewDimension()

```
public void DuplicateDimension(Document document, Dimension dimension)
{
    Line line = dimension.Curve as Line;
    if (null != line)
    {
        Autodesk.Revit.Elements.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.

14.2 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.

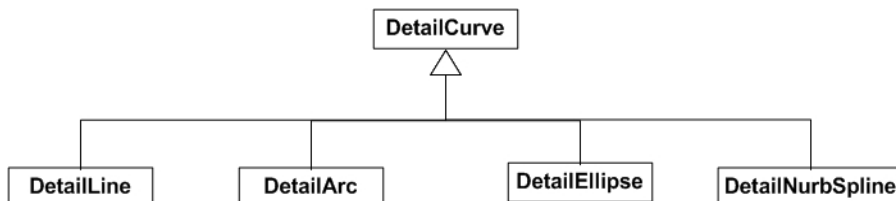


Figure 88: DetailCurve diagram

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 14-4: NewDetailCurve() and NewModelCurve()

```
public DetailCurve NewDetailCurve (View, Curve, SketchPlane)

public ModelCurve NewModelCurve (Curve, SketchPlane)
```

Generally only 2D views such as level view and elevation view are acceptable, otherwise an exception is thrown.

Except for view-related features, DetailCurve is very similar to ModelCurve. For more information about ModelCurve properties and usage, see the [ModelCurve](#) section in the [Sketching](#) chapter.

14.3 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 the [Room](#) section in the [Revit Architecture](#) chapter.

Note: The IndependentTag class represents the tag element in Revit and other specific tags such as keynote, beam system tag, electronic circuit symbol (Revit MEP), and so on. In Revit internal code, the specific tags have corresponding classes derived from IndependentTag. As a result, specific features are not exposed by the API and cannot be created using the NewTag() method. They can be distinguished by the following categories:

Tag Name	BuiltInCategory
Keynote Tag	OST_KeynoteTags
Beam System Tag	OST_BeamSystemTags
Electronic Circuit Tag	OST_ElectricalCircuitTags
Span Direction Tag	OST_SpanDirectionSymbol
Path Reinforcement Span Tag	OST_PathReinSpanSymbol
Rebar System Span Tag	OST_IOSRebarSystemSpanSymbolCtrl

Table 33: Tag Name and Category

In this section, the main focus is on the tag type represented in the following picture.

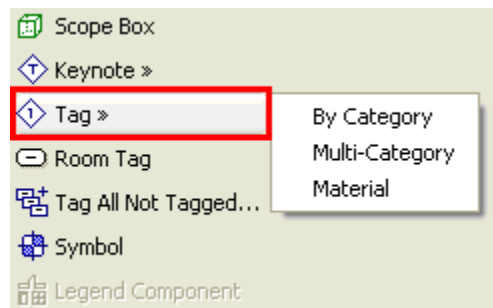


Figure 89: IndependentTag

Every category in the family library has a pre-made tag. Some tags are automatically loaded with the default Revit application template, while others are loaded manually. The IndependentTag objects return different categories based on the host element if it is created using the By Category option. For example, the Wall and Floor IndependentTag are respectively OST_WallTags and OST_FloorTags.

If the tag is created using the Multi-Category or Material style, their categories are respectively OST_MultiCategoryTags and OST_MaterialTags.

Similar to DetailCurve, NewTag() only works in the 2D view, otherwise an exception is thrown. The following code is an example of IndependentTag creation. Run it when the level view is the active view.

Note: You can't change the text displayed in the IndependentTag directly. You need to change the parameter that is used to populate tag text in the Family Type for the Element that's being tagged.

In the example below, that parameter is "Type Mark", although this setting can be changed in the Family Editor in the Revit UI.

Code Region 14-5: Creating an IndependentTag

```
private IndependentTag CreateIndependentTag(Autodesk.Revit.Document document, Wall wall)
{
    // make sure active view is not a 3D view
    Autodesk.Revit.Elements.View view = document.ActiveView;

    // define tag mode and tag orientation for new tag
    TagMode tagMode = TagMode.TM_ADDBY_CATEGORY;
    TagOrientation tagorn = TagOrientation.TAG_HORIZONTAL;

    // Add the tag to the middle of the wall
    LocationCurve wallLoc = wall.Location as LocationCurve;
    XYZ wallStart = wallLoc.Curve.get_EndPoint(0);
    XYZ wallEnd = wallLoc.Curve.get_EndPoint(1);
    XYZ wallMid = wallLoc.Curve.Evaluate(0.5, true);

    IndependentTag newTag = document.Create.NewTag(view, wall, true, tagMode, tagorn,
                                                    wallMid);

    if (null == newTag)
    {
        throw new Exception("Create IndependentTag Failed.");
    }

    // newTag.TagText is read-only, so we change the Type Mark type parameter to
    // set the tag text. The label parameter for the tag family determines
    // what type parameter is used for the tag text.

    WallType type = wall.WallType;

    Parameter foundParameter = type.get_Parameter("Type Mark");
    bool result = foundParameter.Set("Hello");

    // set leader mode free
    // otherwise leader end point move with elbow point

    newTag.LeaderMode = LeaderEndCondition.LEC_FREE;
    XYZ elbowPnt = wallMid + new XYZ(5.0, 5.0, 0.0);
    newTag.LeaderElbow = elbowPnt;
    XYZ headerPnt = wallMid + new XYZ(10.0, 10.0, 0.0);
    newTag.TagHeadPosition = headerPnt;

    return newTag;
}
```

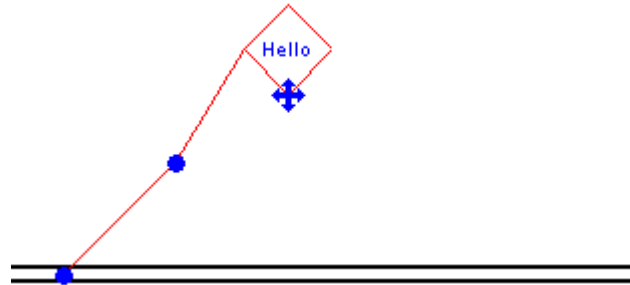


Figure 90: Create IndependentTag using sample code

14.4Text

In the API, the TextNote class represents Text. Its general features are as follows:

Function	API Method or Property
Add text to the application	Document.Create.NewTextNote() or Document.Create.NewTextNotes() methods
Get and set string from the text component	TextNote.Text property
Get and set text component position	TextNote.Coord Property
Get and set text component width	TextNote.Width Property
Get all text component leaders	TextNote.Leaders Property
Add a leader to the text component	TextNote.AddLeader() Method
Remove all leaders from the text component	TextNote.RemoveLeaders() Method

Table 34: General Features of TextNote

Revit supports two kinds of Leaders: straight leaders and arc leaders. Control the TextNote leader type using the TextNoteLeaderType enumerated type:

Function	Member Name
 —Add a right arc leader	TNLT_ARC_R
 —Add a left arc leader	TNLT_ARC_L
 —Add a right leader.	TNLT_STRAIGHT_R
 —Add a left leader.	TNLT_STRAIGHT_L

Table 35: Leader Types

Note: Straight leaders and arc leaders cannot be added to a Text type at the same time.

14.5Annotation Symbol

An annotation symbol is a symbol applied to a family to uniquely identify that family in a project.



Figure 91: Add annotation symbol

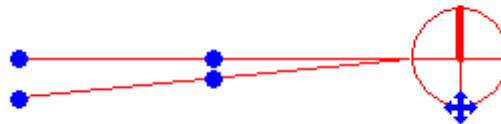


Figure 92: Annotation Symbol with two leaders

14.5.1 Create and Delete

The annotation symbol creation method is available in the `Creation.Document` class.

Code Region 14-6: `NewAnnotationSymbol()`

```
public AnnotationSymbol NewAnnotationSymbol(XYZ location, AnnotationSymbolType
annotationSymbolType, View specView)
```

The annotation symbol can be deleted using the `Document.Delete()` method.

14.5.2 Add and Remove Leader

Add and remove leaders using the `addLeader()` and `removeLeader()` methods.

Code Region 14-7: Using `addLeader()` and `removeLeader()`

```
public void AddAndRemoveLeaders(AnnotationSymbol symbol)
{
    int leaderSize = symbol.Leaders.Size;
    string message = "Number of leaders in Annotation Symbol before add: " + leaderSize;
    symbol.addLeader();
    leaderSize = symbol.Leaders.Size;
    message += "\nNumber of leaders after add: " + leaderSize;
    symbol.removeLeader();
    leaderSize = symbol.Leaders.Size;
    message += "\nNumber of leaders after remove: " + leaderSize;

    MessageBox.Show(message, "Revit");
}
```

15 Sketching

To create elements or edit their profiles in Revit, you must first create sketch objects. Examples of elements that require sketches include:

- Roofs
- Floors
- Stairs
- Railings.

Sketches are also required to define other types of geometry, such as:

- Extrusions
- Openings
- Regions

In the Revit Platform API, sketch functions are represented by 2D and 3D sketch classes such as the following:

- 2D sketch:
 - SketchPlane
 - Sketch
 - ModelCurve
 - and more
- 3D sketch:
 - GenericForm
 - Path3D

In addition to Sketch Elements, ModelCurve is also described in this chapter. For more details about Element Classification, see the [Elements Classification](#) section in the [Elements Essentials](#) chapter.

15.1 The 2D Sketch Class

The Sketch class represents enclosed curves in a plane used to create a 3D model. The key features are represented by the SketchPlane and CurveLoop properties.

When editing a Revit file, you cannot retrieve a Sketch object by iterating Document.Elements enumeration because all Sketch objects are transient Elements. When accessing the Family's 3D modeling information, Sketch objects are important to forming the geometry. For more details, refer to the [3D Sketch](#) section in this chapter.

The following picture shows the key classes to sketch in 2D.

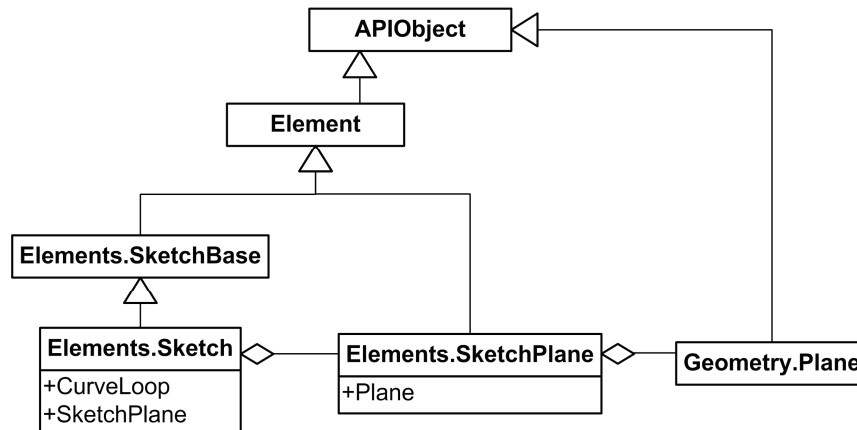


Figure 93: 2D Sketch diagram

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.

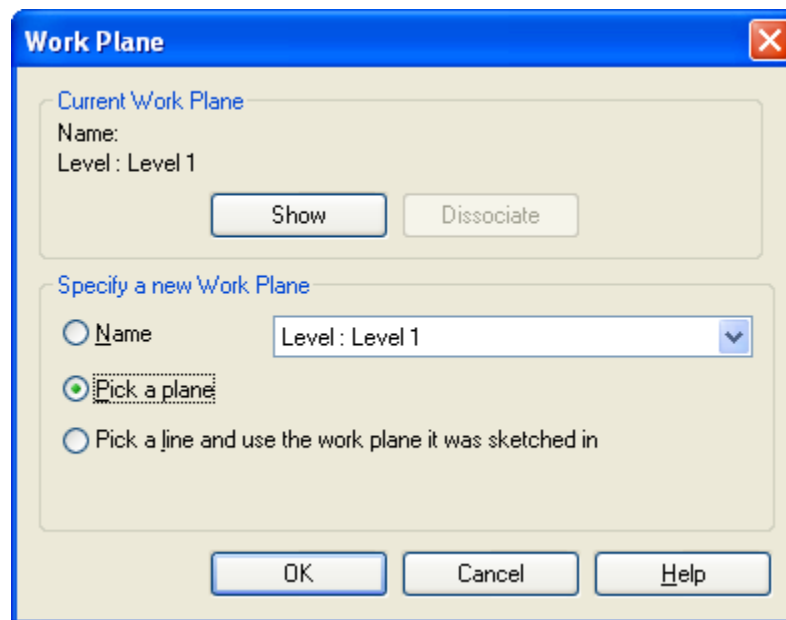


Figure 94: Pick a Plane to identify a new Work Plane

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.

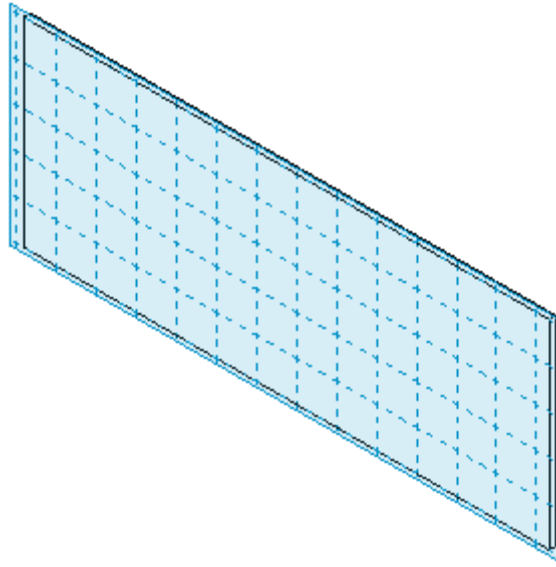


Figure 95: Pick a Plane on a wall as Work Plane

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:

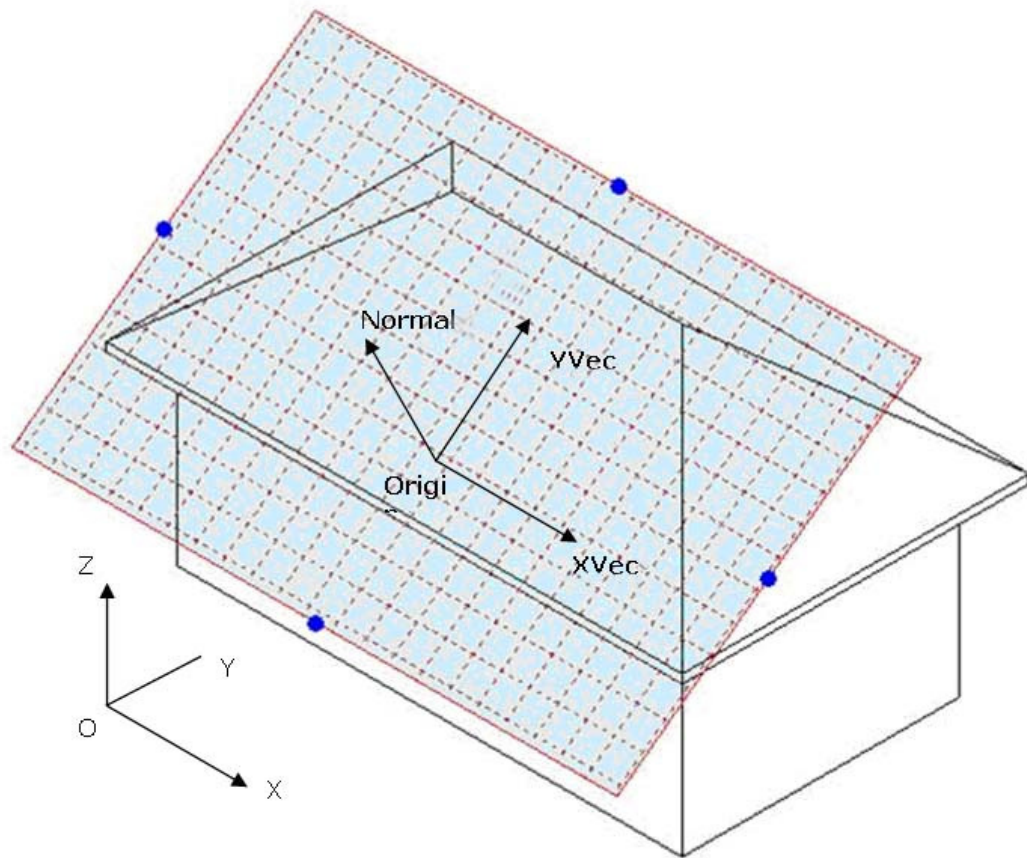


Figure 96: SketchPlane and Plane coordinate system

The following code sample illustrates how to create a new SketchPlane:

Code Region 15-1: Creating a new SketchPlane

```
private SketchPlane CreateSketchPlane(Autodesk.Revit.Application application)
{
    //try to create a new sketch plane
    XYZ newNormal = new XYZ(1, 1, 0); // the normal vector
    XYZ newOrigin = new XYZ(0, 0, 0); // the origin point
    // create geometry plane
    Plane geometryPlane = application.Create.NewPlane(newNormal, newOrigin);

    // create sketch plane
    SketchPlane sketchPlane = application.ActiveDocument.Create.NewSketchPlane(geometryPlane);

    return sketchPlane;
}
```

15.23D Sketch

3D Sketch is used to edit a family or create a 3D object. In the Revit Platform API, you can complete the 3D Sketch using the following classes.

- Extrusion
- Revolution

- Blend
- Sweep

In other words, there are four operations through which a 2D model turns into a 3D model. For more details about sketching in 2D, refer to the [2D Sketch](#) section in this chapter.

15.2.1 Extrusion

Revit uses extrusions to define 3D geometry for families. You create an extrusion by defining a 2D sketch on a plane; Revit then extrudes the sketch between a start and an end point.

Query the Extrusion Form object for a generic form to use in family modeling and massing. The Extrusion class has the following properties:

Property	Description
ExtrusionStart	Returns the Extrusion Start point. It is a Double type.
ExtrusionEnd	Returns the Extrusion End point. It is a Double type.
Sketch	Returns the Extrusion Sketch. It contains a sketch plane and some curves.

Table 36: Extrusion Properties

The value of the ExtrusionStart and ExtrusionEnd properties is consistent with the parameters in the Revit UI. The following figure illustrates the corresponding parameters and the Extrusion result.

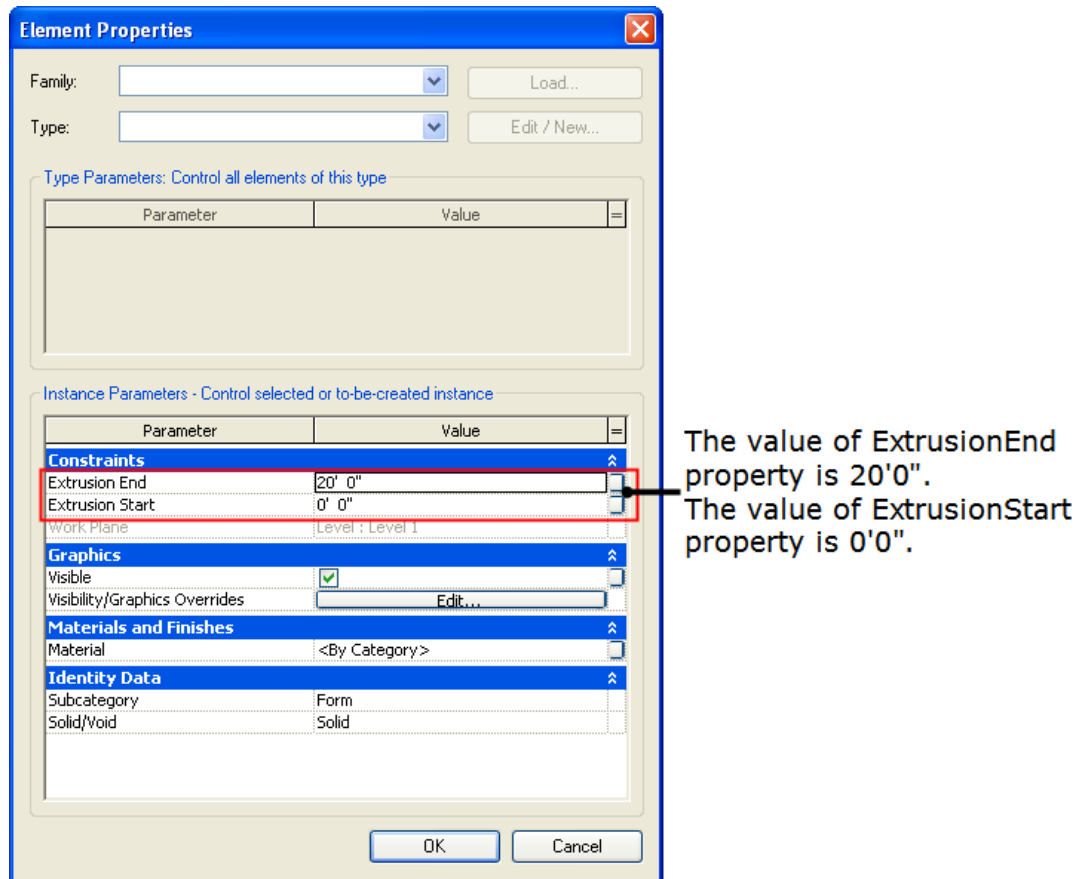


Figure 97: Extrusion parameters in the UI

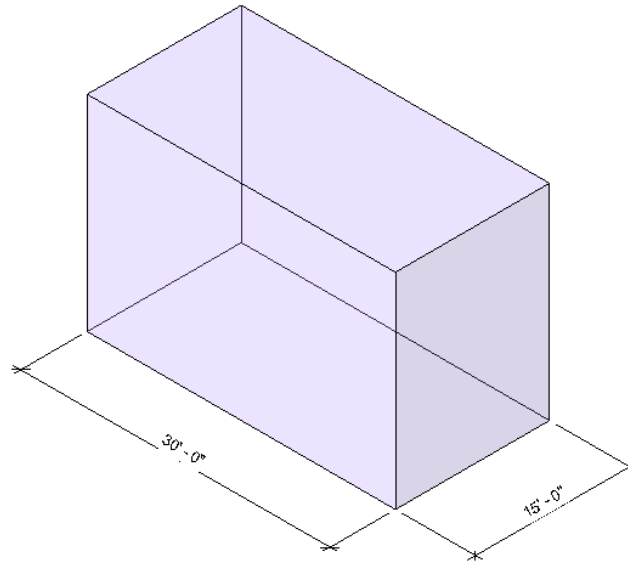


Figure 98: Extrusion result

15.2.2 Revolution

The Revolve command creates geometry that revolves around an axis. You can use the revolve command to create door knobs or other knobs on furniture, a dome roof, or columns.

Query the Revolution Form object for a generic form to use in family modeling and massing. The Revolution class has the following properties:

Property	Description
Axis	Returns the Axis. It is a ModelLine object.
EndAngle	Returns the End Angle. It is a Double type.
Sketch	Returns the Extrusion Sketch. It contains a SketchPlane and some curves.

Table 37: Revolution Properties

EndAngle is consistent with the same parameter in the Revit UI. The following pictures illustrate the Revolution corresponding parameter, the sketch, and the result.

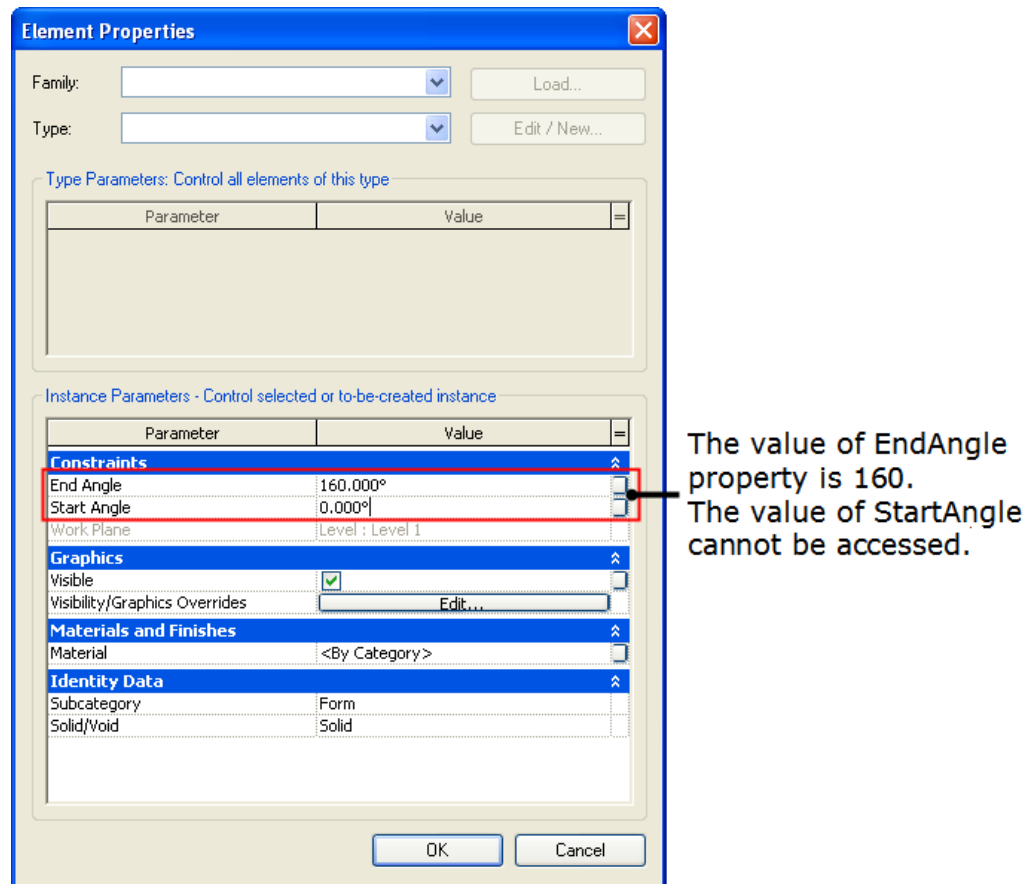


Figure 99: Corresponding parameter

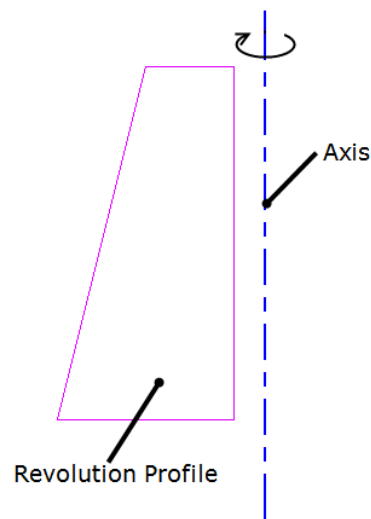


Figure 100: Revolution sketch

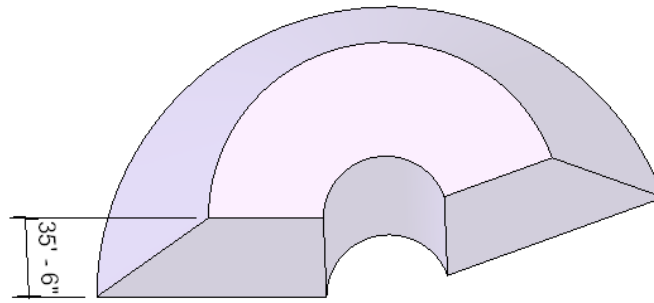


Figure 101: Revolution result

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

15.2.3 Blend

The Blend command blends two profiles together. For example, if you sketch a large rectangle and a smaller rectangle on top of it, Revit Architecture blends the two shapes together.

Query the Blend Form object for a generic form to use in family modeling and massing. The Blend class has the following properties:

Property	Description
BottomSketch	Returns the Bottom Sketch. It is a Sketch object.
TopSketch	Returns the Top Sketch Blend. It is a Sketch object.
FirstEnd	Returns the First End. It is a Double type.
SecondEnd	Returns the Second End. It is a Double type.

Table 38: Blend Properties

The FirstEnd and SecondEnd property values are consistent with the same parameters in the Revit UI. The following pictures illustrate the Blend corresponding parameters, the sketches, and the result.

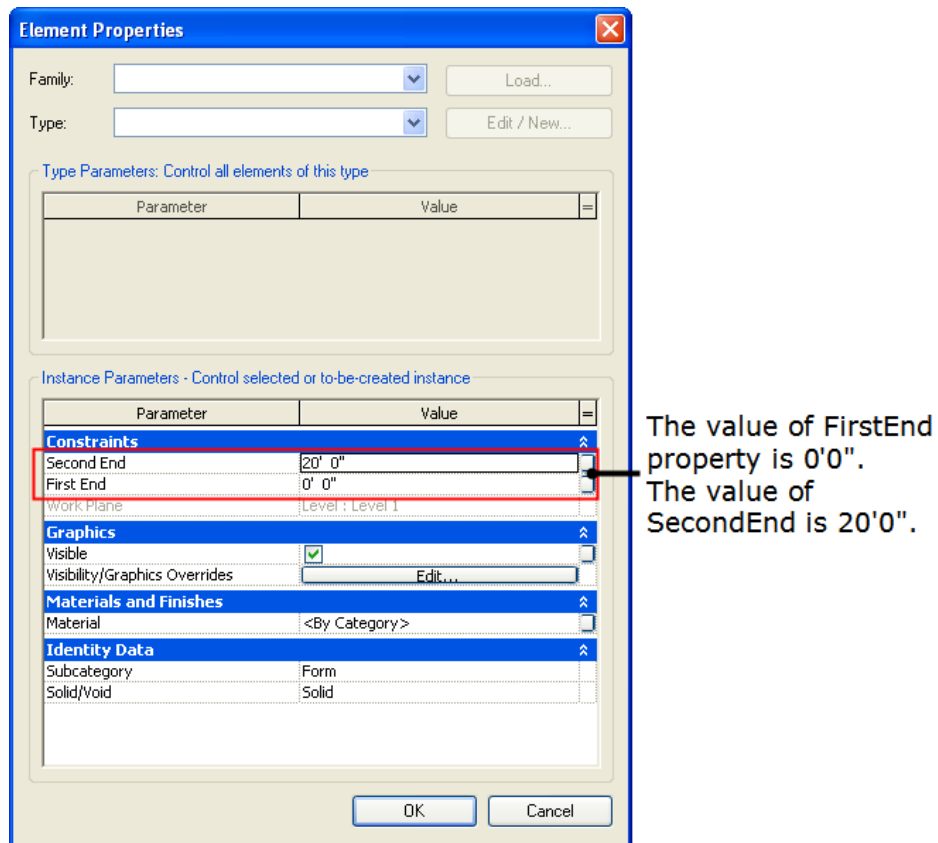


Figure 102: Blend parameters in the UI

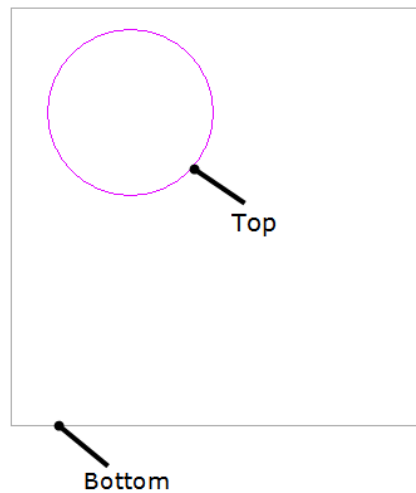


Figure 103: Blend top sketch and bottom sketch

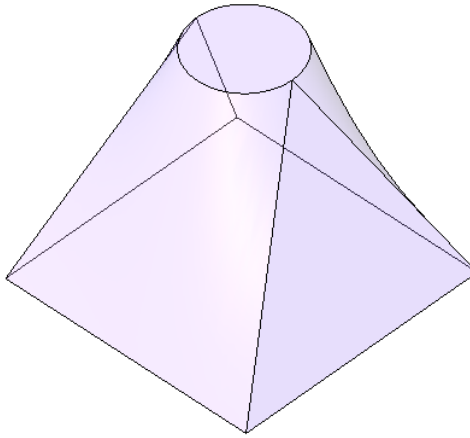


Figure 104: Blend result

15.2.4 Sweep

The Sweep command sweeps one profile along a created 2D path or selected 3D path. The path may be an open or closed loop, but must pierce the profile plane.

Query the Sweep Form object for a generic form for use in family modeling and massing. The Sweep class has the following properties:

Property	Description
Path3d	Returns the 3D Path Sketch. It is a Path3D object.
PathSketch	Returns the Plan Path Sketch. It is a Sketch object.
ProfileSketch	Returns the profile Sketch. It is a Sketch object.
EnableTrajSegmentation	Returns the Trajectory Segmentation state. It is a Boolean.
MaxSegmentAngle	Returns the Maximum Segment Angle. It is a Double type.

Table 39: Sweep Properties

Creating a 2D Path is similar to other forms. The 3D Path is fetched by picking the created 3D curves.

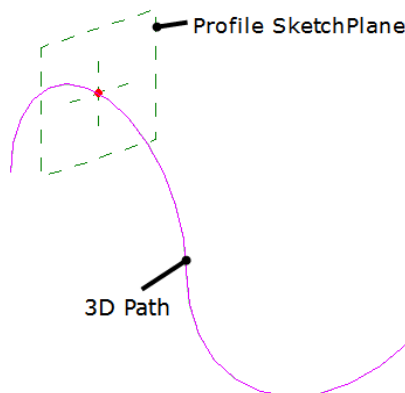


Figure 105: Pick the Sweep 3D path

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.

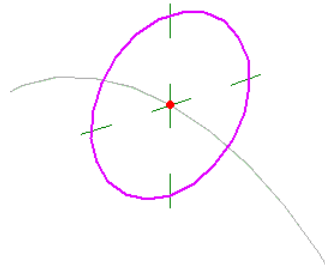


Figure 106: Sweep profile sketch

Note: The ProfileSketch is perpendicular to the path.

Segmented sweeps are useful for creating mechanical duct work elbows. Create a segmented sweep by setting two sweep parameters and sketching a path with arcs.

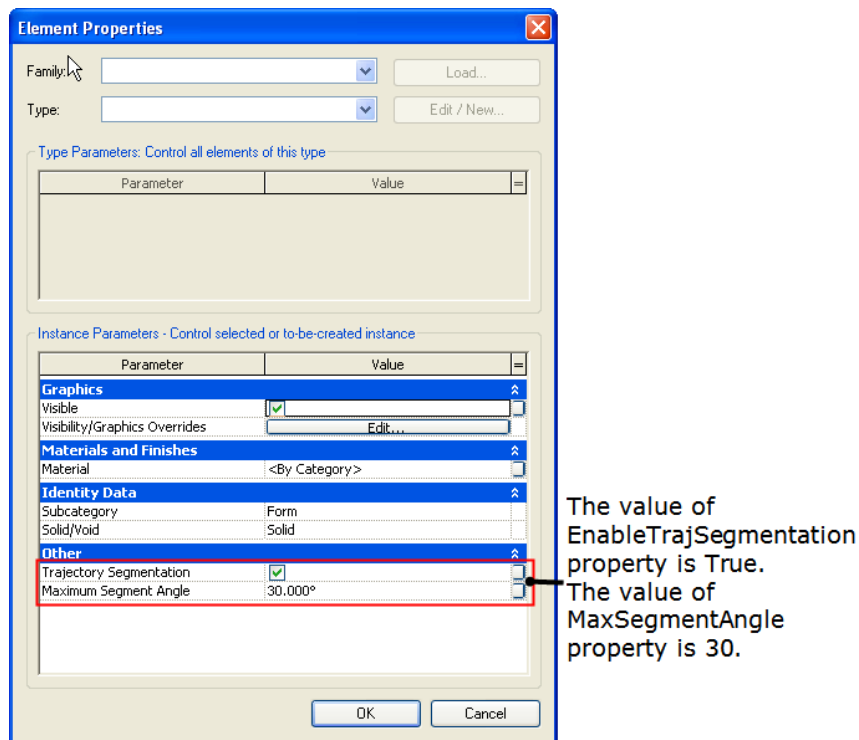


Figure 107: Corresponding segment settings in the UI

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°.

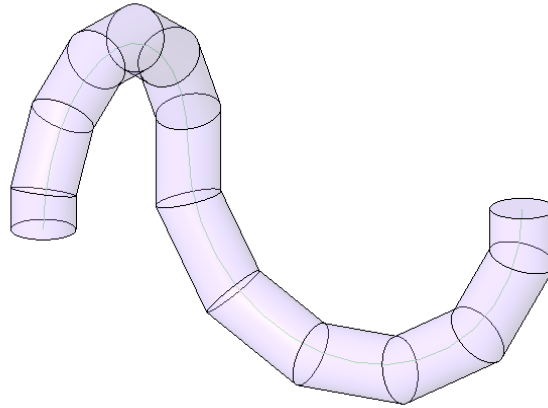


Figure 108: Sweep result

15.3 ModelCurve

ModelCurve represents model lines in the project. It exists in 3D space and is visible in all views. The following picture shows the ModelCurve inheritance hierarchy.

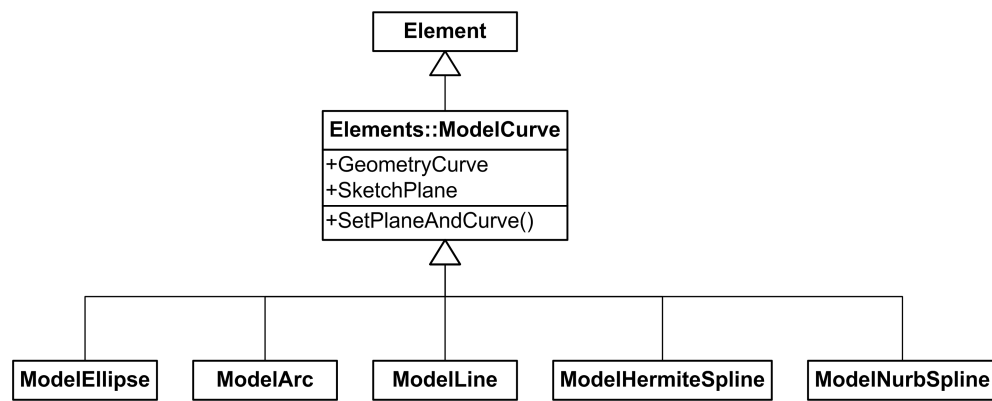


Figure 109: ModelCurve diagram

The following pictures illustrate the four ModelCurve derived classes:



Figure 110: ModelLine and ModelArc



Figure 111: ModelEllipse and ModelNurbSpline

15.3.1 Creating a ModelCurve

The key to creating a ModelCurve is to create the Geometry.Curve and SketchPlane where the Curve is located. Based on the Geometry.Curve type you input, the corresponding ModelCurve returned can be downcast to its correct type.

The following sample illustrates how to create a new model curve (ModelLine and ModelArc):

Code Region 15-2: Creating a new model curve

```
Autodesk.Revit.Application application = document.Application;

// Create a geometry line in revit application
XYZ startPoint = new XYZ(0, 0, 0);
XYZ endPoint = new XYZ(10, 10, 0);
Line geomLine = application.Create.NewLine(startPoint, endPoint, true);

// Create a geometry arc in revit application
XYZ end0 = new XYZ(1, 0, 0);
XYZ end1 = new XYZ(10, 10, 10);
XYZ pointOnCurve = new XYZ(10, 0, 0);
Arc geomArc = application.Create.NewArc(end0, end1, pointOnCurve);

// Create a geometry plane in revit application
XYZ origin = new XYZ(0, 0, 0);
XYZ normal = new XYZ(1, 1, 0);
Plane geomPlane = application.Create.NewPlane(normal, origin);

// Create a sketch plane in current document
SketchPlane sketch = document.Create.NewSketchPlane(geomPlane);

// Create a ModelLine element using the created geometry line and sketch plane
ModelLine line = document.Create.NewModelCurve(geomLine, sketch) as ModelLine;

// Create a ModelArc element using the created geometry arc and sketch plane
ModelArc arc = document.Create.NewModelCurve(geomArc, sketch) as ModelArc;
```

15.3.2 Properties

ModelCurve has properties that help you set specific GeometryCurves. In this section, the GeometryCurve and LineStyle properties are introduced.

15.3.2.1 GeometryCurve

The GeometryCurve property is used to get or set the model curve's geometry curve. Except for ModelHermiteSpline, you can get different Geometry.Curves from the four ModelCurves;

- Line
- Arc
- Ellipse
- Nurbspline.

The following code sample illustrates how to get a specific Curve from a ModelCurve.

Code Region 15-3: Getting a specific Curve from a ModelCurve

```
//get the geometry modelCurve of the model modelCurve
Autodesk.Revit.Geometry.Curve geoCurve = modelCurve.GeometryCurve;

if (geoCurve is Autodesk.Revit.Geometry.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.

15.3.2.2 LineStyle

Line style does not have a specific class but is represented by the Document.Element class.

- All line styles for a ModelCurve are available from the LineStyles property in the Element Properties dialog box.
- The Element object that represents line style cannot provide information for all properties. Most properties will return null or an empty collection.
- The only information returned is the following:
 - Id
 - UniqueId
 - Name.
- Use the getLineStyle() method to retrieve the current line style or use the setLineStyle() method to set the current line style to one returned from the LineStyles property.

16 Views

Views are images produced from a Revit model with privileged access to the data stored in the documents. They can be graphics, such as plans, or text, such as schedules. Each project document has one or more different views. The last focused window is the active view.

In this chapter, you learn how views are generated, the types of views supported by Revit, and the features for each view.

16.1 Overview

This section is a high-level overview discussing the following:

- How views are generated
- View types
- Element visibility
- Creating and deleting views.

16.1.1 View Process

The following figure illustrates how a view is generated.

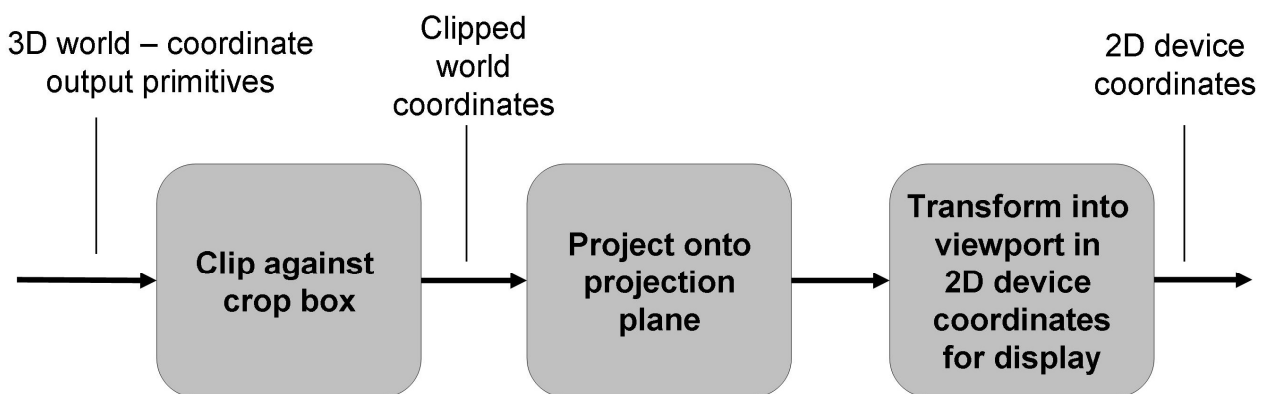


Figure 112: Create view process

Each view is generated by projecting a three-dimensional object onto a two-dimensional projection plane. Projections are divided into two basic classes:

- Perspective
- Parallel

After the projection type is determined, you must specify the conditions under which the 3D model is needed and the scene is to be rendered. For more information about projection, refer to the [View3D](#) section.

World coordinates include the following:

- The viewer's eye position
- The viewing plane location where the projection is displayed.

Revit uses two coordinate systems:

- The global or model space coordinates where the building exists
- The viewing coordinate system.

The viewing coordinate system represents how the model is presented in the observer's view. Its origin is the viewer's eye position whose coordinates in the model space are retrieved by the `View.Origin` property. The X, Y, and Z axes are represented by the `View.RightDirection`, `View.UpDirection`, and `View.ViewDirection` properties respectively.

- `View.RightDirection` is towards the right side of the screen.
- `View.UpDirection` towards the up side of the screen.
- `View.ViewDirection` from the screen to the viewer.

The viewing coordinate system is right-handed. For more information, see the [Perspective Projection picture](#) and the Parallel Projection picture in the [View3D](#) section in this chapter.

Some portions of a 3D model space that do not display, such as those that are behind the viewer or too far away to display clearly, are excluded before being projected onto the projection plane. This action requires cropping the view. The following rules apply to cropping:

- Elements outside of the crop box are no longer in the view.
- The `View.CropBox` property provides the geometry information for the box. It returns an instance of `BoundingBoxXYZ` indicating a rectangular parallelepiped in the viewing coordinate system. The coordinate system and the crop box shape are illustrated in the [View3D](#) section.
- The `View.CropBoxVisible` property determines whether the crop box is visible in the view.
- The `View.CropBoxActive` property determines whether the crop box is actually being used to crop the view.

After cropping, the model is projected onto the projection plane. The following rules apply to the projection:

- The projection contents are mapped to the screen view port for display.
- During the mapping process, the projection contents are scaled so that they are shown properly on the screen.
- The `View.Scale` property is the ratio of the actual model size to the view size.
- The view boundary on paper is the crop region, which is a projection of the crop box on the projection plane.
- The size and position of the crop region is determined by the `View.OutLine` property.

16.1.2 View Types

A project model can have several view types. The following picture demonstrates the different types of views in the Project browser.

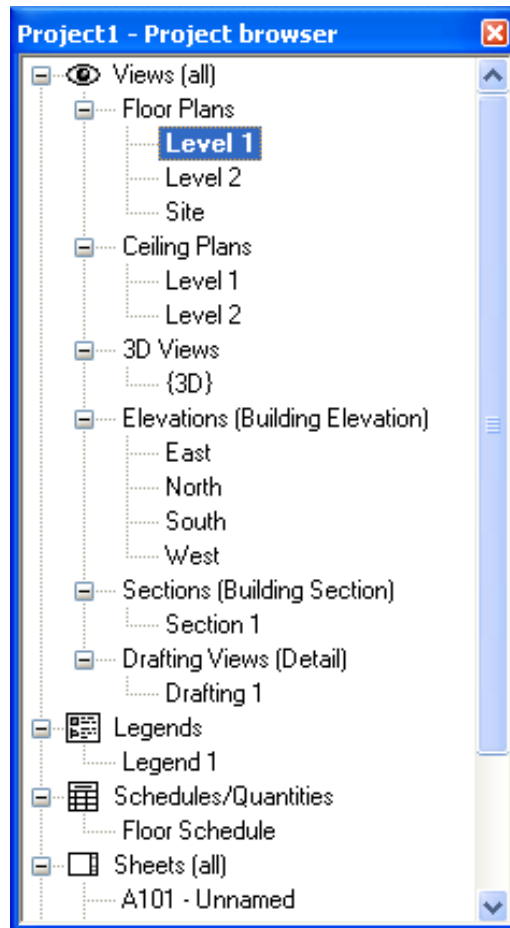


Figure 113: Different views in the Project browser

In the API, there are two ways to classify views. The first way is by using the view element `View.ViewType` property. It returns an enumerated value indicating the view type. The following table lists all available view types.

Member Name	Description
AreaPlan	Area view.
CeilingPlan	Reflected ceiling plan view.
ColumnSchedule	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.

Member Name	Description
PressureLossReport	Pressure Loss Report view.
Rendering	Rendering view.
Report	Report view.
Schedule	Schedule view.
Section	Cross section view.
ThreeD	3-D view.
Undefined	Undefined/unspecified view.
Walkthrough	Walkthrough view.

Table 40: Autodesk.Revit.Enums.ViewTypes

The second way to classify views is by the class type. The following figure illustrates the view class hierarchy in the Revit Platform API.

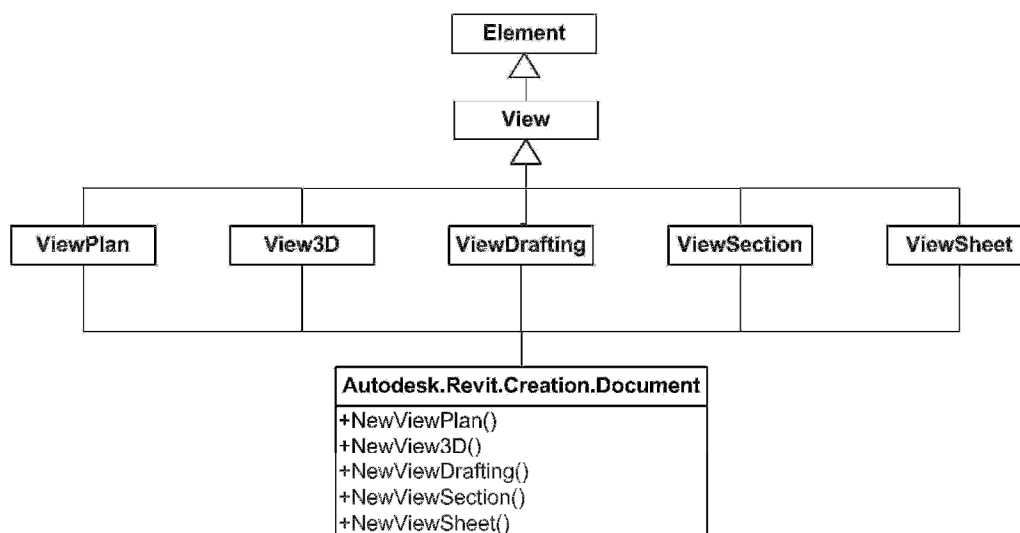


Figure 114: View classes in the Revit Platform API diagram

The following table lists the view types and the corresponding views in the Project browser.

Project Browser Views	View Type	Class Type
Area Plans	ViewType.AreaPlan	Elements.ViewPlan
Ceiling Plans	ViewType.CeilingPlan	Elements.ViewPlan
Graphic Column Schedule	ViewType.ColumnSchedule	Elements.View
Detail Views	ViewType.Detail	Elements.ViewSection
Drafting Views	ViewType.DraftingView	Elements.ViewDrafting
Sheets	ViewType.DrawingSheet	Elements.ViewSheet
Elevations	ViewType.Elevation	Elements.View
Structural Plans (Revit Structure)	ViewType.EngineeringPlan	Elements.ViewPlan
Floor Plans	ViewType.FloorPlan	Elements.ViewPlan
Legends	ViewType.Legend	Elements.View
Reports (Revit MEP)	ViewType.LoadsReport	Elements.View

Project Browser Views	View Type	Class Type
Reports (Revit MEP)	ViewType.PanelSchedule	Elements.View
Reports (Revit MEP)	ViewType.PresureLossReport	Elements.View
Renderings	ViewType.Rendering	Elements.ViewDrafting
Reports	ViewType.Report	Elements.View
Schedules/Quantities	ViewType.Schedule	Elements.View
Sections	ViewType.Section	Elements.View
3D Views	ViewType.ThreeD	Elements.View3D
Walkthroughs	ViewType.Walkthrough	Elements.View3D

Table 41: Project Browser Views

The following example shows how to get the class type of a view, in this case, the ActiveView of the Document.

Code Region 16-1: Determining the View class type

```
// Get the active view of the current document.
Autodesk.Revit.Elements.View view = document.ActiveView;

// Get the class type of the active view, and format the prompt string
String prompt = "Revit is currently in ";
if (view is Autodesk.Revit.Elements.View3D)
{
    prompt += "3D view.";
}
else if (view is Autodesk.Revit.Elements.ViewSection)
{
    prompt += "section view.";
}
else if (view is Autodesk.Revit.Elements.ViewSheet)
{
    prompt += "sheet view.";
}
else if (view is Autodesk.Revit.Elements.ViewDrafting)
{
    prompt += "drafting view.";
}
else
{
    prompt += "normal view, the view name is " + view.Name;
}

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

This example shows how to use the ViewType property of a view to determine the view's type.

Code Region 16-2: Determining the View type

```

public void GetViewType(Autodesk.Revit.Elements.View view)
{
    // Get the view type of the given view, and format the prompt string
    String prompt = "The view is ";

    switch (view.ViewType)
    {
        case Autodesk.Revit.Enums.ViewType.AreaPlan:
            prompt += "an area view.";
            break;
        case Autodesk.Revit.Enums.ViewType.CeilingPlan:
            prompt += "a reflected ceiling plan view.";
            break;
        case Autodesk.Revit.Enums.ViewType.ColumnSchedule:
            prompt += "a column schedule view.";
            break;
        case Autodesk.Revit.Enums.ViewType.CostReport:
            prompt += "a cost report view.";
            break;
        case Autodesk.Revit.Enums.ViewType.Detail:
            prompt += "a detail view.";
            break;
        case Autodesk.Revit.Enums.ViewType.DraftingView:
            prompt += "a drafting view.";
            break;
        case Autodesk.Revit.Enums.ViewType.DrawingSheet:
            prompt += "a drawing sheet view.";
            break;
        case Autodesk.Revit.Enums.ViewType.Elevation:
            prompt += "an elevation view.";
            break;
        case Autodesk.Revit.Enums.ViewType.EngineeringPlan:
            prompt += "an engineering view.";
            break;
        case Autodesk.Revit.Enums.ViewType.FloorPlan:
            prompt += "a floor plan view.";
            break;
        // ...
        default:
            break;
    }

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

```

16.1.3 Element Visibility in a View

Views keep track of visible elements. All elements that are graphical and visible in the view can be retrieved using the `View.Elements` property. This property returns a set of elements visible in 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.

Elements are shown or hidden in a view by category.

- The `View.getVisibility()` method queries a category to determine if it is visible or invisible in the view.
- The `View.setVisibility()` method sets all elements in a specific category to visible or invisible.

The set returned by `View.Elements` only contains elements visible in the current view. You cannot retrieve elements that are not graphical or elements that are invisible. `Document.Elements` 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.

Note: `Autodesk.Revit.Document` also has a property named `Elements`. It is important to distinguish the `Autodesk.Revit.Elements.View.Elements` property from the `Autodesk.Revit.Document.Elements` property based on the previous description and the following code sample.

The following code sample counts the number of elements in the active document and active view. The number of elements in the active view changes if you hide some elements while the number of elements in the document is constant.

Code Region 16-3: Counting elements in the active view

```
private void CountElements(Autodesk.Revit.Document document)
{
    // Count the number of elements in the document
    int count = 0;
    ElementIterator iter = document.Elements;
    iter.Reset();
    while (iter.MoveNext())
    {
        Autodesk.Revit.Element elem = iter.Current as Autodesk.Revit.Element;
        if (null != elem)
        {
            ++count;
        }
    }
    StringBuilder message = new StringBuilder();
    message.AppendLine("Elements within Document: " + count.ToString());
    message.AppendLine("Elements within active View: "
        + document.ActiveView.Elements.Size.ToString());

    MessageBox.Show(message.ToString());
}
```

16.1.4 Creating and Deleting Views

The Revit Platform API provides five methods to create the corresponding view elements derived from Autodesk.Revit.Elements.View class.

Method	Parameters
View3D NewView3D(XYZ viewDirection)	viewDirection: Vector pointing towards the viewer's eye.
ViewPlan NewViewPlan(string pViewName, Level pLevel, ViewType viewType)	pViewName: Name for the new plan view. It must be unique or a null pointer. pLevel: Level associated with the plan view. viewType: Type of plan view created. It must be Floor Plan or Ceiling Plan (Structural Plan in Structure).
ViewSection NewViewSection(BoundingBoxXYZ box)	box: View orientation and bounds. The X axis points towards the right of screen; Y - towards up; Z - towards the user.
ViewSheet NewViewSheet(FamilySymbol titleBlock)	titleBlock: The titleblock family symbol applied to the sheet.
ViewDrafting NewViewDrafting()	

Table 42: Create View Element Methods

If a view is created successfully, these methods return a reference to the view, otherwise it returns null. The methods are described in the following sections.

Delete a view by using the Document.Delete method with the view ID. You can also delete elements associated with a view. For example, deleting the level element causes Revit to delete the corresponding plan view or deleting the camera element causes Revit to delete the corresponding 3D view.

Note: You cannot gain access to Schedule views in the Revit Platform API as there is no NewScheduleView() method.

16.2 The View3D Class

View3D is a freely-oriented three-dimensional view. There are two kinds of 3D views, perspective and orthographic. The difference is based on the projection ray relationship. The View3D.IsPerspective property indicates whether a 3D view is perspective or orthographic.

16.2.1 Perspective View

The following picture illustrates how a perspective view is created.

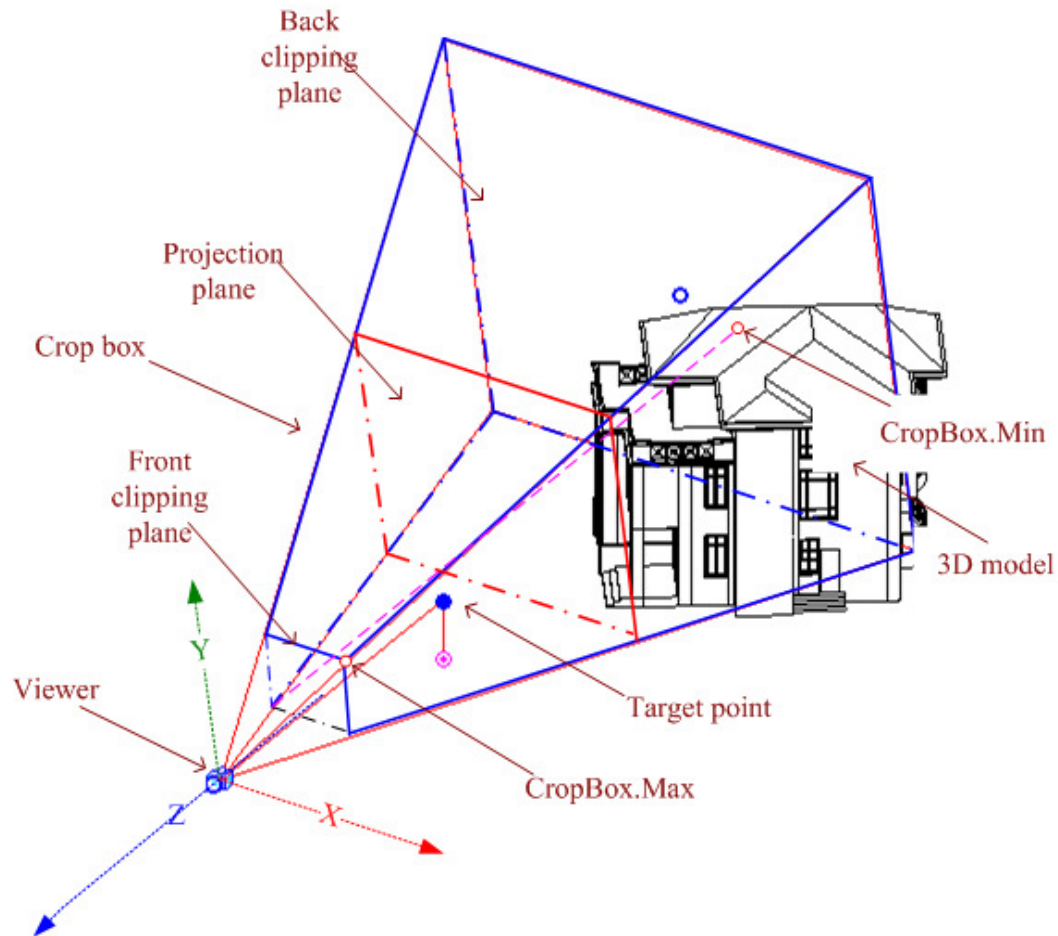


Figure 115: Perspective projection

- The straight projection rays pass through each point in the model and intersect the projection plane to form the projection contents.
- To facilitate the transformation from the world coordinate onto the view plane, the viewing coordinate system is based on the viewer.
- Its origin, represented by the View.Origin property, is the viewer position.
- The viewer's world coordinates are retrieved using the View3D.EyePosition property. Therefore, in 3D views, View.Origin is always equal to View3D.EyePosition.
- As described, the viewing coordinate system is determined as follows:
 - The X-axis is determined by View.RightDirection.
 - The Y-axis is determined by View.UpDirection.
 - The Z-axis is determined by View.ViewDirection.
- The view direction is from the target point to the viewer in the 3D space, and from the screen to the viewer in the screen space.

The perspective view crop box is part of a pyramid with the apex at the viewer position. It is the geometry between the two parallel clip planes. The crop box bounds the portion of the model that is clipped out and projected onto the view plane.

- The crop box is represented by the View.CropBox property, which returns a BoundingBoxXYZ object.

- The CropBox.Min and CropBox.Max points are marked in the previous picture. Note that the CropBox.Min point in a perspective view is generated by projecting the crop box front clip plane onto the back clip plane.

Crop box coordinates are based on the viewing coordinate system. Use Transform.OfPoint() to transform CropBox.Min and CropBox.Max to the world coordinate system. For more detail about Transform, refer to the [Geometry.Transform](#) section in the [Geometry](#) chapter.

The project plane plus the front and back clip plane are all plumb to the view direction. The line between CropBox.Max and CropBox.Min is parallel to the view direction. With these factors, the crop box geometry can be calculated.

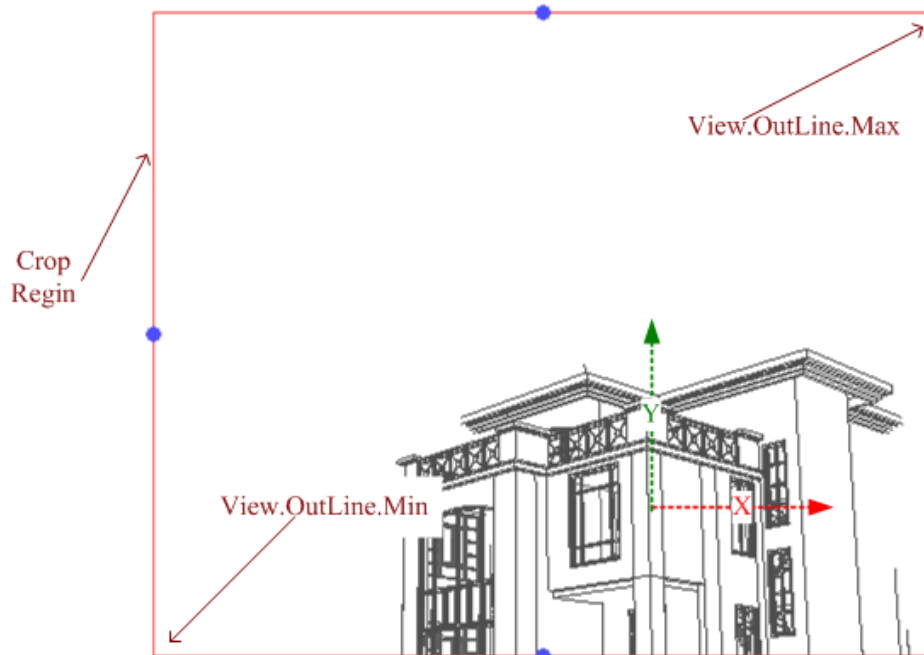


Figure 116: Perspective 3D view

The previous picture shows the projection plane on screen after cropping. The crop region is the rectangle intersection of the projection plane and crop box.

- Geometry information is retrieved using the View.CropRegion property. This property returns a BoundingBoxUV instance.
- The View.OutLine.Max property points to the upper right corner.
- The View.OutLine.Min property points to the lower left corner.
- Like the crop box, the crop region coordinates are based on the viewing coordinate system. The following expressions are equal.

$\text{View.CropBox.Max.X(Y)} / \text{View.OutLine.Max.X(Y)}$

$= \text{View.CropBox.Min.X(Y)} / \text{View.OutLine.Min.X(Y)}$

Since the size of an object's perspective projection varies inversely with the distance from that object to the center of the projection, scale is meaningless for perspective views. The perspective 3D view Scale property always returns zero. Currently, the API cannot create Perspective views.

16.2.2 Orthographic View

Autodesk.Revit.Creation.Document provides the NewView3D method to create an orthographic 3D view.

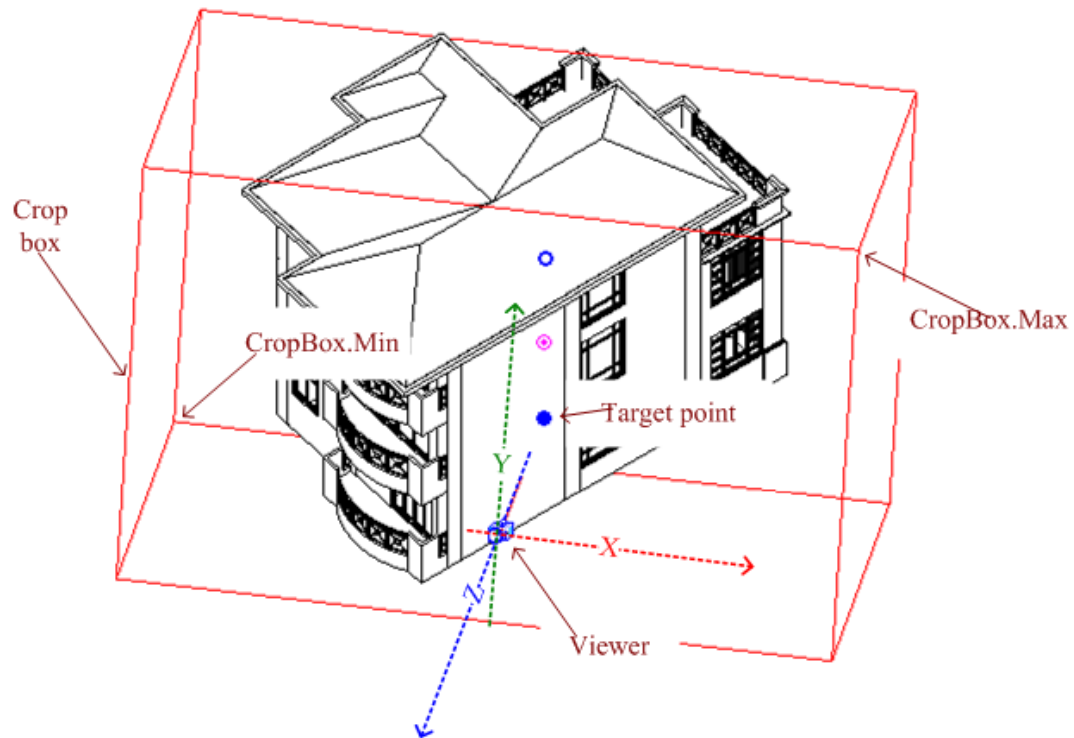


Figure 117: Parallel projection

Orthographic views are generated using parallel projection rays by projecting the model onto a plane that is normal to the rays. The viewing coordinate system is similar to the perspective view, but the crop box is a parallelepiped with faces that are parallel or normal to the projection rays. The View.CropBox property points to two diagonal corners whose coordinates are based on the viewing coordinate system.

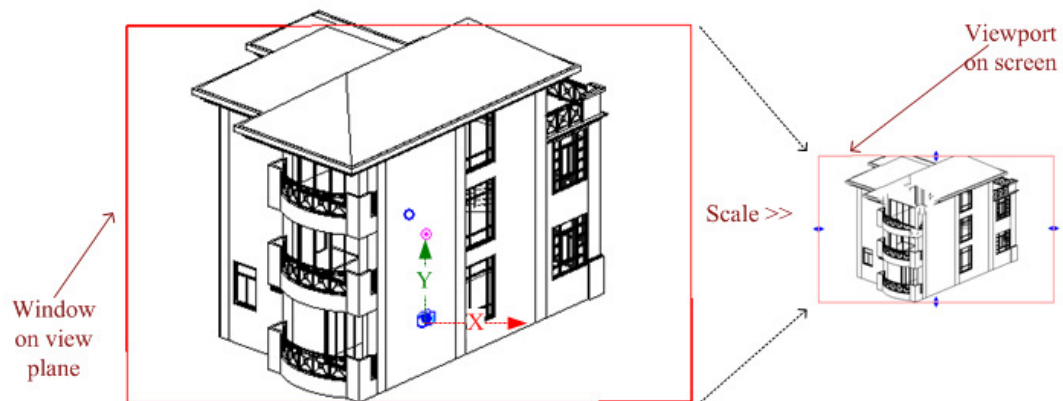


Figure 118: Scale the window on view plane to screen viewport

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:

$\text{View.CropBox.Max.X(Y)} / \text{View.OutLine.Max.X(Y)}$

$== \text{View.CropBox.Min.X(Y)} / \text{View.OutLine.Min.X(Y)}$

$== \text{View.Scale}$

Code Region 16-4: NewView3D()

```
public View3D NewView3D(XYZ viewDirection);
```

Create an orthographic 3D view by passing the view vector to the Autodesk.Revit.Creation.Document.NewView3D method. This vector can be a unit vector or not. Revit determines the following:

- Position of the viewer.
- How to create the viewing coordinate system using the view direction.
- How to create the crop box to crop the model.

Once the view is created, you can resize the crop box to view different portions of the model. The API does not support modifying the viewing coordinate system.

The following code sample illustrates how to create a 3D view. The created view is orthographic.

Code Region 16-5: Creating a 3D view

```
// Create a new View3D
XYZ direction = new XYZ(1, 1, 1);
View3D view3D = document.Create.NewView3D(direction);
if (null == view3D)
{
    throw new Exception("Failed to create new View3D");
}
```

16.2.3 3D Views SectionBox

Each view has a crop box. The crop box focuses on a portion of the model to project and show in the view. For 3D views, there is another box named section box.

- The section box determines which model portion appears in a 3D view.
- The section box is used to clip the 3D model's visible portion.
- The part outside the box is invisible even if it is in the crop box.
- The section box is different from the crop box in that it can be rotated and moved with the model.

The section box is particularly useful for large models. For example, if you want to render a large building, use a section box. The section box limits the model portion used for calculation. To display the section box, in the 3D view Element Properties dialog box, select Section Box in the Extents section. You can also set it using the API:

Code Region 16-6: Showing the Section Box

```
private void ShowHideSectionBox(Autodesk.Revit.Elements.View3D view3D)
{
    foreach (Parameter p in view3D.Parameters)
```



```

{
    // Get Section Box parameter
    if (p.Definition.Name.Equals("Section Box"))
    {
        // Show Section Box
        p.Set(1);
        // Hide Section Box
        // p.Set(0);
        break;
    }
}

```

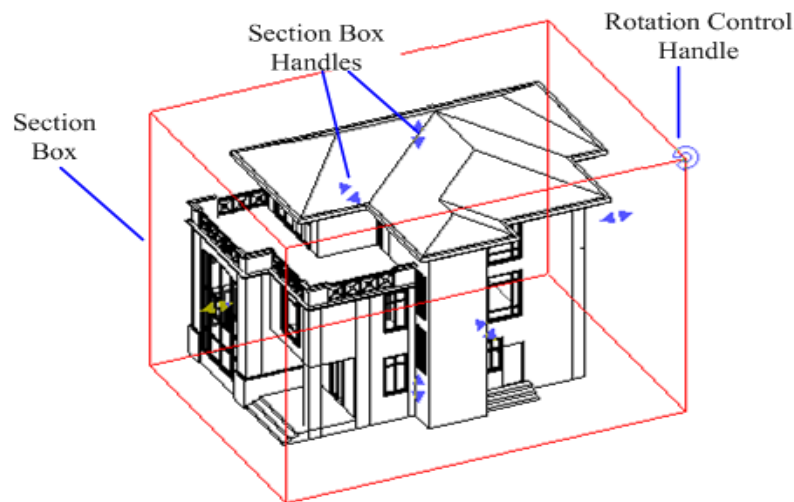


Figure 119: Section box

The `View3D.SectionBox` property is used to get and change the box extents. In some cases, setting the `View3D.SectionBox` can have a side effect. Setting the property to certain values can change the box capacity and display it in the view. However, you can assign a null value to the `SectionBox` to keep the modified value and make the section box invisible in the view. To avoid displaying the section box, change the section box value, then set the section box to null. The following code sample illustrates this process. Notice it only works when the Section Box check box is selected in the View property dialog box.

Code Region 16-7: Hiding the Section Box

```

private void ExpandSectionBox(Autodesk.Revit.Elements.View3D view)
{
    // The original section box
    BoundingBoxXYZ sectionBox = view.SectionBox;

    // Expand the section box
    XYZ deltaXYZ = sectionBox.Max - sectionBox.Min;
    sectionBox.Max += deltaXYZ / 2;
    sectionBox.Min -= deltaXYZ / 2;

    //After resetting the section box, it will be shown in the view.
}

```

```
//It only works when the Section Box check box is
//checked in View property dialog.
view.SectionBox = sectionBox;

//Setting the section box to null will make it hidden.
view.SectionBox = null;          // line x
}
```

Note: If you set view.SectionBox to null, it has the same effect as hiding the section box using the Section Box parameter. The current section box is stored by view and is restored when you show the section box using the SectionBox parameter.

16.3 ViewPlan

Plan views are level-based. There are two types of plan views, floor plan view and ceiling plan view.

- Generally the floor plan view is the default view opened in a new project.
- Most projects include at least one floor plan view and one ceiling plan view.
- Plan views are usually created after adding new levels to the project.

Adding new levels using the API does not add plan views automatically.

Autodesk.Revit.Creation.Document provides a NewViewPlan() method to create a plan view.

Code Region 16-8: NewViewPlan()

```
public ViewPlan NewViewPlan(string pViewName, Level pLevel, ViewType viewType);
```

The viewType parameter must be FloorPlan or CeilingPlan. The level parameter represents a level element in the project to which the plan view is associated.

The following code creates a floor plan and a ceiling plan based on a certain level.

Code Region 16-9: Creating a floor plan and ceiling plan

```
// Create a Level and a Floor Plan based on it
double elevation = 10.0;
Level level1 = document.Create.NewLevel(elevation);
ViewPlan floorView = document.Create.NewViewPlan(null, level1, ViewType.FloorPlan);

// Create another Level and a Ceiling Plan based on it
elevation += 10.0;
Level level2 = document.Create.NewLevel(elevation);
ViewPlan ceilingView = document.Create.NewViewPlan(null, level2, ViewType.CeilingPlan);
```

16.4 ViewDrafting

The drafting view is not associated with the model. It allows the user to create detail drawings that are not included in the model.

- In the drafting view, the user can create details in different view scales (coarse, fine, or medium).
- You can use 2D detailing tools, including:

- Detail lines
- Detail regions
- Detail components
- Insulation
- Reference planes
- Dimensions
- Symbols
- Text

These tools are the same tools used to create a detail view.

- Drafting views do not display model elements.

Use the `Autodesk.Revit.Creation.NewViewDrafting()` method to create a drafting view. Model elements are not displayed in the drafting view.

16.5 ViewSection

Section views cut through the model to expose the interior structure. The `Autodesk.Revit.Creation.NewViewSection()` method creates the section view.

Code Region 16-10: NewViewSection()

```
public ViewSection NewViewSection(BoundingBoxXYZ box);
```

The box parameter is the section view crop box. It provides the orientation and bounds which are required for the section view. Usually, another view's crop box is used as the parameter. You can also build a custom `BoundingBoxXYZ` instance to represent the orientation and bounds.

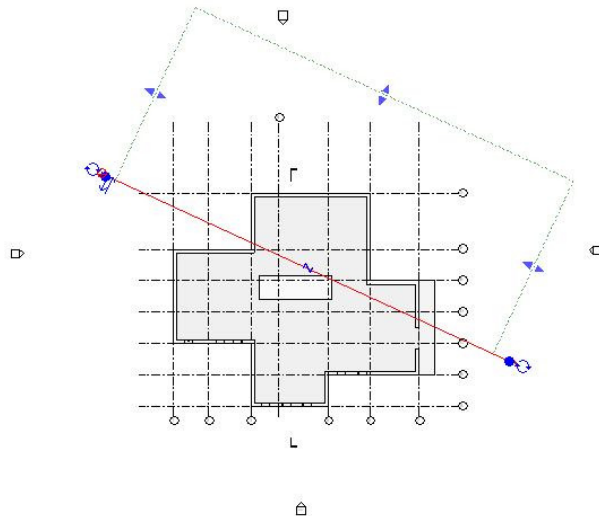


Figure 120: Plan view section

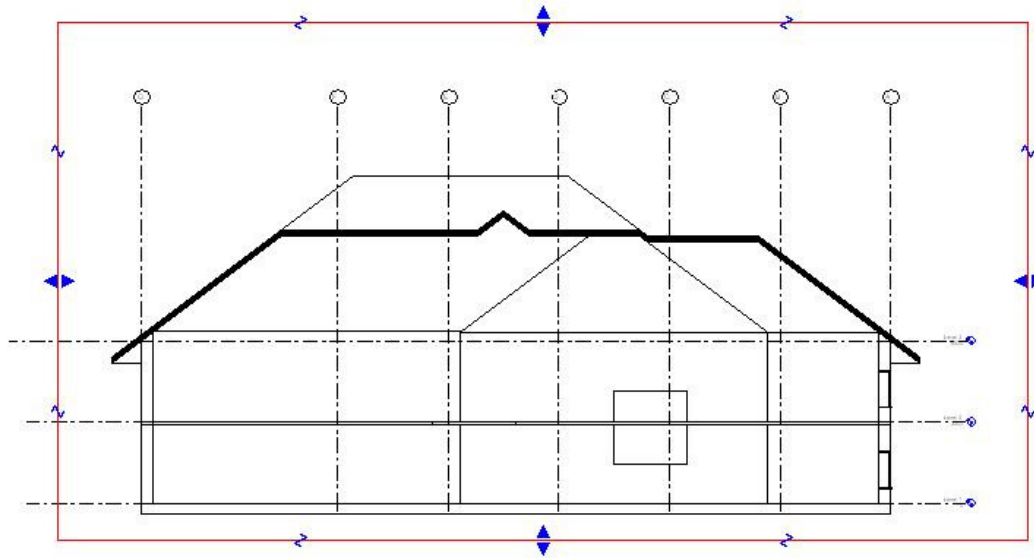


Figure 121: Section view section

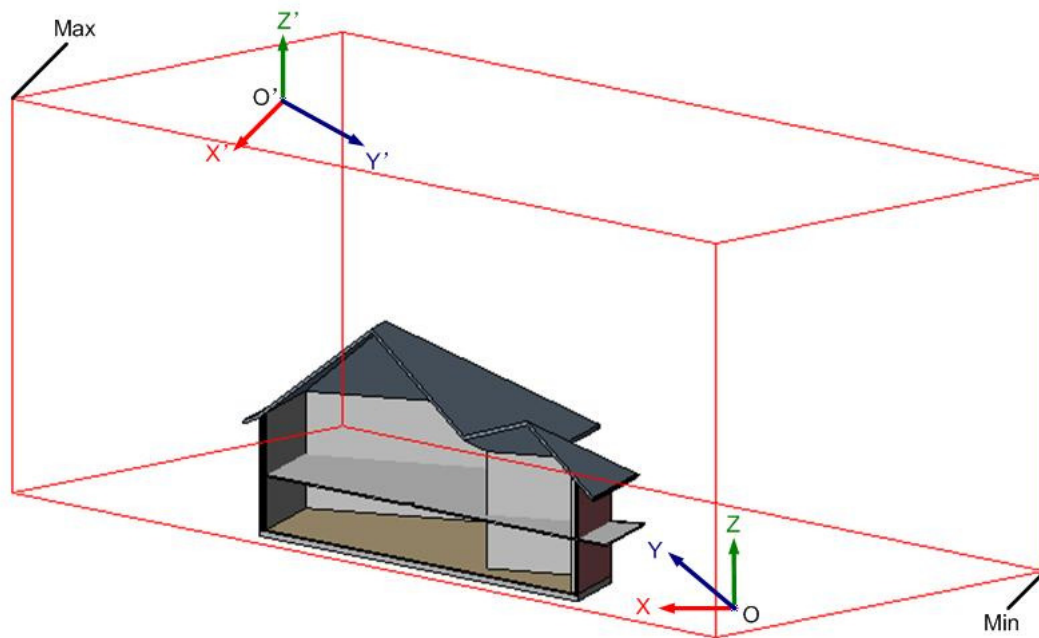


Figure 122: Section view box

For example, apply the box to the section view create method. The coordinate system O-XYZ is the model coordinate system and the box Max and Min points are based on this system. The created section view introduces a new system, O'-X'Y'Z', as the viewing coordinate system.

- Point O' is the section view origin.
- Its coordinates are retrieved using the View.Origin property.
- The section view crop box is calculated using the box.

Note: When you create a section using the command View > New > Section, the created view is located in the Sections (Building Section) node in the Project Browser. The view class type is Elements.View, and the view type is ViewType.Section. However, when you create a section using the Autodesk.Revit.Creation.NewViewSection method(), the created view is in the

Detail Views (Detail) node. The class type is Elements.ViewSection and the view type is ViewType.Detail.

16.6 ViewSheet

A sheet contains views and a title block. When creating a sheet view with the Autodesk.Revit.Creation.NewViewSheet() method, a title block family symbol is a required parameter for the method. The Autodesk.Revit.Document.TitleBlocks property contains all title blocks in the document. Choose one title block to create the sheet.

Code Region 16-11: AddView()

```
public void AddView(View newView, UV location);
```

The newly created sheet has no views. The ViewSheet.AddView() method is used to add views.

- The Geometry.UV location parameter identifies where the added views are located. It points to the added view's center coordinate (measured in inches).
- The coordinates, [0, 0], are relative to the sheet's lower left corner.

Each sheet has a unique sheet number in the complete drawing set. The number is displayed before the sheet name in the Project Browser. It is convenient to use the sheet number in a view title to cross-reference the sheets in your drawing set. You can retrieve or modify the number using the SheetNumber property. The number must be unique; otherwise an exception is thrown when you set the number to a duplicate value.

The following example illustrates how to create and print a sheet view. Begin by finding an available title block in the document and use it to create the sheet view. Next, add the document active view. The active view is placed in the center of the sheet. Finally, print the sheet by calling the View.Print() method.

Code Region 16-12: Creating a sheet view

```
private void CreateSheetView(Autodesk.Revit.Document document, View3D view3D)
{
    // Get an available title block from document
    FamilySymbolSet fsSet = document.TitleBlocks;
    if (fsSet.Size == 0)
    {
        throw new Exception("No title blocks");
    }

    FamilySymbol fs = null;
    foreach (FamilySymbol f in fsSet)
    {
        if (null != f)
        {
            fs = f;
            break;
        }
    }

    // Create a sheet view
```

```

ViewSheet viewSheet = document.Create.NewViewSheet(fs);
if (null == viewSheet)
{
    throw new Exception("Failed to create new ViewSheet.");
}

// Add passed in view onto the center of the sheet
UV location = new UV((viewSheet.Outline.Max.U - viewSheet.Outline.Min.U) / 2,
                    (viewSheet.Outline.Max.V - viewSheet.Outline.Min.V) / 2);

viewSheet.AddView(view3D, location);

// Print the sheet out
if (viewSheet.CanBePrinted)
{
    if (MessageBox.Show("Print the sheet?", "Revit", MessageBoxButtons.YesNo) ==
        DialogResult.Yes)
    {
        viewSheet.Print();
    }
}
}

```

Note: You cannot add a sheet view to another sheet and you cannot add a view to more than one sheet; otherwise an argument exception occurs.

16.6.1 Printer Setup

You may want to change the settings of the printer before printing a sheet. The API exposes the settings for the printer with the PrintManager class, and related Autodesk.Revit classes:

Class	Functionality
Autodesk.Revit.PrintManager	Represents the Print information in Print Dialog (File->Print) within the Revit UI.
Autodesk.Revit.PrintParameters	An object that contains settings used for printing the document.
Autodesk.Revit.PrintSetup	Represents the Print Setup (File->Print Setup...) within the Revit UI.
Autodesk.Revit.PaperSize	An object that represents a Paper Size of Print Setup within the Autodesk Revit project.
Autodesk.Revit.PaperSizeSet	A set that can contain any number of paper size objects.
Autodesk.Revit.PaperSource	An object that represents a Paper Source of Print Setup within the Autodesk Revit project.
Autodesk.Revit.PaperSourceSet	A set that can contain any number of paper source objects.
Autodesk.Revit.ViewSheetSetting	Represents the View/Sheet Set (File->Print)

	within the Revit UI.
Autodesk.Revit.Elements.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.

17 Material

In the Revit Platform API, material data is stored and managed as an Element. The Material element includes Elements.Material and its subclasses:

- MaterialGeneric
- MaterialConcrete
- MaterialWood
- MaterialSteel
- MaterialOther.

Material features are represented by properties, such as FillPattern, Color, Render and so on.

In this chapter, you learn how to access material elements and how to manage the Material objects in the document. The section, [Walkthrough: Get Window Materials](#), provides a walkthrough showing how to get a window material.

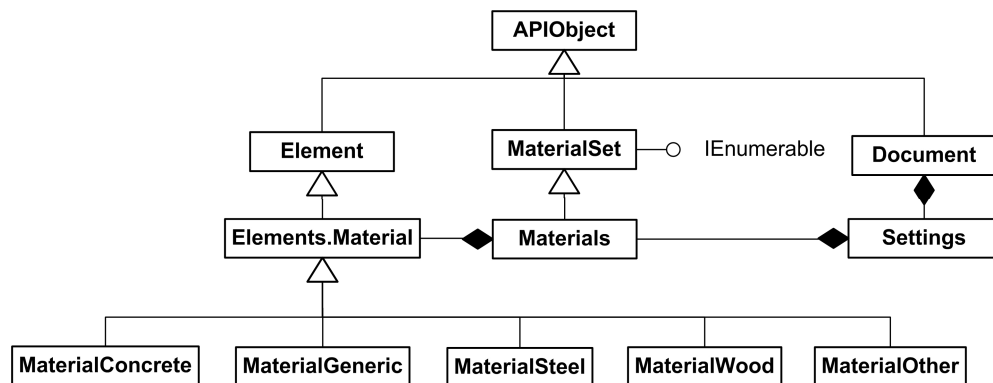


Figure 123: Material-related classes diagram

17.1 General Material Information

Before you begin the walkthrough, read through the following section for a better understanding of the Material class.

17.1.1 Classification

All Elements.Material objects are available in the Settings class Materials property. For more details, refer to the [Management in Document](#) section in this chapter. Material objects are also available in Document, Category, Element, Face, and so on, and are discussed in the pertinent sections in this chapter. Wherever you get a material object, it is represented as the Elements.Material class requiring you to downcast the object to its derived type.

There are two ways to downcast the Elements.Material object to its derived type. Use Runtime Type Information (RTTI) or BuiltInParameter.

17.1.1.1 Use Runtime Type Information (RTTI)

The basic way to downcast the object is to use RTTI, as illustrated below.

Code Region 17-1: Downcasting using RTTI

```

ElementIterator iter = document.Elements;
while (iter.MoveNext())
{

```



```

MaterialSteel materialSteel = iter.Current as MaterialSteel;
if (null != materialSteel)
{
    //use it as a MaterialSteel object
}
}

```

17.1.1.2 Use BuiltInParameter

The second option is to use the BuiltInParameter PHY_MATERIAL_PARAM_TYPE. It is an Elements.Material class integer parameter. The parameter value identifies the Material type, as shown in the following table.

Value	Type in API	Type in Revit
0	MaterialOther	Unassigned
1	MaterialConcrete	Concrete
2	MaterialSteel	Steel
3	MaterialGeneric	Generic
4	MaterialWood	Wood

Table 43: PHY_MATERIAL_PARAM_TYPE Parameter Values

Note: Do not convert the integer parameter to Structural.Enums.Material. Structural.Enums.Material represents a structural FamilyInstance family parameter in the Revit Structure product. The FamilyInstance or FamilySymbol Material property is determined by the Family. It is not related to the Material object. In Revit, when editing a Structural Family (Settings > Family Category and Parameter), note that the Family Parameter, Structural Material Type, corresponds to the Structural.Enums.Material enumerated type.

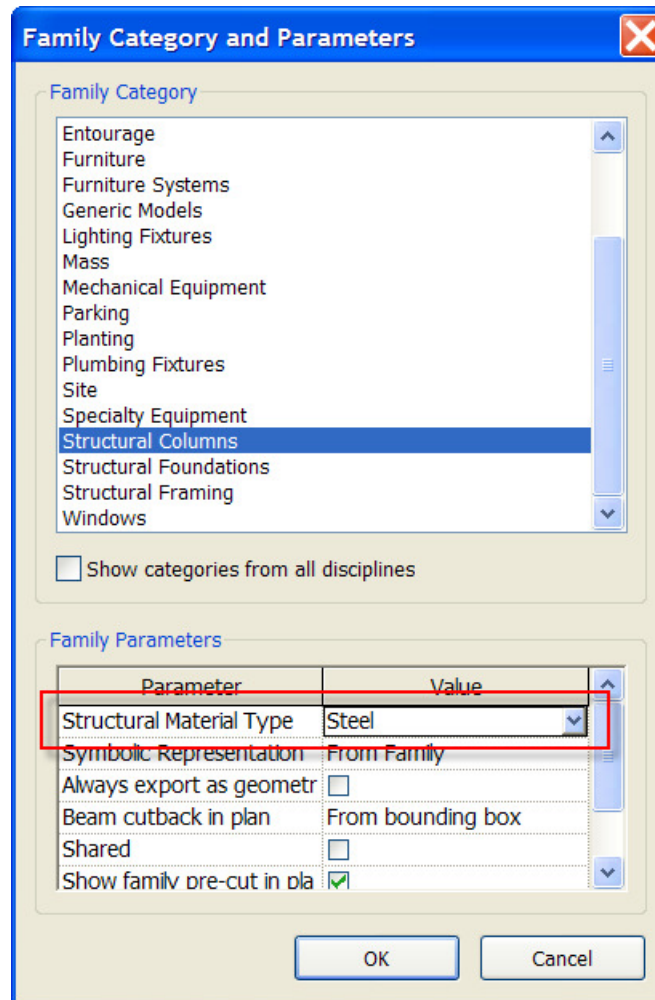


Figure 124: Family parameter and Enum material

Although there is no enumerated type for the Integer you can define one, as illustrated in the following code:

Code Region 17-2: Determining material type

```
private void GetMaterialType(Material material)
{
    Parameter parameter = material.get_Parameter(BuiltInParameter.PHY_MATERIAL_PARAM_TYPE);
    int materialType = parameter.AsInteger();
    switch (materialType)
    {
        case 0: // Other
            MaterialOther matOther = material as MaterialOther;
            if (null != matOther)
            {
                // use MaterialOther object
            }
            break;
        case 1: // Concrete
            MaterialConcrete matConcrete = material as MaterialConcrete;
            if (null != matConcrete)
```

```

        {
            // use MaterialConcrete object
        }
        break;
    case 2:    // Steel
        MaterialSteel matSteel = material as MaterialSteel;
        if (null != matSteel)
        {
            // use MaterialSteel object
        }
        break;
    case 3:    // Generic
        MaterialGeneric matGeneric = material as MaterialGeneric;
        if (null != matGeneric)
        {
            // use MaterialGeneric object
        }
        break;
    case 4:    // Wood
        MaterialWood matWood = material as MaterialWood;
        if (null != matWood)
        {
            // use MaterialWood object
        }
        break;
    }
}

```

There is no way to change the basic Material type. For example, you cannot change a MaterialSteel object to a MaterialWood object.

Note: The API does not provide access to the values of Concrete Type for Concrete material.

17.1.2 Properties

The material object properties identify a specific type of material including color, fill pattern, and more.

17.1.2.1 Properties and Parameter

Elements.Material and its subclasses provide properties that have a counterpart Parameter. For example, the Color property corresponds to the BuiltInParameter MATERIAL_PARAM_COLOR.

In some cases, it is better to use a parameter rather than a property. For example, the AsValueString() method (refer to the [Parameter](#) chapter sections AsValueString() and SetValueString()) is used to get a Parameter value in the unit. The result is converted to the same value as the one you see in Revit.

Code Region 17-3: Getting a material parameter

```

public void GetYoungModulus(MaterialSteel materialSteel)
{
    //get young mod x as a parameter
}

```

```

Parameter parameter =
    materialSteel.get_Parameter(BuiltInParameter.PHY_MATERIAL_PARAM_YOUNG_MOD1);
double dYOUNG_MODX = parameter.AsDouble();
string strYOUNG_MOD = parameter.AsValueString();
string message = "Young's Modulus X from parameter: " + dYOUNG_MODX;

// get young mod x as property
double dYOUNG_MODX2 = materialSteel.YoungModulusX;
message += "\nYoung's Modulus X from property: " + dYOUNG_MODX2;

message += "\nYoung's Modulus X AsValueString; " + strYOUNG_MOD;

MessageBox.Show(message, "Revit");
}

```

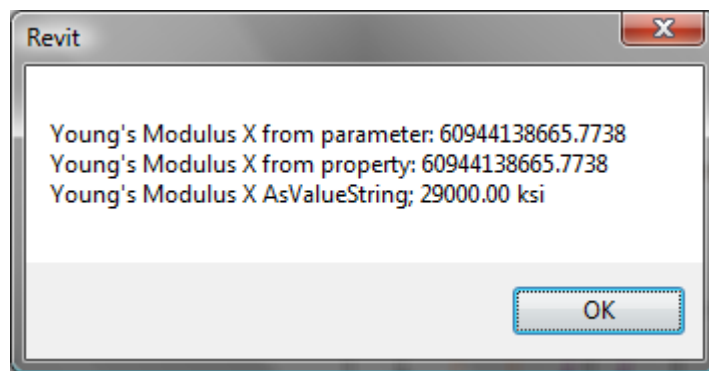


Figure 125: Result from Getting a material parameter code

17.1.2.2 Rendering Information

Collections of rendering data are organized into objects called Assets, which are read-only. You can obtain all available Appearance-related assets from the `Application.Assets` property. An appearance asset can be accessed from a material via the `Material.RenderAppearance` property.

The following figure shows the Render Appearance Library dialog box, which shows how some rendering assets are displayed in the UI.

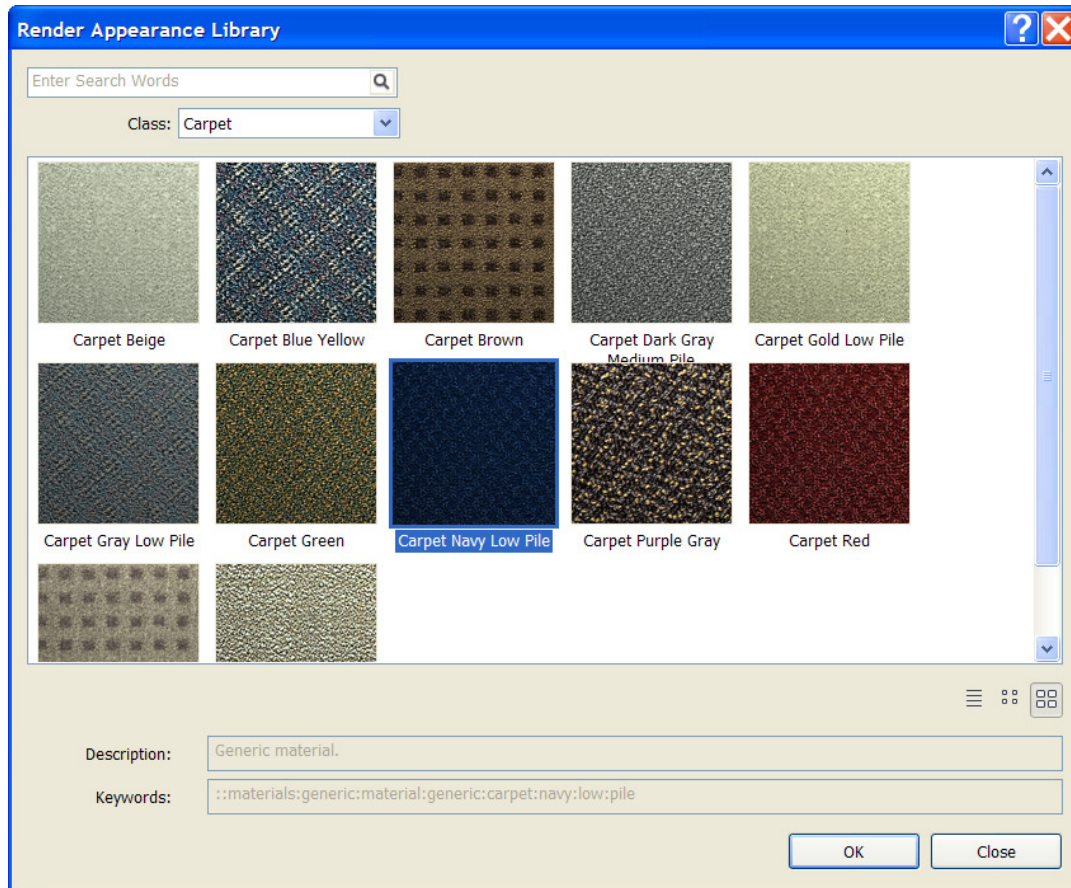


Figure 126: Render Appearance Library

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`.

17.1.2.3 FillPattern

All `FillPatterns` in a document are available in the `Settings` class `FillPatterns` property. Currently the `FillPattern` object only contains the Name and ID information. There are two kinds of `FillPatterns`: Drafting and Model. In the UI, you can only set Drafting fill patterns to `Material.CutPattern`. However, the classification is not exposed in the API. The following example shows how to change the material `FillPattern`.

Code Region 17-4: Setting the fill pattern

```
public void SetFillPattern(Document document, Material material)
{
    foreach (FillPattern fillPattern in document.Settings.FillPatterns)
    {
        // always set successfully
        material.CutPattern = fillPattern;
        material.SurfacePattern = fillPattern;
    }
}
```

17.2 Material Management

The MaterialSet object retrieved from the Settings class manages all Materials in the Document. Every Elements.Material object in the Document is identified by a unique name.

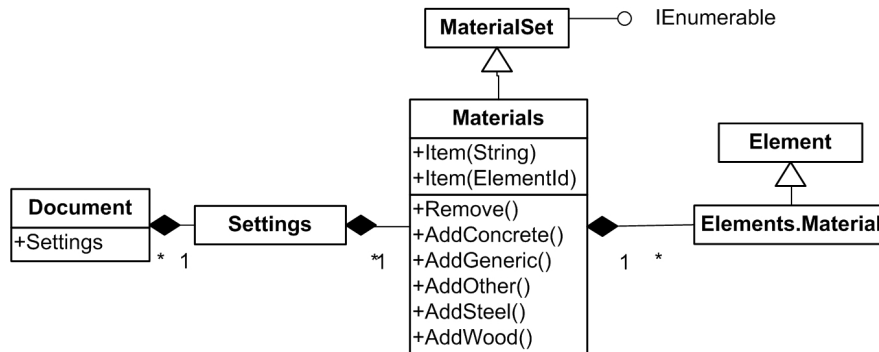


Figure 127: Material management diagram

The following example illustrates how to use the material name to get material and how to get the Document material set.

Code Region 17-5: Getting a material by name

```

Materials materials = document.Settings.Materials;
Material floorMaterial = null;
string floorMaterialName = "Default Floor";
floorMaterial = materials.get_Item(floorMaterialName);
if (null != floorMaterial)
{
    MessageBox.Show("Material found.", "Revit");
}

//or travel all the materials
foreach (Material material in materials)
{
    if (floorMaterialName == material.Name)
    {
        floorMaterial = material;
        break;
    }
}
if (null != floorMaterial)
{
    MessageBox.Show("Material found.", "Revit");
}

```

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).

17.2.1 Creating Materials

There are two ways to create a new Elements.Material object in the API.

- Duplicate an existing Material
- Add a new Material with a specific type.

When using the Duplicate() method, the returned Material object has the same type as the original and it is added to the Materials collection automatically.

Code Region 17-6: Duplicating a material

```
private bool DuplicateMaterial(Material material)
{
    bool duplicated = false;
    //try to duplicate a new instance of Material class using duplicate method
    //make sure the name of new material is unique in MaterialSet
    string newName = "new" + material.Name;
    Material myMaterial = material.Duplicate(newName);
    if (null == myMaterial)
    {
        MessageBox.Show("Failed to duplicate a material!");
    }
    else
    {
        duplicated = true;
    }

    return duplicated;
}
```

Use the Materials class to add a new Material directly. No matter how it is applied, it is necessary to specify a unique name. The unique name is the Elements.Material object key.

Code Region 17-7: Adding a new Material

```
private MaterialConcrete AddConcrete(Document document)
{
    string newName = "myConcreteMaterial";
    MaterialConcrete myConcrete = document.Settings.Materials.AddConcrete(newName);

    if (document.Settings.Materials.Contains(newName) == true)
    {
        MessageBox.Show("Concrete material added successfully!", "Revit");
    }

    return myConcrete;
}
```

17.2.2 Deleting Materials

There are two ways to delete a material.

- Materials.Remove()
- Document.Delete()

The `Materials.Remove()` method is specific to the `Elements.Material` class.

Code Region 17-8: Removing a material

```
private bool CreateOtherMaterial(Autodesk.Revit.Document document)
{
    string oldName = "myOtherMaterial";
    MaterialOther myOther = document.Settings.Materials.AddOther(oldName);
    if (null == myOther)
    {
        return false;
    }

    //try to duplicate a new instance using duplicate method
    //make sure the name of new material is unique in MaterialSet
    string newName = "new" + myOther.Name;
    MaterialOther newOther = myOther.Duplicate(newName) as MaterialOther;
    if (null == newOther)
    {
        MessageBox.Show("Failed to duplicate an other material!");
        return false;
    }
    else
    {
        MessageBox.Show("Duplicated other material.");
    }

    // Remove the original created material
    document.Settings.Materials.Remove(oldName);

    if (document.Settings.Materials.Contains(oldName) == false)
    {
        MessageBox.Show("Material (myOtherMaterial) removed successfully!", "Revit");
    }

    return true;
}
```

`Document.Delete()` is a more generic method. See the [Editing Elements](#) chapter for details.

Note: Though you can delete material using `Document.Delete()`, the `Materials` collection is not updated immediately after it is called. As a result, it is easy to cause Material management inconsistencies in your Add-in application. The best approach is to only use `Materials.Remove()`.

17.3 Element Material

One element can have several elements and components. For example, `FamilyInstance` has `SubComponents` and `Wall` has `CompoundStructure` which contain several `CompoundStructureLayers`. (For more details about `SubComponents` refer to the `Family Instances`

chapter and refer to the [Walls, Floors, Roofs and Openings](#) chapter for more information about CompoundStructure.)

In the Revit Platform API, get an element's materials using the following guidelines:

- If the element contains elements, get the materials separately.
- If the element contains components, get the material for each component from the parameters or in specific way (see [Material](#) section in the [Walls, Floors, Roofs and Openings](#) chapter).
- If the component's material returns null, get the material from the corresponding Element.Category sub Category.

17.3.1 Material in a Parameter

If the Element object has a Parameter where ParameterType is ParameterType.Material, you can get the element material from the Parameter. For example, a structural column FamilySymbol (a FamilyInstance whose Category is BuiltInCategory.OST_StructuralColumns) has the Column Material parameter. Get the Material using the ElementId. The following code example illustrates how to get the structural column Material that has one component.

Code Region 17-9: Getting an element material from a parameter

```
public void GetMaterial(Document document, FamilyInstance familyInstance)
{
    foreach (Parameter parameter in familyInstance.Parameters)
    {
        Definition definition = parameter.Definition;
        // material is stored as element id
        if (parameter.StorageType == StorageType.ElementId)
        {
            if (definition.ParameterGroup == BuiltInParameterGroup.PG_MATERIALS &&
                definition.ParameterType == ParameterType.Material)
            {
                Autodesk.Revit.Elements.Material material = null;
                ElementId materialId = parameter.AsElementId();
                if (-1 == materialId.Value)
                {
                    //Invalid ElementId, assume the material is "By Category"
                    if (null != familyInstance.Category)
                    {
                        material = familyInstance.Category.Material;
                    }
                }
                else
                {
                    material = document.Settings.Materials.get_Item(materialId);
                }

                MessageBox.Show("Element material: " + material.Name, "Revit");

                break;
            }
        }
    }
}
```

```

    }
}
}

```

Note: If the material property is set to By Category in the UI, the ElementId for the material is -1 and cannot be used to retrieve the Material object as shown in the sample code. Try retrieving the Material from Category as described in the next section.

Some material properties contained in other compound parameters are not accessible from the API. As an example, in the following figure, for System Family: Railing, the Rail Structure parameter's StorageType is StorageType.None. As a result, you cannot get material information in this situation.

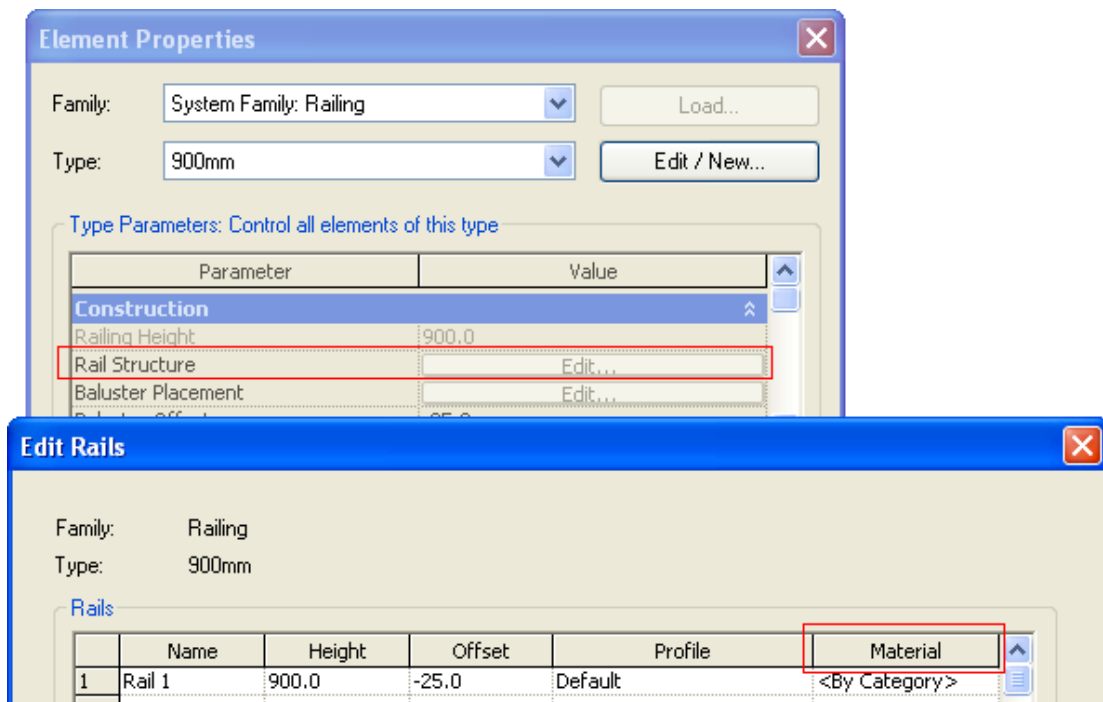


Figure 128: Rail structure property

17.3.2 Material and Category

Only model elements can have material.

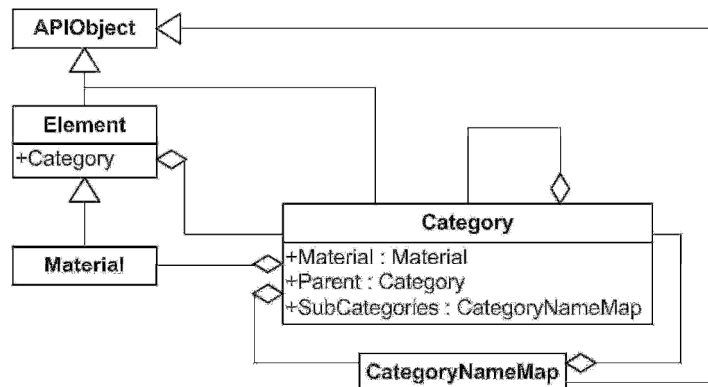


Figure 129: Material and Category diagram

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.

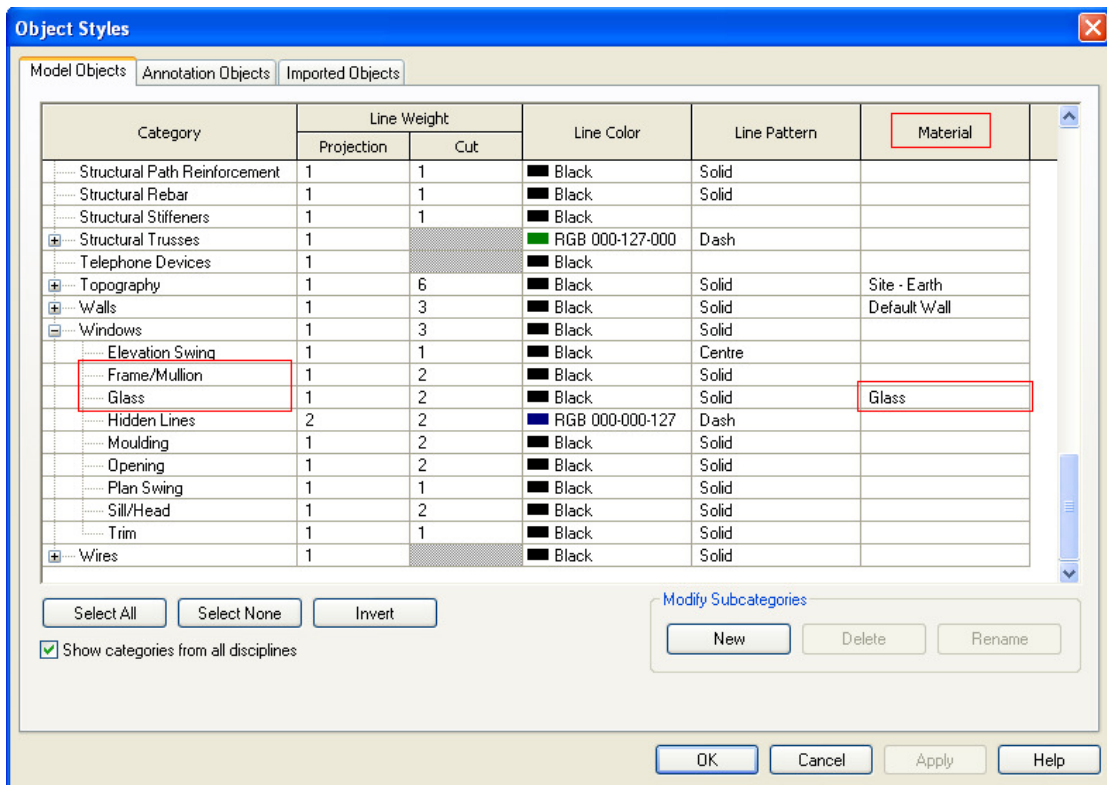


Figure 130: Category material

Only Model elements can have the Material property assigned. Querying Material for a category that corresponds to other than Model elements (e.g. Annotations or Imported) will therefore always result in a null. For more details about the Element and Category classifications, refer to the [Elements Essentials](#) chapter.

If an element has more than one component, some of the Category.Subcategories correspond to the components.

In the previous Object Styles dialog box, the Windows Category and the Frame/Mullion and Glass subcategories are mapped to components in the windows element. In the following picture, it seems the window symbol Glass Pane Material parameter is the only way to get the window pane

material. However, the value is By Category and the corresponding Parameter returns -1 as an invalid ElementId.

In this case, the pane's Material is not null and it depends on the Category OST_WindowsFrameMullionProjection's Material property which is a subcategory of the window's category, OST_Windows. If it returns null as well, the pane's Material is determined by the parent category OST_Windows. For more details, refer to the [Walkthrough: Get Materials of a Window](#) section in this chapter.

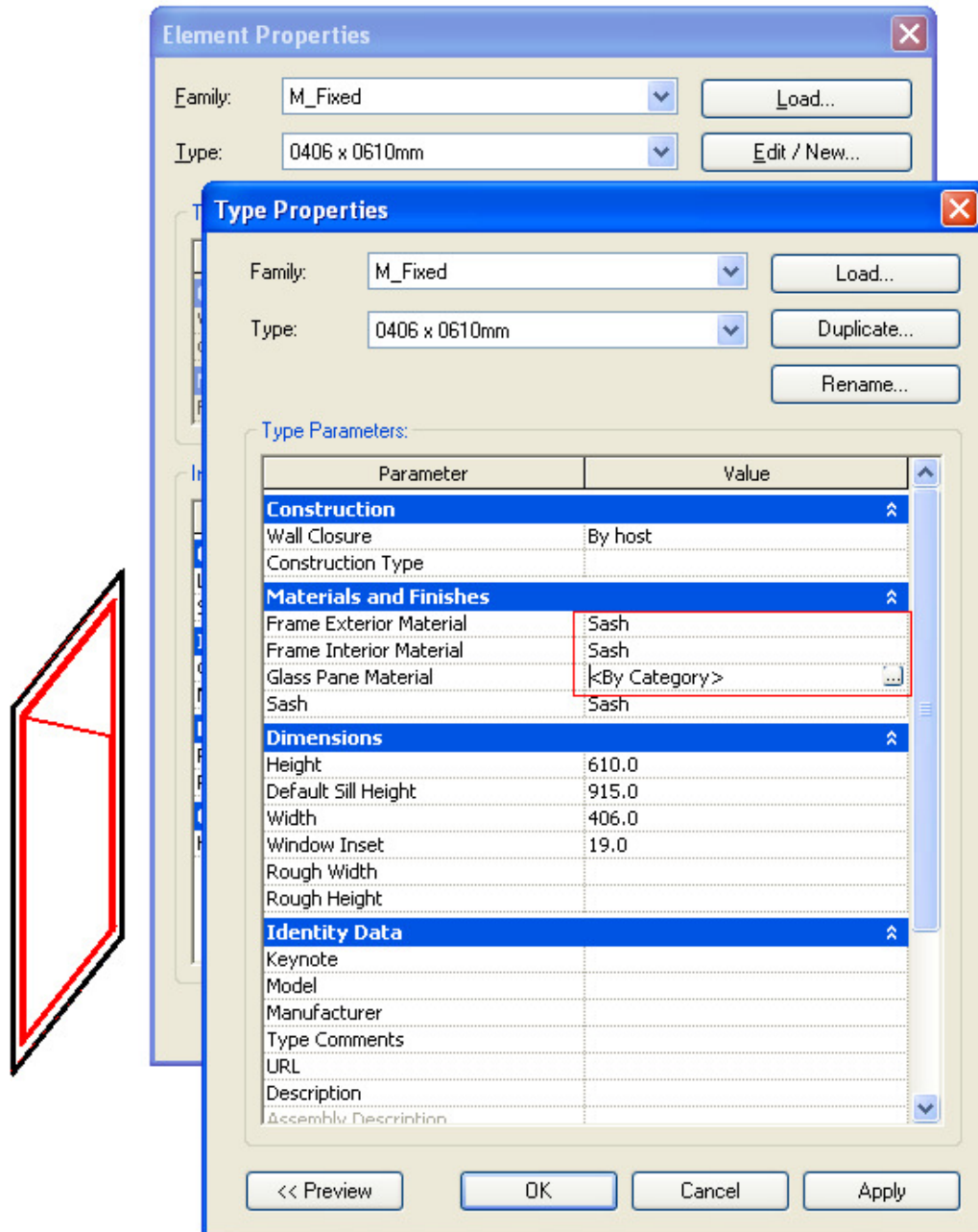


Figure 131: Window material

17.3.3 CompoundStructureLayer Material

You can get the CompoundStructureLayer object from HostObjAttributes. For more details, refer to the [Walls, Floors, Roofs and Openings](#) chapter.

17.3.4 Retrieve Element Materials

The following diagram shows the workflow to retrieve Element Materials:

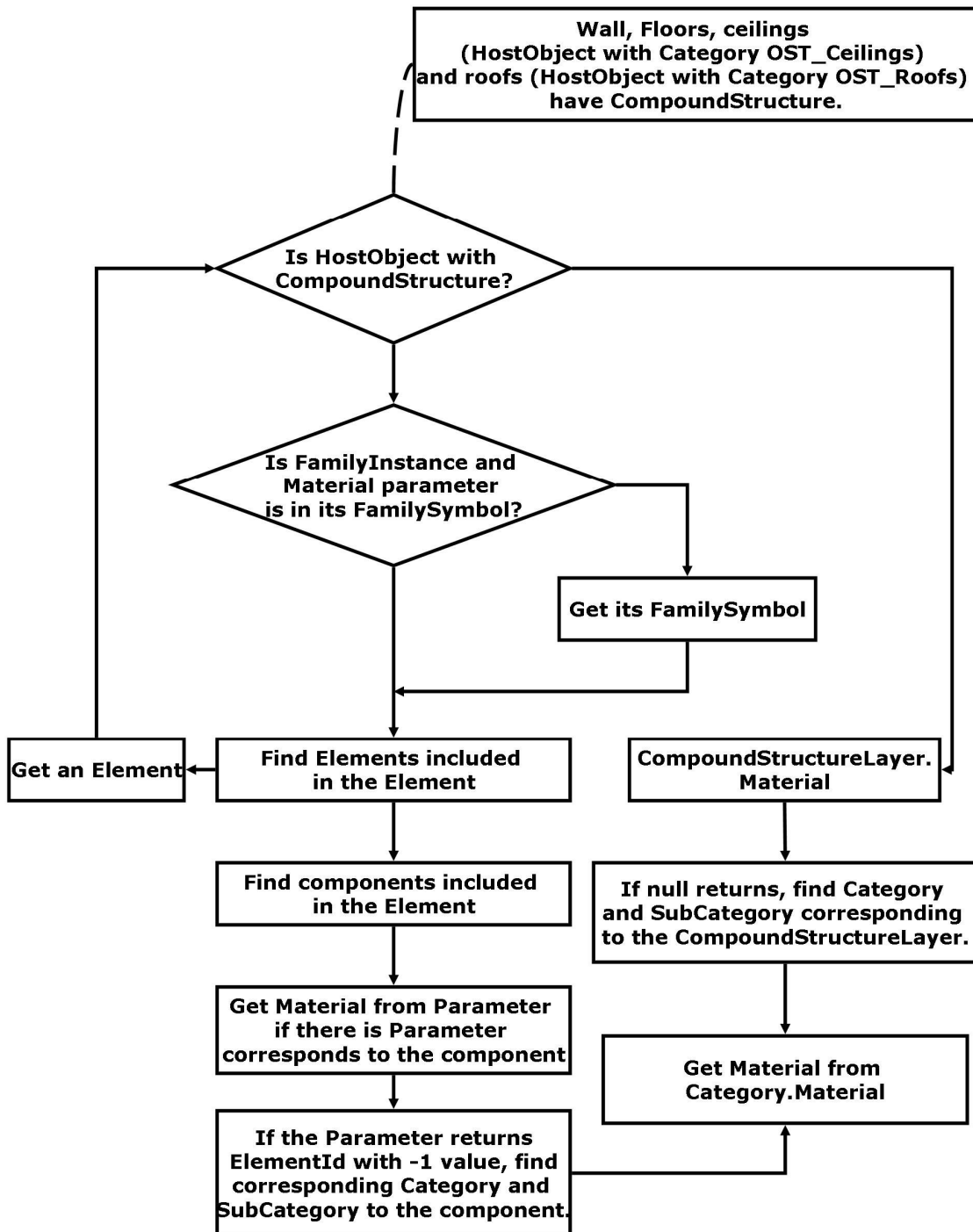


Figure 132: Getting Element Material workflow

This workflow illustrates the following process:

- The workflow shows how to get the Elements.Material object (not Structural.Enums.Material enumerated type) that belongs to the element.
- There are two element classifications when retrieving the Material:

- HostObject with CompoundStructure – Get the Elements.Material object from the CompoundStructureLayer class Material property.
- Others - Get the Material from the Parameters.
- When you get a null Material object or an invalid ElementId with a value of -1, try the Material from the corresponding category. Note that a FamilyInstance and its FamilySymbol usually have the same category.
- The more you know about the Element object, the easier it is to get the material. For example:
 - If you know the Element is a beam, you can get the instance parameter Beam Material
 - If you know the element is a window, you can cast it to a FamilyInstance and get the FamilySymbol.
- After that you can get the Parameters such as Frame Exterior Material or Frame Interior Material to get the Elements.Material object. If you get null try to get the Material object from the FamilySymbol Category.
- Not all Element Materials are available in the API.

17.3.5 Walkthrough: Get Window Materials

The following code illustrates how to get the Window Materials.

Code Region 17-10: Getting window materials walkthrough

```
public void GetMaterial(Document document, FamilyInstance window)
{
    Materials materials = document.Settings.Materials;
    FamilySymbol windowSymbol = window.Symbol;
    Category category = windowSymbol.Category;
    Autodesk.Revit.Elements.Material frameExteriorMaterial = null;
    Autodesk.Revit.Elements.Material frameInteriorMaterial = null;
    Autodesk.Revit.Elements.Material sashMaterial = null;
    // Check the parameters first
    foreach (Parameter parameter in windowSymbol.Parameters)
    {
        switch (parameter.Definition.Name)
        {
            case "Frame Exterior Material":
                frameExteriorMaterial = materials.get_Item(parameter.AsElementId());
                break;
            case "Frame Interior Material":
                frameInteriorMaterial = materials.get_Item(parameter.AsElementId());
                break;
            case "Sash":
                sashMaterial = materials.get_Item(parameter.AsElementId());
                break;
            default:
                break;
        }
    }
    // Try category if the material is set by category
```

```
if (null == frameExteriorMaterial)
    frameExteriorMaterial = category.Material;
if (null == frameInteriorMaterial)
    frameInteriorMaterial = category.Material;
if (null == sashMaterial)
    sashMaterial = category.Material;
// Show the result because the category may have a null Material,
// the Material objects need to be checked.
string materialsInfo = "Frame Exterior Material: " + (null != frameExteriorMaterial ?
frameExteriorMaterial.Name : "null") + "\n";
    materialsInfo += "Frame Interior Material: " + (null != frameInteriorMaterial ?
frameInteriorMaterial.Name : "null") + "\n";
    materialsInfo += "Sash: " + (null != sashMaterial ? sashMaterial.Name : "null") + "\n";
    MessageBox.Show(materialsInfo);
}
```

18 Geometry

The Geometry namespace contains graphic-related types used to describe the graphical representation in the API. The API provides three classes used to describe and store the geometry information data according to their base classes:

- [Geometry Node class](#) – Includes classes derived from the GeometryObject class.
- [Geometry Helper class](#) – Includes classes derived from the APIObject class and value types
- [Collection class](#) – Includes classes derived from the IEnumerable or IEnumerator interface.

In this chapter, you learn how to use various graphic-related types, how to retrieve geometry data from an element, how to transform an element, and more.

18.1 Example: Retrieve Geometry Data from a Wall

This walkthrough illustrates how to get geometry data from a wall. The following information is covered:

- Getting the wall geometry edges.
- Getting the wall geometry faces.

Note: Retrieving the geometry data from Element in this example is limited because Instance is not considered. For example, sweeps included in the wall are not available in the sample code. The goal for this walkthrough is to give you a basic idea of how to retrieve geometry data but not cover all conditions. For more information about retrieving geometry data from Element, refer to [Example: Retrieving Geometry Data from a Beam](#).

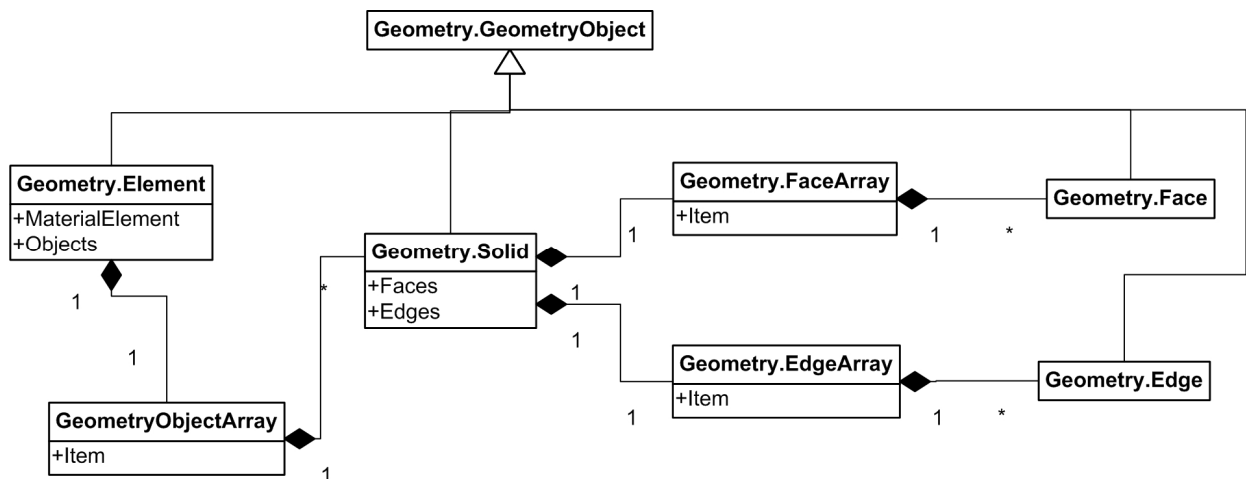


Figure 133: Wall geometry diagram

18.1.1 Create Geometry Options

In order to get the wall's geometry information, you must create a Geometry.Options object which provides detailed customized options. The code is as follows:

Code Region 18-1: Creating Geometry.Options

```

Autodesk.Revit.Geometry.Options geomOption = application.Create.NewGeometryOptions();
if (null != geomOption)
{
    geomOption.ComputeReferences = true;
    geomOption.DetailLevel = Autodesk.Revit.Geometry.Options.DetailLevels.Fine;
}
  
```



```

// Either the DetailLevel or the View can be set, but not both
//geomOption.View = commandData.Application.ActiveDocument.ActiveView;

MessageBox.Show("Geometry Option created successfully.", "Revit");
}

```

Note: For more information, refer to the [Geometry.Options](#) section in this chapter.

18.1.2 Retrieve Faces and Edges

Wall geometry is a solid made up of faces and edges. Complete the following steps to get the faces and edges:

1. Retrieve a `Geometry.Element` instance using the `Wall` class `Geometry` property. This instance contains all geometry objects in the `Object` property, such as a solid, a line, and so on.
2. Iterate the `Object` property to get a geometry solid instance containing all geometry faces and edges in the `Faces` and `Edges` properties.
3. Iterate the `Faces` property to get all geometry faces.
4. Iterate the `Edges` property to get all geometry edges.

The sample code follows:

Code Region 18-2: Retrieving faces and edges

```

private void GetFacesAndEdges(Wall wall)
{
    String faceInfo = "";

    Autodesk.Revit.Geometry.Options opt = new Options();
    Autodesk.Revit.Geometry.Element geomElem = wall.get_Geometry(opt);
    foreach (GeometryObject geomObj in geomElem.Objects)
    {
        Solid geomSolid = geomObj as Solid;
        if (null != geomSolid)
        {
            int faces = 0;
            double totalArea = 0;
            foreach (Face geomFace in geomSolid.Faces)
            {
                faces++;
                faceInfo += "Face " + faces + " area: " + geomFace.Area.ToString() + "\n";
                totalArea += geomFace.Area;
            }
            faceInfo += "Number of faces: " + faces + "\n";
            faceInfo += "Total area: " + totalArea.ToString() + "\n";
            foreach (Edge geomEdge in geomSolid.Edges)
            {
                // get wall's geometry edges
            }
        }
    }
}

```

```

    }
    }
    MessageBox.Show(faceInfo);
}

```

18.2 Geometry Node Class

The Geometry Node class describes the graphical representation in the API. Eight Geometry Node classes are in the API. The classes are:

- Profile - A geometric profile consisting of a loop of curves.
- Face - A 3D solid face. Some derived classes give detailed faces.
- Edge - A 3D solid edge.
- Curve - A parametric curve. Some derived classes provide detailed curves.
- Element - Geometric representation of an element containing all geometry information in its property. This element is not an Element type.
- Mesh - A triangular mesh used to describe the shape of a 3D face.
- Instance - An instance of another element (symbol). Specifically, you can retrieve a symbol's geometry information from the instance referring to it.
- Solid - 3D solid.

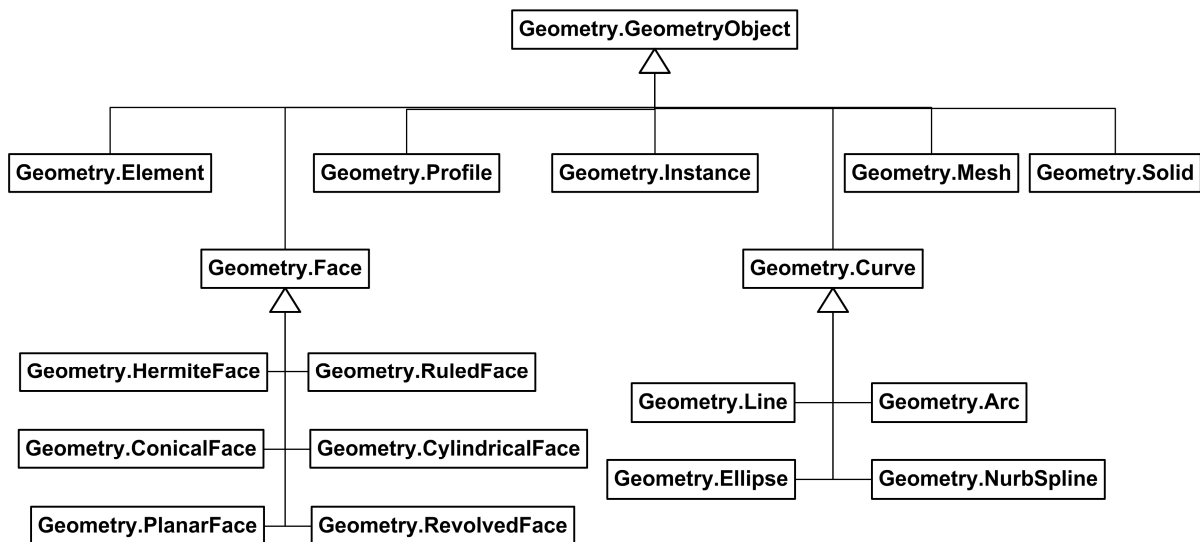


Figure 134: Geometry node hierarchy diagram

18.2.1 Geometry.Instance

The Instance class represents an instance of another element (symbol), specially positioned by this element. It is used to store geometry information for the symbol. The Instance class can pack any geometry information data, including another instance, making it a good way to build a geometry tree using the instance in a complicated geometry representation.

The Instance class stores geometry data in the SymbolGeometry property using a local coordinate system. It also provides a Transform instance to convert the local coordinate system to a world coordinate space. To get the same geometry data as the Revit application from the Instance class,

use the transform property to convert each geometry object retrieved from the SymbolGeometry property.

Generally, three geometry objects can be retrieved from an instance. They are:

- Curve – Curve or its derived class.
- Solid – Contains faces and edges.
- Instance – Another instance which forms this instance.

Users need to transform the retrieved geometry objects using the Transform property. For more details, refer to the [Geometry.Transform](#) section in this chapter.

The following diagram illustrates how to get geometry objects from an instance.

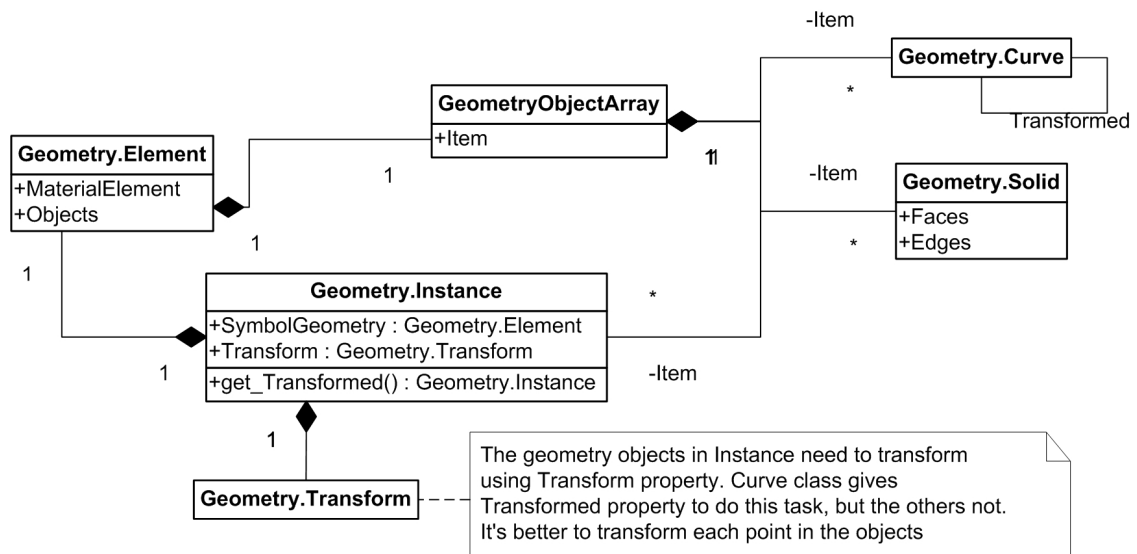


Figure 135: Instance geometry diagram

Note: The diagram identifies the main geometry objects you can retrieve in the Instance class. However, it does not cover all situations. Sometimes other geometry objects can be retrieved based on different elements in the Revit application.

Two samples are presented to explain the main usage.

18.2.1.1 Sample 1

Get curves from an instance, and perform the coordinate transformation.

Code Region 18-3: Getting curves from an instance

```

public void GetAndTransformCurve(Autodesk.Revit.Application app, Autodesk.Revit.Element
element, Options geoOptions)
{
    // Get geometry element of the selected element
    Autodesk.Revit.Geometry.Element geoElement = element.get_Geometry(geoOptions);

    // Get geometry object
    foreach (GeometryObject geoObject in geoElement.Objects)
    {
        // Get the geometry instance which contains the geometry information
        Autodesk.Revit.Geometry.Instance instance =
            geoObject as Autodesk.Revit.Geometry.Instance;

        if (null != instance)
    }
}
  
```

```

    {
        foreach (GeometryObject o in instance.SymbolGeometry.Objects)
        {
            // Get curve
            Curve curve = o as Curve;
            if (curve != null)
            {
                // transform the curve to make it in the instance's coordinate space
                curve = curve.get_Transformed(instance.Transform);
            }
        }
    }
}

```

18.2.1.2 Sample 2

Get the solid information from an instance, and perform the coordinate transformation.

Code Region 18-4: Getting solid information from an instance

```

private void GetAndTransformSolidInfo(Autodesk.Revit.Application application,
Autodesk.Revit.Element element, Options geoOptions)
{
    // Get geometry element of the selected element
    Autodesk.Revit.Geometry.Element geoElement = element.get_Geometry(geoOptions);
    // Get geometry object
    foreach (GeometryObject geoObject in geoElement.Objects)
    {
        // Get the geometry instance which contains the geometry information
        Autodesk.Revit.Geometry.Instance instance =
            geoObject as Autodesk.Revit.Geometry.Instance;

        if (null != instance)
        {
            foreach (GeometryObject instObj in instance.SymbolGeometry.Objects)
            {
                Solid solid = instObj as Solid;
                if (null == solid || 0 == solid.Faces.Size || 0 == solid.Edges.Size)
                {
                    continue;
                }

                Transform instTransform = instance.Transform;
                // Get the faces and edges from solid, and transform the formed points
                foreach (Face face in solid.Faces)
                {
                    Mesh mesh = face.Triangulate();
                    foreach (XYZ ii in mesh.Vertices)
                    {
                        XYZ point = ii;

```

```

        XYZ transformedPoint = instTransform.OfPoint(point);
    }
}
foreach (Edge edge in solid.Edges)
{
    foreach (XYZ ii in edge.Tessellate())
    {
        XYZ point = ii;
        XYZ transformedPoint = instTransform.OfPoint(point);
    }
}
}
}
}

```

Note: You find similar situations when you get geometry information for doors, windows, and other elements. For more details about the retrieved geometry, refer to [Example: Retrieving Geometry Data from a Beam](#) in this chapter.

18.2.2 Geometry.Mesh

Geometry mesh is a triangular mesh used to describe the shape of a 3D face. The face object can be a curved surface which can be described by its endpoints just like a polygon. As a result, the mesh object provides a way to split the face into many small triangles where the endpoints locate the split face.

The following code sample illustrates how to get the geometry of a mass. The mass geometry faces are described by the mesh.

Code Region 18-5: Drawing the geometry of a mass

```

private void TriangulateFace(Face face)
{
    // Get mesh
    Mesh mesh = face.Triangulate();
    for (int i = 0; i < mesh.NumTriangles; i++)
    {
        MeshTriangle triangle = mesh.get_Triangle(i);
        XYZ vertex1 = triangle.get_Vertex(0);
        XYZ vertex2 = triangle.get_Vertex(1);
        XYZ vertex3 = triangle.get_Vertex(2);
    }
}

```

The following pictures illustrate how a mass looks in the Revit application (left picture) and the same mass drawn with the API (right picture).

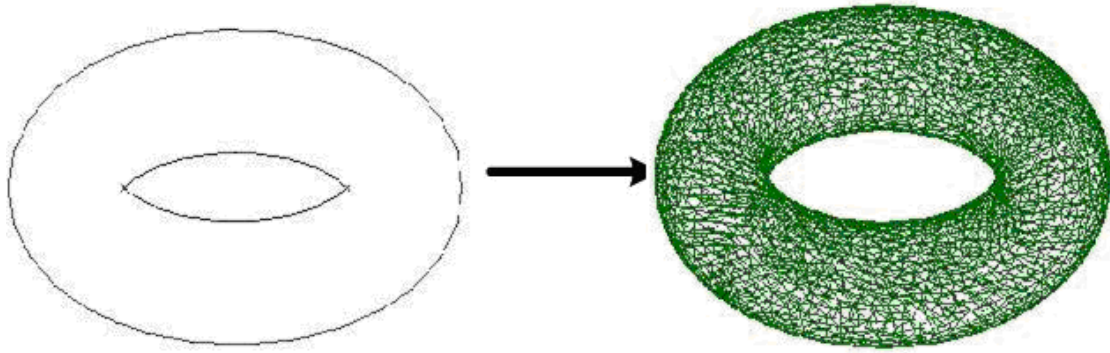


Figure 136: Draw a mass using the API

Note: The approximation tolerance used for display purposes is defined internally by Revit including distance tolerance and angle tolerance.

18.2.3 Solid

The Solid class defines a 3D solid from which users can get faces and edges. The Solid class defines a geometry solid, such as a cube or a cylinder. You can get the following using the solid's properties:

- Faces
- Edges
- Surface area
- Volume

Note: Sometimes the API can have unused solids containing zero edges and faces. Check the Edges and Faces properties before completing further work.

Code Region 18-6: Getting a solid from a GeometryObject

```
// instObj is a GeometryObject
Solid solid = instObj as Solid;
if (null == solid || 0 == solid.Faces.Size || 0 == solid.Edges.Size)
{
    continue;
}
```

18.3 Geometry Helper Class

Several Geometry Helper classes are in the API. The Helper classes are used to describe geometry information for certain elements, such as defining a CropBox for a view using the BoundingBoxXYZ class.

- BoundingBoxXYZ - A 3D rectangular box used in cases such as defining a 3D view section area.
- Transform - Transforming the affine 3D space.
- Reference - A stable reference to a geometric object in a Revit model, which is used when creating elements like dimensions.
- Plane - A flat surface in geometry.
- Options - User preferences for parsing geometry.

- XYZ - Object representing coordinates in 3D space.
- UV - Object representing coordinates in 2D space.
- BoundingBoxUV - A 2D rectangle parallel to the coordinate axes.

The following diagram displays the properties:

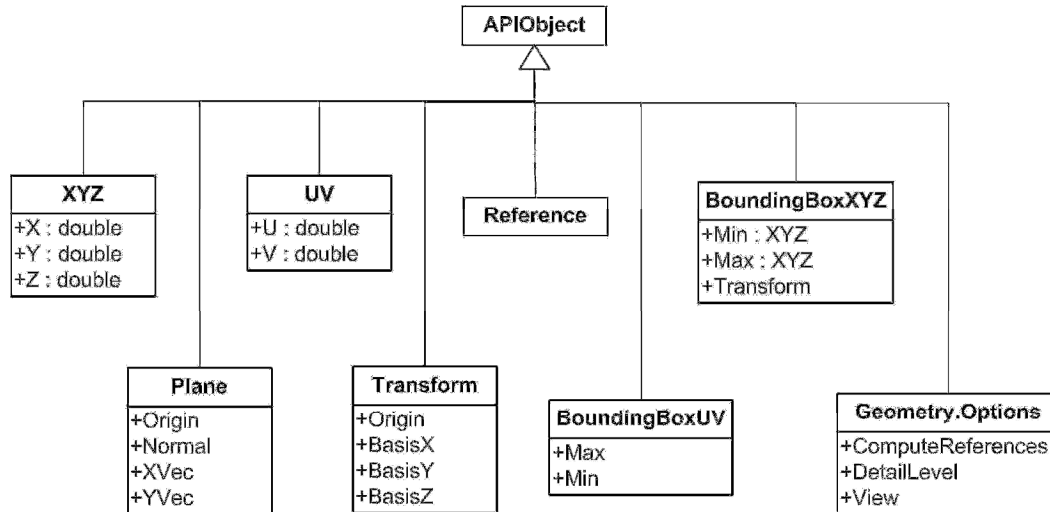


Figure 137: Geometry helper hierarchy diagram

18.3.1 Geometry.Transform

Transforms are limited to 3x4 transformations (Matrix) in the Revit application, transforming an object's place in the model space relative to the rest of the model space and other objects. The transforms are built from the position and orientation in the model space. Three direction Vectors (BasisX, BasisY and BasisZ properties) and Origin point provide all of the transform information. The matrix formed by the four values is as follows:

$$\begin{pmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} & \text{Origin.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} & \text{Origin.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} & \text{Origin.Z} \end{pmatrix}$$

Applying the transformation to the point is as follows:

$$\begin{pmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} & \text{Origin.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} & \text{Origin.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} & \text{Origin.Z} \end{pmatrix} \times \begin{pmatrix} \text{XYZ.X} \\ \text{XYZ.Y} \\ \text{XYZ.Z} \\ 1 \end{pmatrix}$$

The Transform OfPoint method implements the previous function. The following code is a sample of the same transformation process.

Code Region 18-7: Transformation example

```

public static XYZ TransformPoint(XYZ point, Transform transform)
{
    double x = point.X;
    double y = point.Y;

```

```

double z = point.Z;

//transform basis of the old coordinate system in the new coordinate // system
XYZ b0 = transform.getBasis(0);
XYZ b1 = transform.getBasis(1);
XYZ b2 = transform.getBasis(2);
XYZ origin = transform.Origin;

//transform the origin of the old coordinate system in the new
//coordinate system
double xTemp = x * b0.X + y * b1.X + z * b2.X + origin.X;
double yTemp = x * b0.Y + y * b1.Y + z * b2.Y + origin.Y;
double zTemp = x * b0.Z + y * b1.Z + z * b2.Z + origin.Z;

return new XYZ(xTemp, yTemp, zTemp);
}

```

The Geometry.Transform class properties and methods are identified in the following sections.

18.3.1.1 Identity

Transform the Identity.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

18.3.1.2 Reflection

Reflect a specified plane.

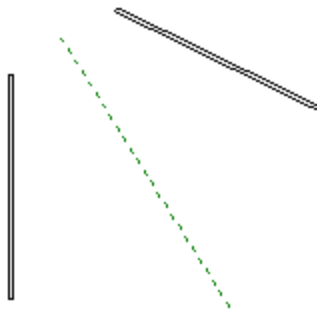


Figure 138: Wall Reflection relationship

As the previous picture shows, one wall is mirrored by a reference plane. The Reflection property needs the geometry plane information for the reference plane.

Code Region 18-8: Using the Reflection property

```

private Transform Reflect(ReferencePlane refPlane)
{
    Transform mirTrans = Transform.get_Reflection(refPlane.Plane);
}

```



```

return mirTrans;
}

```

18.3.1.3 Rotation

Rotate by a specified angle around a specified axis and point.

18.3.1.4 Translation

Translate by a specified vector. Given a vector XYZ data, a transformation is created as follow:

$$(x \ y \ z) \rightarrow \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \end{pmatrix}$$

18.3.1.5 Determinant

Transformation determinant.

$$\begin{vmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} \end{vmatrix}$$

18.3.1.6 HasReflection

This is a Boolean value that indicates whether the transformation produces a reflection.

18.3.1.7 Scale

A value that represents the transformation scale.

18.3.1.8 Inverse

An inverse transformation. Transformation matrix A is invertible if a transformation matrix B exists such that $A*B = B*A = I$ (identity).

18.3.1.9 IsIdentity

Boolean value that indicates whether this transformation is an identity.

18.3.1.10 IsTranslation

Boolean value that indicates whether this transformation is a translation.

Geometry.Transform provides methods to perform basal matrix operations.

18.3.1.11 Multiply

Multiplies a transformation by a specified transformation and returns the result.

Operator* - Multiplies two specified transforms.

18.3.1.12 ScaleBasis

Scales the basis vectors and returns the result.

18.3.1.13 ScaleBasisAndOrigin

Scales the basis vectors and the transformation origin returns the result.

18.3.1.14 OfPoint

Applies the transformation to the point. The Origin property is used.

18.3.1.15 OfVector

Applies the transform to the vector. The Origin property is not used.

18.3.1.16 AlmostEqual

Compares two transformations. AlmostEqual is consistent with the computation mechanism and accuracy in the Revit core code. Additionally, Equal and the == operator are not implemented in the Transform class.

The API provides several shortcuts to complete geometry transformation. The Transformed property in several geometry classes is used to do the work, as shown in the following table.

Class Name	Function Description
Curve.get_Transformed(Transform transform)	Applies the specified transformation to a curve
Instance.get_Transformed(Transform transform)	Transforms the instance.
Profile.get_Transformed(Transform transform)	Transforms the profile and returns the result.
Mesh.get_Transformed(Transform transform)	Transforms the mesh and returns the result.

Table 44: Transformed Methods

Note: The transformed method clones itself then returns the transformed cloned result.

18.3.2 Geometry.Reference

The Reference class does not contain properties or methods. However, it is very useful in element creation.

- Dimension creation requires references.
- The reference identifies a path within a geometric representation tree in a flexible manner.
- The tree is used to view specific geometric representation creation.

The API exposes four types of references based on different Pick pointer types. They are retrieved from the API in different ways:

1. For Point – Curve.EndPointReference property
2. For Curve (Line, Arc, and etc.) - Curve.Reference property
3. For Face – Face.Reference property
4. For Cut Edge – Edge.Reference property

Different reference types cannot be used arbitrarily. For example:

- The NewLineBoundaryConditions() method requires a reference for Line
- The NewAreaBoundaryConditions() method requires a reference for Face
- The NewPointBoundaryConditions() method requires a reference for Point.

18.3.3 Geometry.Options

The Geometry.Options object defines a user preference for parsing geometry. It provides the following customized options:

- ComputeReferences – Indicates whether to compute the geometry reference when retrieving geometry information.
- View - Gets geometry information from a special view.

- DetailLevel – Indicates the preferred detail level.

18.3.3.1 ComputeReferences

If you set this property to false, the API does not compute a geometry reference. All Reference properties retrieved from the geometry tree return nothing. For more details about references, refer to the Reference section.

18.3.3.2 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).

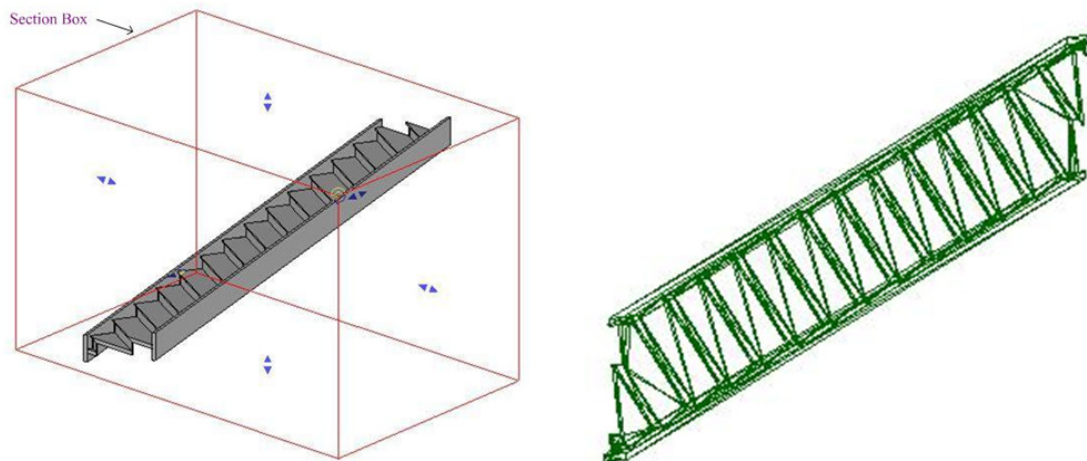


Figure 139: Different section boxes display different geometry

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.

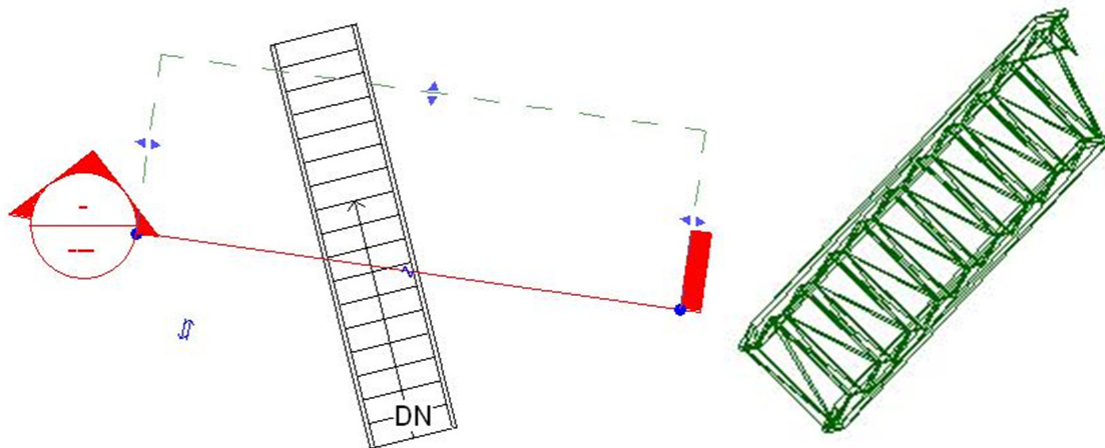


Figure 140: Retrieve Geometry section view

18.3.3.3 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 141: 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:

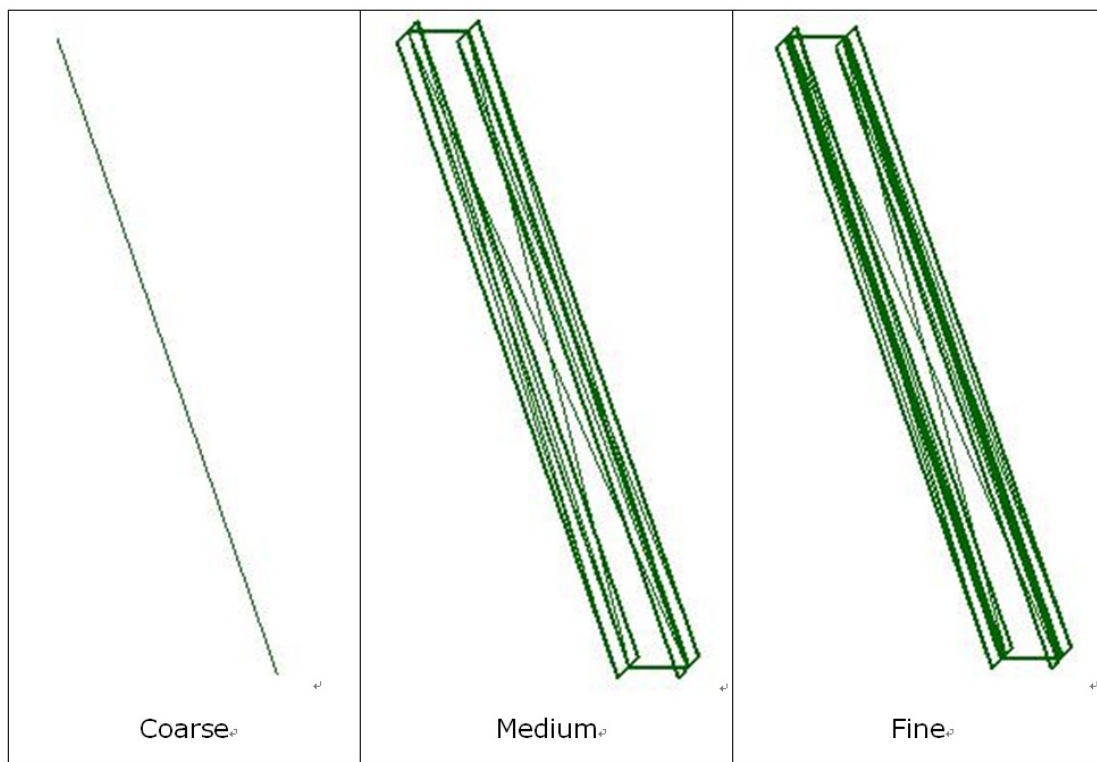


Figure 142: Detail geometry for a beam

18.3.4 Geometry.BoundingBoxXYZ

`BoundingBoxXYZ` is a reference class derived from the `APIObject` class. It 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`.

18.3.4.1 Define the View Boundaries

`BoundingBoxXYZ` can be used to define the view boundaries through `View.CropBox` property. The following pictures use a section view to show how `BoundingBoxXYZ` is used in the Revit application.

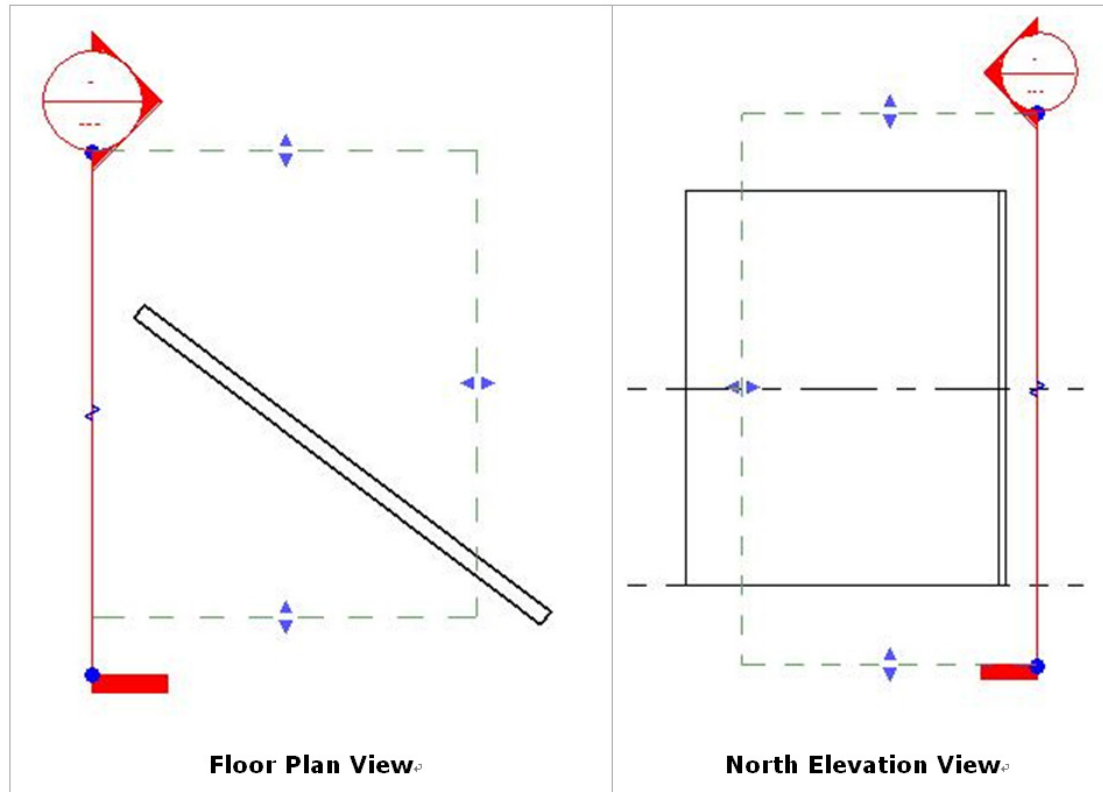


Figure 143: BoundingBoxXYZ in section view

The dash lines in the previous pictures show the section view boundary exposed as the CropBox property (a BoundingBoxXYZ instance).

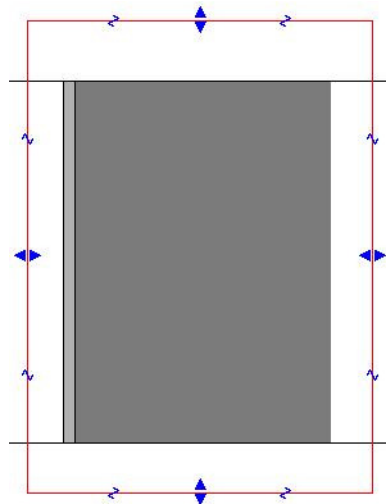


Figure 144: Created section view

The previous picture displays the corresponding section view. The wall outside the view boundary is not displayed.

18.3.4.2 Define a Section Box

BoundingBoxXYZ is also used to define a section box for a 3D view retrieved from the View3D.SectionBox property. Select the Section Box property in the Properties Dialog box. The section box is shown as follows:

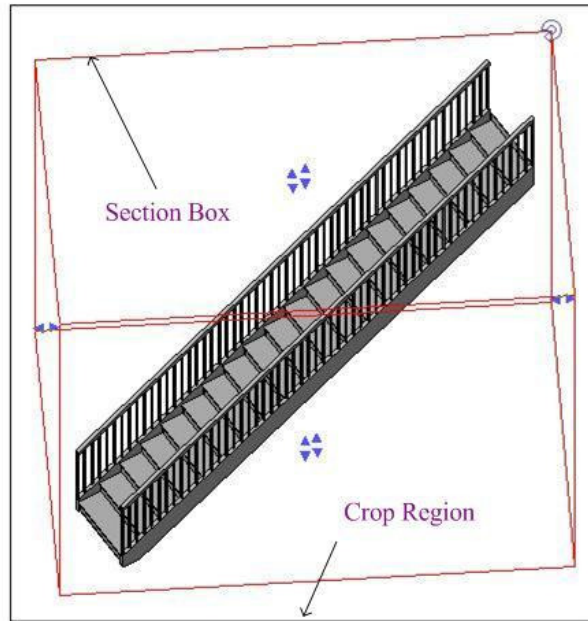


Figure 145: 3D view section box

18.3.4.3 Other Uses

- Defines a box around an element's geometry. (Element.BoundingBox Property). The BoundingBoxXYZ instance retrieved in this way is parallel to the coordinate axes.
- Used in the NewViewSection() method in the Creation.Document class.

The following table identifies the main uses for this class.

Property Name	Usage
Max/Min	Maximum/Minimum coordinates. These two properties define a 3D box parallel to any coordinate axis. The Transform property provides a transform matrix that can transform the box to the appropriate position.
Transform	Transform from the box coordinate space to the model space.
Enabled	Indicates whether the bounding box is turned on.

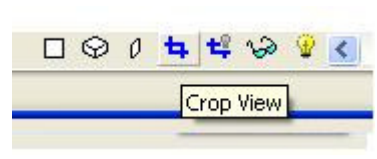
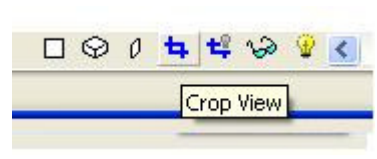
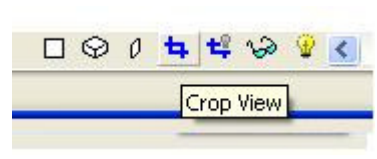
Property Name	Usage				
MaxEnabled/ MinEnabled	Defines whether the maximum/minimum bound is active for a given dimension. - If the Enable property is false, these two properties should also return false. <table border="1"> <tr> <td></td><td>If the crop view is turned on, both MaxEnabled property and MinEnabled property return true.</td></tr> <tr> <td></td><td>If the crop view is turned off, both MaxEnabled property and MinEnabled property return false.</td></tr> </table> - This property indicates whether the view's crop box face can be used to clip the element's view. - If BoundingBoxXYZ is retrieved from the View3D.SectionBox property, the return value depends on whether the Section Box property is selected in the 3D view Properties dialog box. If so, all Enabled properties return true. - If BoundingBoxXYZ is retrieved from the Element.BoundingBox property, all the Enabled properties are true.		If the crop view is turned on, both MaxEnabled property and MinEnabled property return true.		If the crop view is turned off, both MaxEnabled property and MinEnabled property return false.
	If the crop view is turned on, both MaxEnabled property and MinEnabled property return true.				
	If the crop view is turned off, both MaxEnabled property and MinEnabled property return false.				
Bounds	Wrapper for the Max/Min properties.				
BoundEnabled	Wrapper for the MaxEnabled/MinEnabled properties.				

Table 45: BoundingBoxXYZ properties

The following code sample illustrates how to rotate BoundingBoxXYZ to modify the 3D view section box.

Code Region 18-9: Rotating BoundingBoxXYZ

```
private void RotateBoundingBox(View3D view3d)
{
    BoundingBoxXYZ box = view3d.SectionBox;
    if (false == box.Enabled)
    {
        MessageBox.Show("The section box for View3D isn't Enable.");
        return;
    }
    // Create a rotation transform,
    XYZ origin = new XYZ(0, 0, 0);
    XYZ axis = new XYZ(0, 0, 1);
    Transform rotate = Transform.get_Rotation(origin, axis, 2);
    // Transform the View3D's SectionBox with the rotation transform
    box.Transform = box.Transform.Multiply(rotate);
    view3d.SectionBox = box;
}
```

18.3.5 Geometry.BoundingBoxUV

BoundingBoxUV is a value class that defines a 2D rectangle parallel to the coordinate axes. It supports the Min and Max data members. Together they define the BoundingBoxUV's boundary. BoundingBoxUV is retrieved from the View.Outline property which is the boundary view in the paper space view.

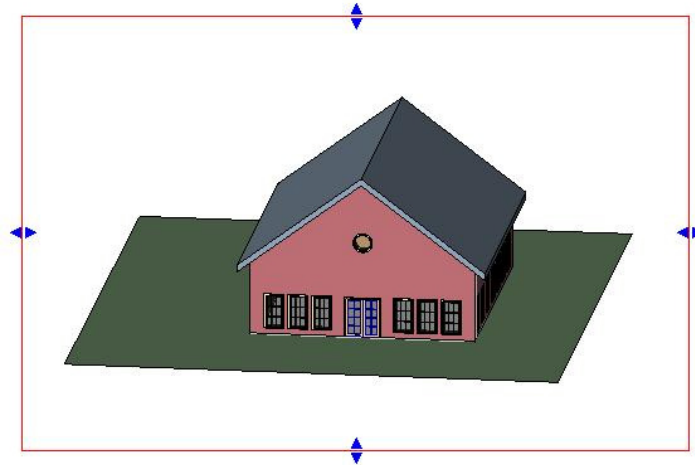


Figure 146: View outline

Two points define a BoundingBoxUV.

- Min point - The bottom-left endpoint.
- Max point - The upper-right endpoint.

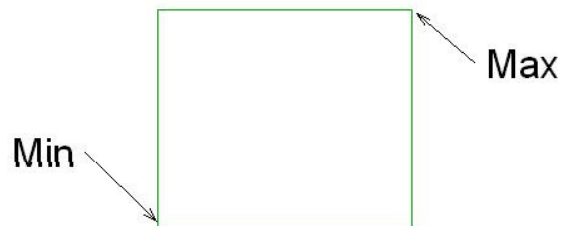


Figure 147: BoundingBoxUV Max and Min

Note: BoundingBoxUV cannot present a gradient rectangle as the following picture shows.

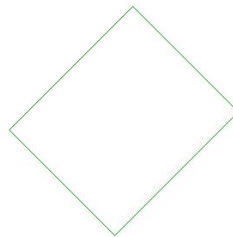


Figure 148: Gradient rectangle

18.4 Collection Classes

The API provides the following collection classes based on the items they contain:

Class/Type	Corresponding Collection Classes	Corresponding Iterators
Curve	CurveArray	CurveArrayIterator

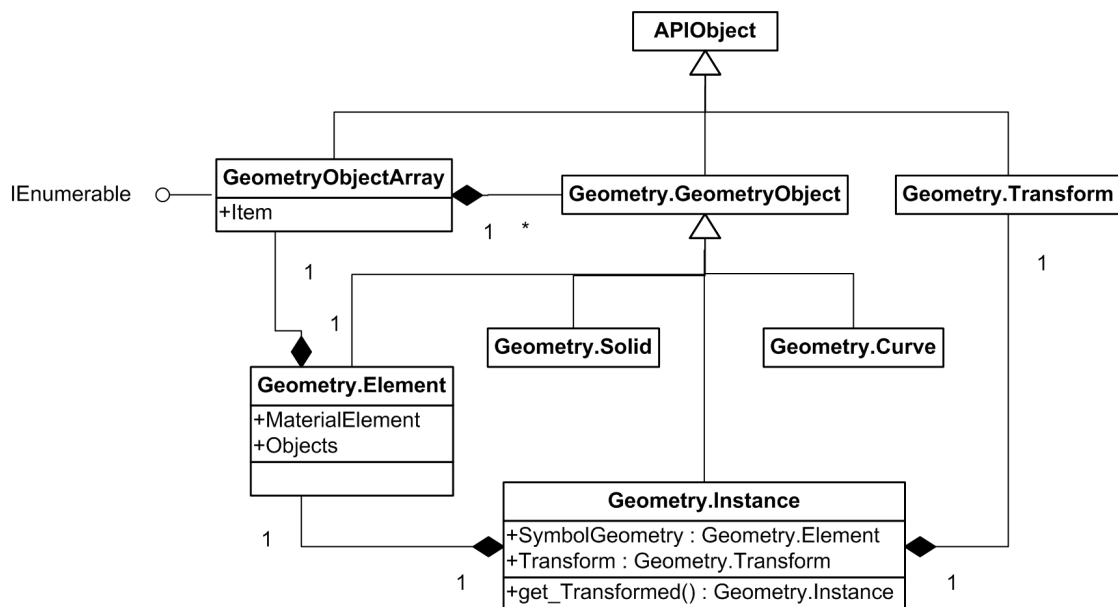
Class/Type	Corresponding Collection Classes	Corresponding Iterators
Edge	EdgeArray, EdgeArrayArray	EdgeArrayIterator, EdgeArrayArrayIterator
Face	FaceArray	FaceArrayIterator
GeometryObject	GeometryObjectArray	GeometryObjectArrayIterator
Instance	InstanceArray	InstanceArrayIterator
Mesh	MeshArray	MeshArrayIterator
Reference	ReferenceArray	ReferenceArrayIterator
Solid	SolidArray	SolidArrayIterator
UV	UVArray	UVArrayIterator
XYZ	XYZArray	XYZArrayIterator
Double value	DoubleArray	DoubleArrayIterator

Table 46: Geometry Collection Classes

All of these classes use very similar methods and properties to do similar work. For more details, refer to the [Collection](#) chapter.

18.5 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.

**Figure 149: Beam geometry diagram**

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 18-10: Getting solids and curves from a beam

```

public void GetCurvesFromABeam(Autodesk.Revit.Elements.FamilyInstance beam,
                                Autodesk.Revit.Geometry.Options options)
{

```

```

Autodesk.Revit.Geometry.Element geomElem = beam.get_Geometry(options);

Autodesk.Revit.Geometry.CurveArray curves = new CurveArray();
Autodesk.Revit.Geometry.SolidArray solids = new SolidArray();

//Find all solids and insert them into solid array
AddCurvesAndSolids(geomElem, ref curves, ref solids);
}

private void AddCurvesAndSolids(Autodesk.Revit.Geometry.Element geomElem,
                                ref Autodesk.Revit.Geometry.CurveArray curves,
                                ref Autodesk.Revit.Geometry.SolidArray solids)
{
    foreach (Autodesk.Revit.Geometry.GeometryObject geomObj in geomElem.Objects)
    {
        Autodesk.Revit.Geometry.Curve curve = geomObj as Autodesk.Revit.Geometry.Curve;
        if (null != curve)
        {
            curves.Append(curve);
            continue;
        }
        Autodesk.Revit.Geometry.Solid solid = geomObj as Autodesk.Revit.Geometry.Solid;
        if (null != solid)
        {
            solids.Append(solid);
            continue;
        }
        //If this GeometryObject is Instance, call AddCurvesAndSolids
        Autodesk.Revit.Geometry.Instance geomInst =
            geomObj as Autodesk.Revit.Geometry.Instance;

        if (null != geomInst)
        {
            Autodesk.Revit.Geometry.Element transformedGeomElem
                = geomInst.GetInstanceGeometry(geomInst.Transform);
            AddCurvesAndSolids(transformedGeomElem, ref curves, ref solids);
        }
    }
}
}

```

Note: For more information about how to retrieve the Geometry.Options type object, refer to the [Geometry.Options](#) section in this chapter.

19 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.

The following diagram displays project location and associated classes in the Site namespace.

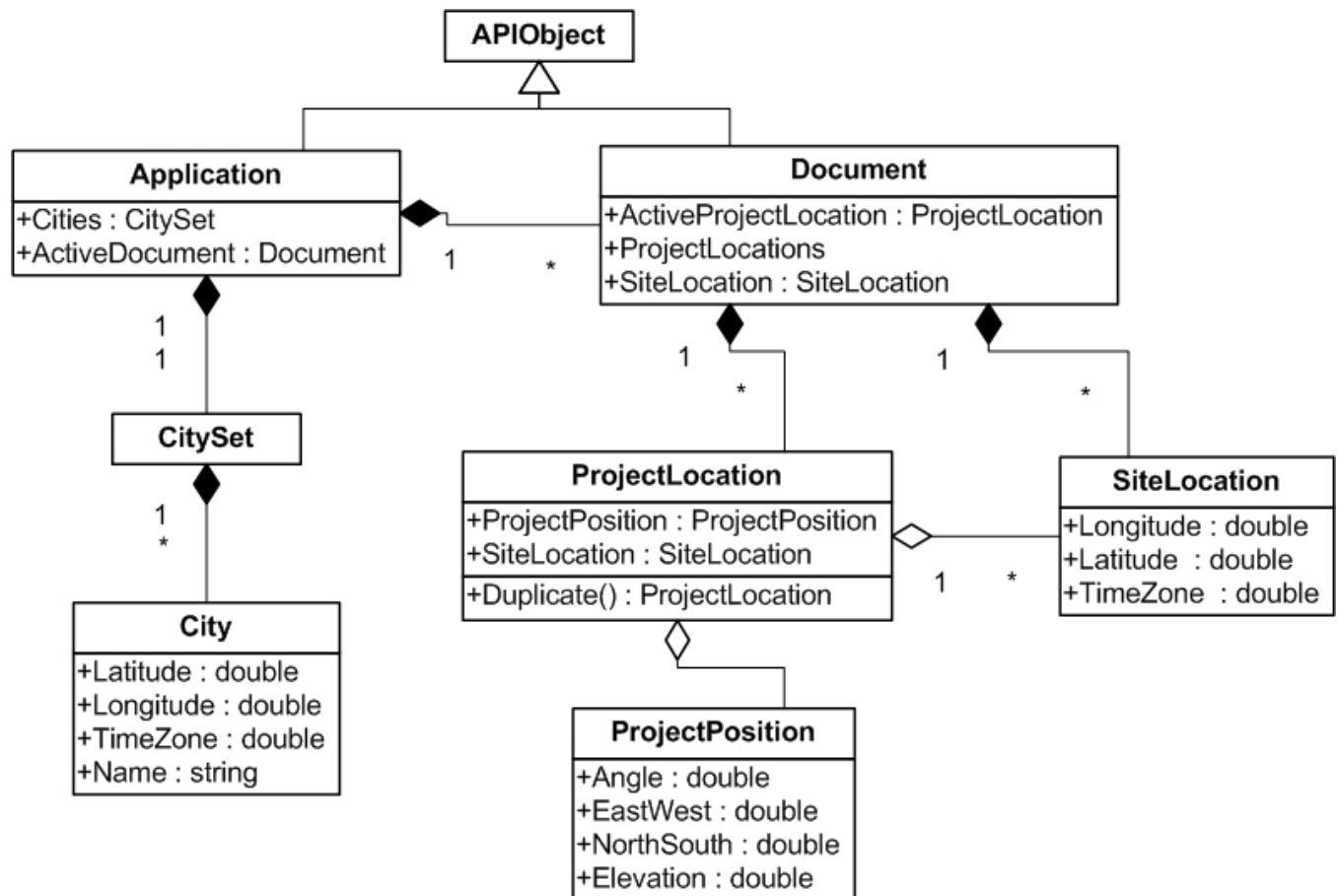


Figure 150: Site-related class diagram

19.1 Place

In the Revit Platform API, the SiteLocation class contains place information including Latitude, Longitude, and Time Zone. This information identifies where the project is located in the world.

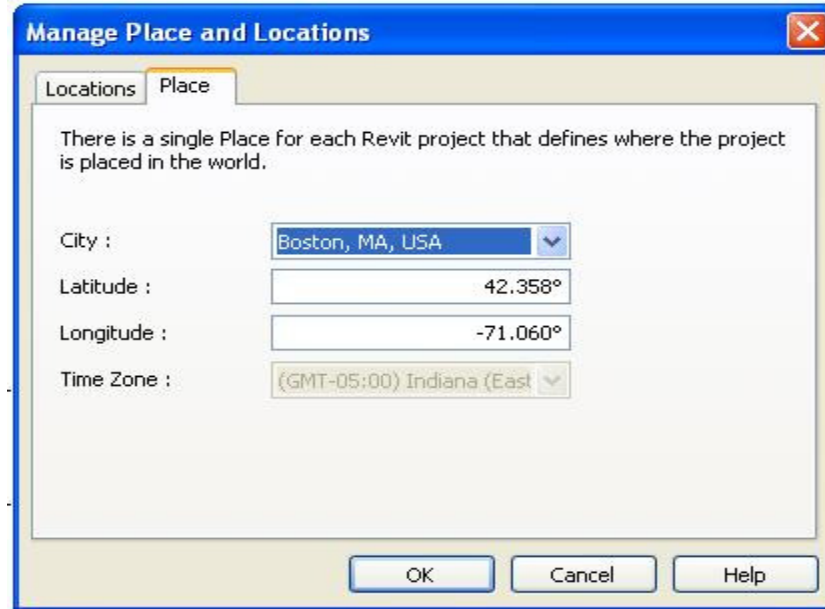


Figure 151: Project Place

19.2 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.

19.3 ProjectLocation

A project only has one site which is the absolute location on the earth. However, it can have different locations relative to the projects around it. Depending on the coordinates and origins in use, there can be many ProjectLocation objects in one project.

By default each Revit project contains at least one named location, Internal. It is the active project location. You can retrieve it using the Document.ActiveProjectLocation property. All existing ProjectLocation objects are retrieved using the Document.ProjectLocations property.

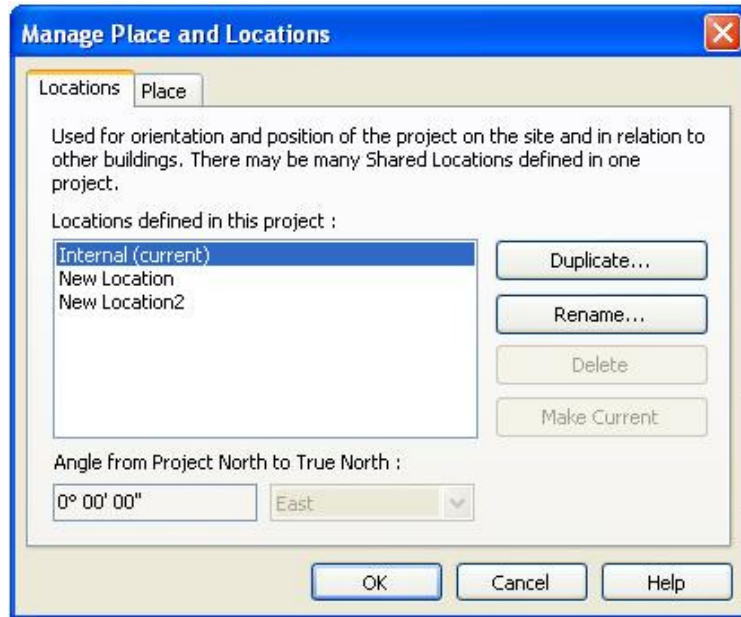


Figure 152: Project locations

19.4 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 153: 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

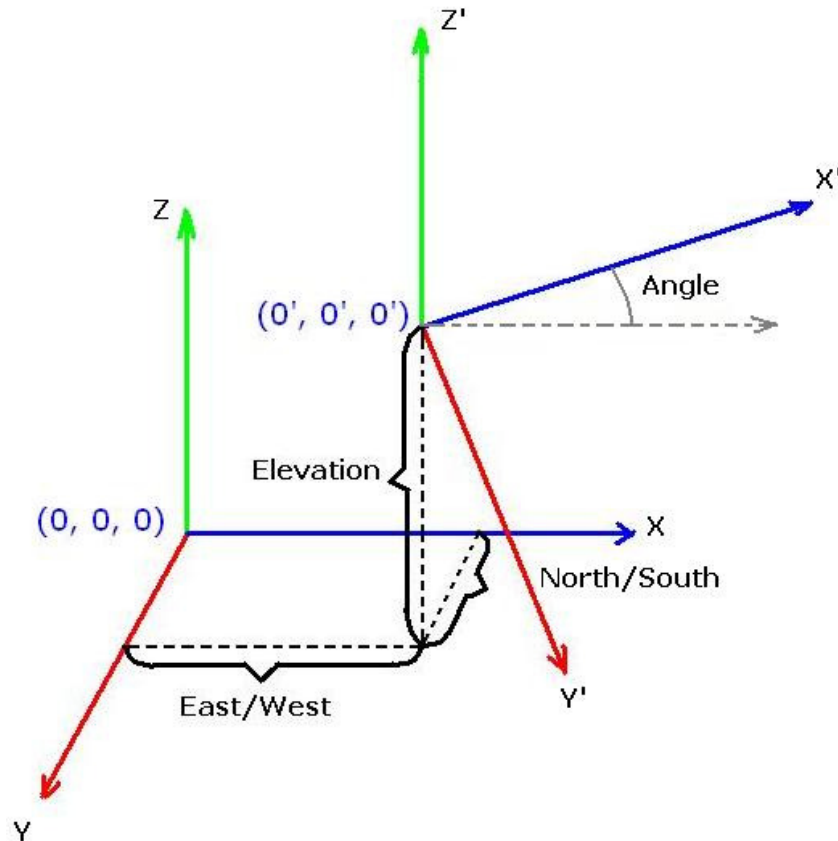


Figure 154: Geographical offset and rotation sketch map

The following sample code illustrates how to retrieve the ProjectLocation object.

Code Region 19-1: Retrieving the ProjectLocation object

```
public void ShowActiveProjectLocationUsage(Autodesk.Revit.Document document)
{
    // Get the project location handle
    ProjectLocation projectLocation = document.ActiveProjectLocation;

    // Show the information of current project location
    XYZ origin = new XYZ(0, 0, 0);
    ProjectPosition position = projectLocation.get_ProjectPosition(origin);
    if (null == position)
    {
        throw new Exception("No project position in origin point.");
    }

    // Format the prompt string to show the message.
    String prompt = "Current project location information:\n";
    prompt += "\n\t" + "Origin point position:";
    prompt += "\n\t\t" + "Angle: " + position.Angle;
    prompt += "\n\t\t" + "East to West offset: " + position.EastWest;
    prompt += "\n\t\t" + "Elevation: " + position.Elevation;
    prompt += "\n\t\t" + "North to South offset: " + position.NorthSouth;
}
```

```

// Angles are in radians when coming from Revit API, so we
// convert to degrees for display
const double angleRatio = Math.PI / 180;    // angle conversion factor

SiteLocation site = projectLocation.SiteLocation;
prompt += "\n\t" + "Site location:";
prompt += "\n\t\t" + "Latitude: " + site.Latitude / angleRatio + "°";
prompt += "\n\t\t" + "Longitude: " + site.Longitude / angleRatio + "°";
prompt += "\n\t\t" + "TimeZone: " + site.TimeZone;

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

```

Note: There is only one active project location at a time. To see the result after changing the ProjectLocation geographical offset and rotation, change the Orientation property from Project North to True North in the plan view Properties dialog box.

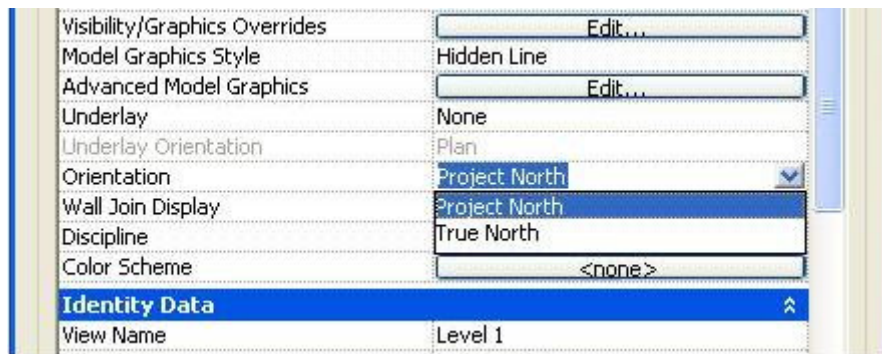


Figure 155: Set orientation value in plan view Properties dialog box

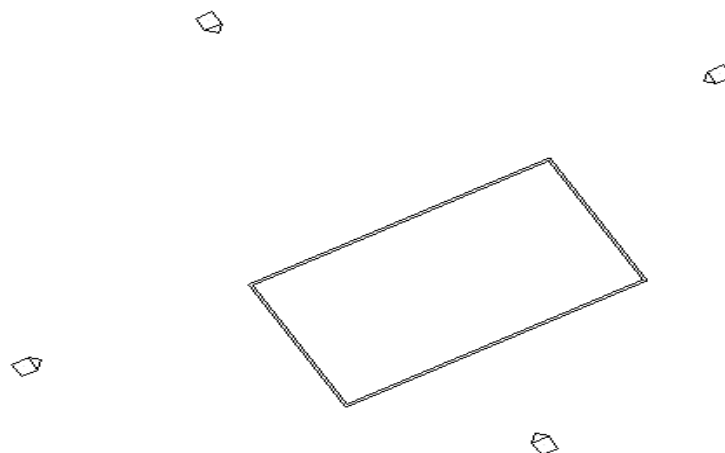


Figure 156: Project is rotated 30 degrees from Project North to True North

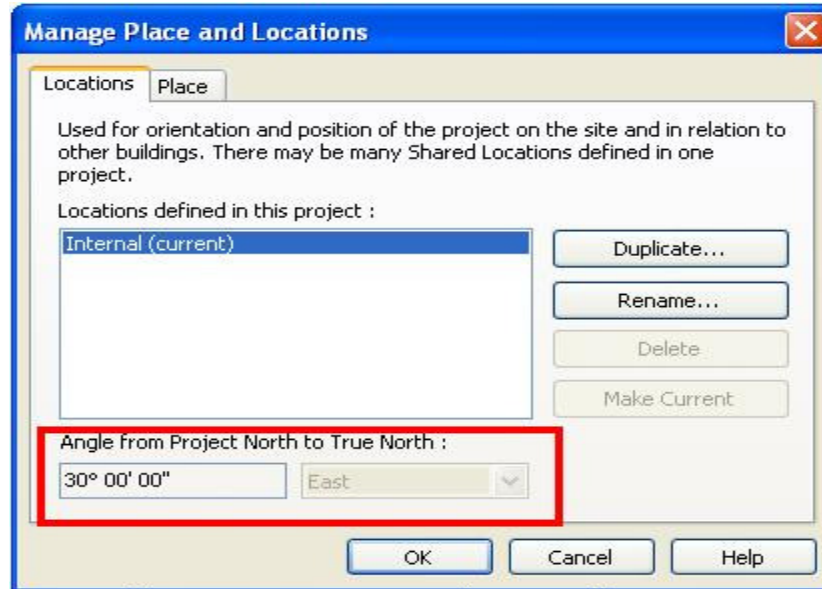


Figure 157: Project location information

19.4.1 Creating and Deleting Project Locations

Create new project locations by duplicating an existing project location using the `Duplicate()` method. The following code sample illustrates how to create a new project location using the `Duplicate()` method.

Code Region 19-2: Creating a project location

```
public ProjectLocation DuplicateLocation(Autodesk.Revit.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 19-3: Deleting a project location

```
public void DeleteLocation(Autodesk.Revit.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)
        {
            ElementIdSet elemSet = document.Delete(projectLocation);
            if (elemSet.Size > 0)
            {
                MessageBox.Show("Project Location Deleted!", "Revit");
            }
        }
    }
}
}
```

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.

20 Shared Parameters

Shared Parameters are parameter definitions stored in an external text file. The definitions are identified by a unique identifier generated when the definition is created and can be used in multiple projects.

This chapter introduces how to gain access to shared parameters through the Revit Platform API. The following overview shows how to get a shared parameter and bind it to Elements in certain Categories:

- Set SharedParametersFileName
- Get the External Definition
- Binding

20.1 Definition File

The DefinitionFile object represents a shared parameter file. The definition file is a common text file. Do not edit the definition file directly; instead, edit it using the UI or the API.

20.1.1 Definition File Format

The shared parameter definition file is a text file (.txt) with two blocks: GROUP and PARAM.

Code Region 20-1: Parameter definition file example

```
# This is a Revit shared parameter file.
# Do not edit manually.
*GROUP ID      NAME
GROUP 1        MyGroup
GROUP 2        AnotherGroup
*PARAM GUID    NAME    DATATYPE    DATACATEGORY    GROUP    VISIBLE
PARAM 4b217a6d-87d0-4e64-9bbc-42b69d37dda6    MyParam    TEXT    1    1
PARAM 34b5cb95-a526-4406-806d-dae3e8c66fa9    Price    INTEGER    2    1
PARAM 05569bb2-9488-4be4-ae21-b065f93f7dd6    areaTags    FAMILYTYPE    -2005020    1
```

- The GROUP block contains group entries that associate every parameter definition with a group. The following fields appear in the GROUP block:
 - ID - Uniquely identifies the group and associates the parameter definition with a group.
 - Name - The group name displayed in the UI.

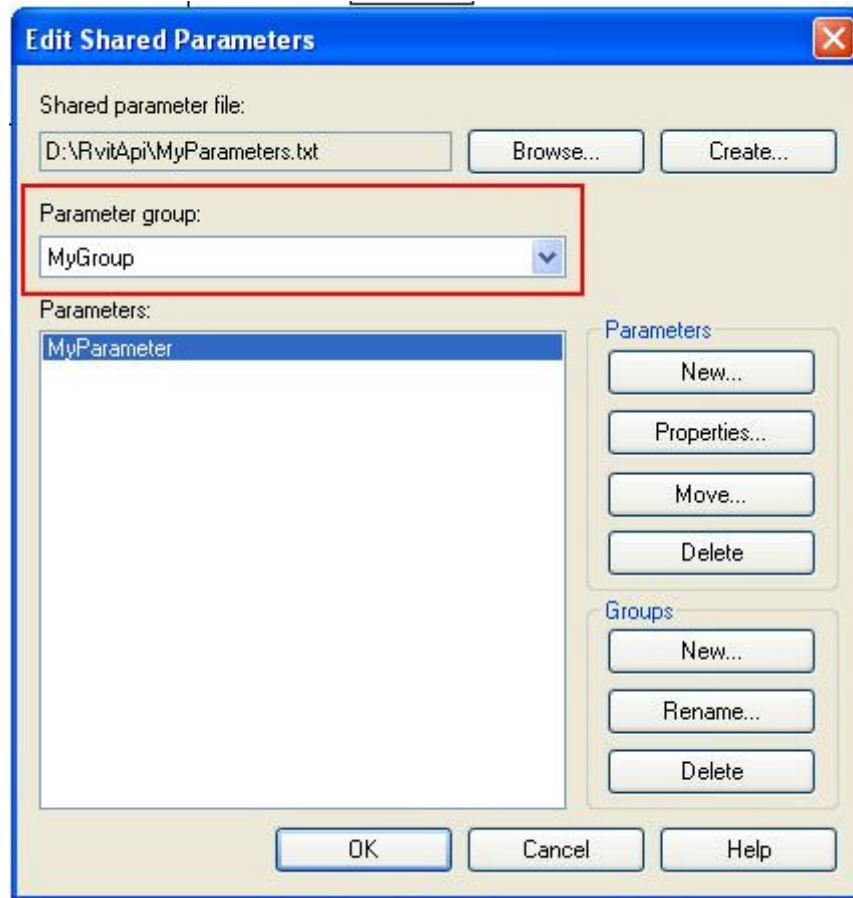


Figure 158: Edit Shared Parameters (Group and Parameter)

- The PARAM block contains parameter definitions. The following fields appear in the PARAM block:
 - GUID - Identifies the parameter definition.
 - NAME - Parameter definition name.
 - DATATYPE - Parameter type. This field can be a common type (TEXT, INTEGER, etc.), structural type (FORCE, MOMENT, etc.) or common family type (Area Tags, etc). Common type and structural type parameters are specified in the text file directly (e.g.: TEXT, FORCE). If the value of the DATATYPE field is FAMILYTYPE, an extra number is added. For example, FAMILYTYPE followed by -2005020 represents Family type: Area Tags.

Code Region 20-2: Shared Parameter FAMILYTYPE example

```
FAMILYTYPE    -2005020
```



Figure 159: New parameter definition

- **GROUP** - A group ID used to identify the group that includes the current parameter definition.
- **VISIBLE** - Identifies whether the parameter is visible. The value of this field is 0 or 1.
0 = invisible
1 = visible

As the definition file sample shows, there are two groups:

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

20.2 Definition File Access

In the add-in code, complete the following steps to gain access to the definition file:

1. Specify the `Autodesk.Revit.Options.Application.SharedParametersFilename` property with an existing text file or a new one.
2. Open the shared parameters file, using the `Application.OpenSharedParameterFile()` method.
3. Open an existing group or create a new group using the `DefinitionFile.Groups` property.
4. Open an existing external parameter definition or create a new definition using the `DefinitionGroup.Definitions` property.

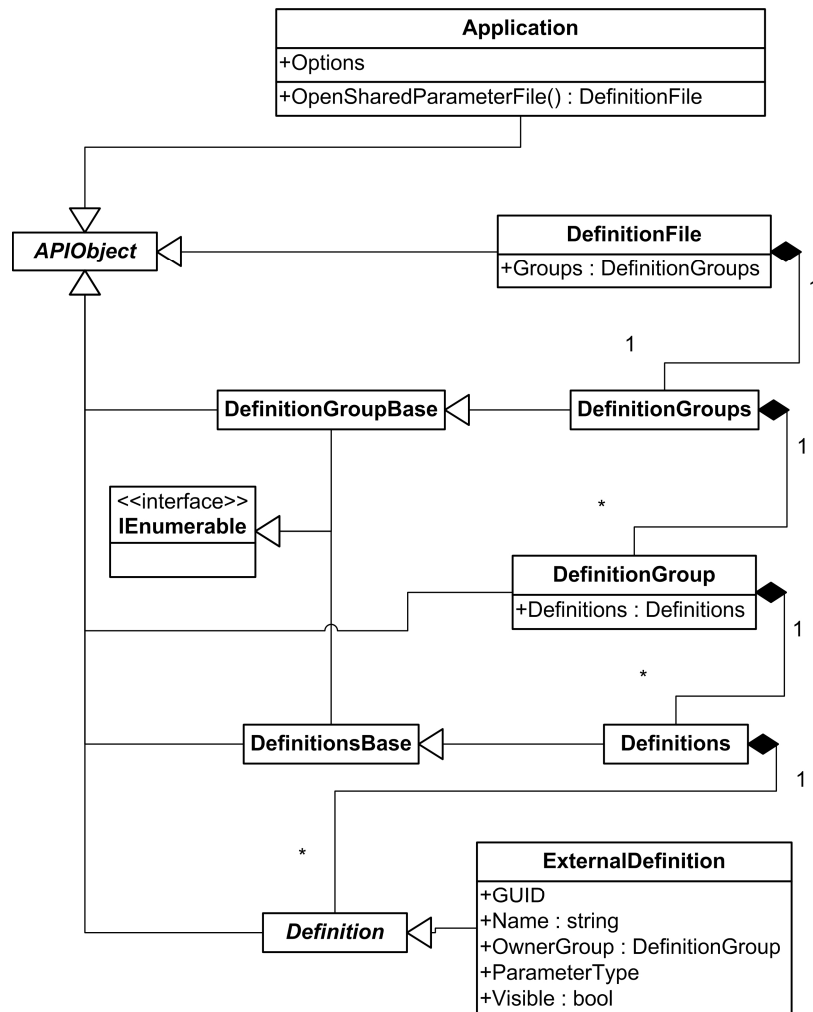


Figure 160: Get the ExternalDefinition type object diagram

The following list provides more information about the classes and methods in the previous diagram.

- Autodesk.Revit.Parameters.DefinitionFile Class - The DefinitionFile object represents one shared parameter file.
 - The object contains a number of Group objects.
 - Shared parameters are grouped for easy management and contain shared parameter definitions.
 - You can add new definitions as needed.
 - The DefinitionFile object is retrieved using the Application.OpenSharedParameterFile method.
- Autodesk.Revit.Parameters.ExternalDefinition Class - The ExternalDefinition class is derived from the Definition class.
 - The ExternalDefinition object is created by a DefinitionGroup object from a shared parameter file.
 - External parameter definitions must belong to a Group which is a collection of shared parameter definitions.
- Autodesk.Revit.Options.Application.SharedParametersFilename Property - Get and set the shared parameter file path using the Autodesk.Revit.Options.SharedParametersFilename property.

- By default, Revit does not have a shared parameter file.
- Initialize this property before using. If it is not initialized, an exception is thrown.
- Autodesk.Revit.Application.OpenSharedParameterFile Method - This method returns an object representing a Revit shared parameter file.
- Revit uses one shared parameter file at a time.
- The file name for the shared parameter file is set in the Revit Application Options object. If the file does not exist, an exception is thrown.

20.2.1 Create a Shared Parameter File

Because the shared parameter file is a text file, you can create it using code or create it manually.

Code Region 20-3: Creating a shared parameter file

```
private void CreateExternalSharedParamFile(string sharedParameterFile)
{
    System.IO.FileStream fileStream = System.IO.File.Create(sharedParameterFile);
    fileStream.Close();
}
```

20.2.2 Access an Existing Shared Parameter File

Because you can have many shared parameter files for Revit, it is necessary to specifically identify the file and external parameters you want to access. The following two procedures illustrate how to access an existing shared parameter file.

20.2.2.1 Get DefinitionFile from an External Parameter File

Set the shared parameter file path to `app.Options.SharedParametersFilename` as the following code illustrates, then invoke the `Autodesk.Revit.Application.OpenSharedParameterFile` method.

Code Region 20-4: Getting the definition file from an external parameter file

```
private DefinitionFile SetAndOpenExternalSharedParamFile(Autodesk.Revit.Application
application, string sharedParameterFile)
{
    // set the path of shared parameter file to current Revit
    application.Options.SharedParametersFilename = sharedParameterFile;
    // open the file
    return application.OpenSharedParameterFile();
}
```

Note: Consider the following points when you set the shared parameter path:

- During each installation, Revit cannot detect whether the shared parameter file was set in other versions. You must bind the shared parameter file for the new Revit installation again.
- If `Options.SharedParametersFilename` is set to a wrong path, an exception is thrown only when `OpenSharedParameterFile` is called.
- Revit can work with multiple shared parameter files. Even though only one parameter file is used when loading a parameter, the current file can be changed freely.

20.2.2.2 Traverse All Parameter Entries

The following sample illustrates how to traverse the parameter entries and display the results in a message box.

Code Region 20-5: 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);
        }
    }

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

20.2.3 Change the Parameter Definition Owner Group

The following sample shows how to change the parameter definition group owner.

Code Region 20-6: Changing parameter definition group owner

```
private void ReadEditExternalParam(DefinitionFile file)
{
    // get ExternalDefinition from shared parameter file
    DefinitionGroups myGroups = file.Groups;
    DefinitionGroup myGroup = myGroups.get_Item("MyGroup");
    if (null == myGroup)
        return;

    Definitions myDefinitions = myGroup.Definitions;
    ExternalDefinition myExtDef =
        myDefinitions.get_Item("MyParam") as ExternalDefinition;
    if (null == myExtDef)
        return;

    StringBuilder strBuilder = new StringBuilder();
}
```

```

// iterate every property of the ExternalDefinition
strBuilder.AppendLine("GUID: " + myExtDef.GUID.ToString())
    .AppendLine("Name: " + myExtDef.Name)
    .AppendLine("OwnerGroup: " + myExtDef.OwnerGroup.Name)
    .AppendLine("Parameter Group" + myExtDef.ParameterGroup.ToString())
    .AppendLine("Parameter Type" + myExtDef.ParameterType.ToString())
    .AppendLine("Is Visible: " + myExtDef.Visible.ToString());

MessageBox.Show(strBuilder.ToString());

// change the OwnerGroup of the ExternalDefinition
myExtDef.OwnerGroup = myGroups.get_Item("AnotherGroup");
}

```

20.3 Binding

In the add-in code, complete the following steps to bind a specific parameter:

1. Use an InstanceBinding or a TypeBinding object to create a new Binding object that includes the categories to which the parameter is bound.
2. Add the binding and definition to the document using the Document.ParameterBindings object.

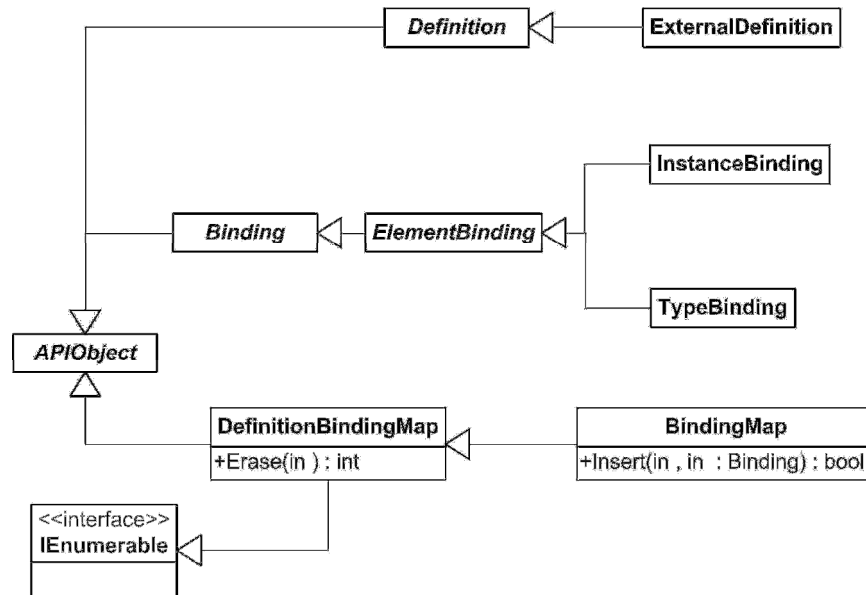


Figure 161: Binding diagram

The following list provides more information about the classes and methods in the previous diagram.

- Autodesk.Revit.Parameters.BindingMap Class - The BindingMap object is retrieved from the Document.ParameterBindings property.
- Parameter binding connects a parameter definition to elements within one or more categories.
- The map is used to interrogate existing bindings as well as generate new parameter bindings using the Insert method.

- Parameters.BindingMap.Insert (Definition, Binding) Method - The binding object type dictates whether the parameter is bound to all instances or just types.
 - A parameter definition cannot be bound to both instances and types.
 - If the parameter binding exists, the method returns false.

20.3.1 Type Binding

The Autodesk.Revit.Parameters.TypeBinding objects are used to bind a property to a Revit type, such as a wall type. It differs from Instance bindings in that the property is shared by all instances identified in type binding. Changing the parameter for one type affects all instances of the same type.

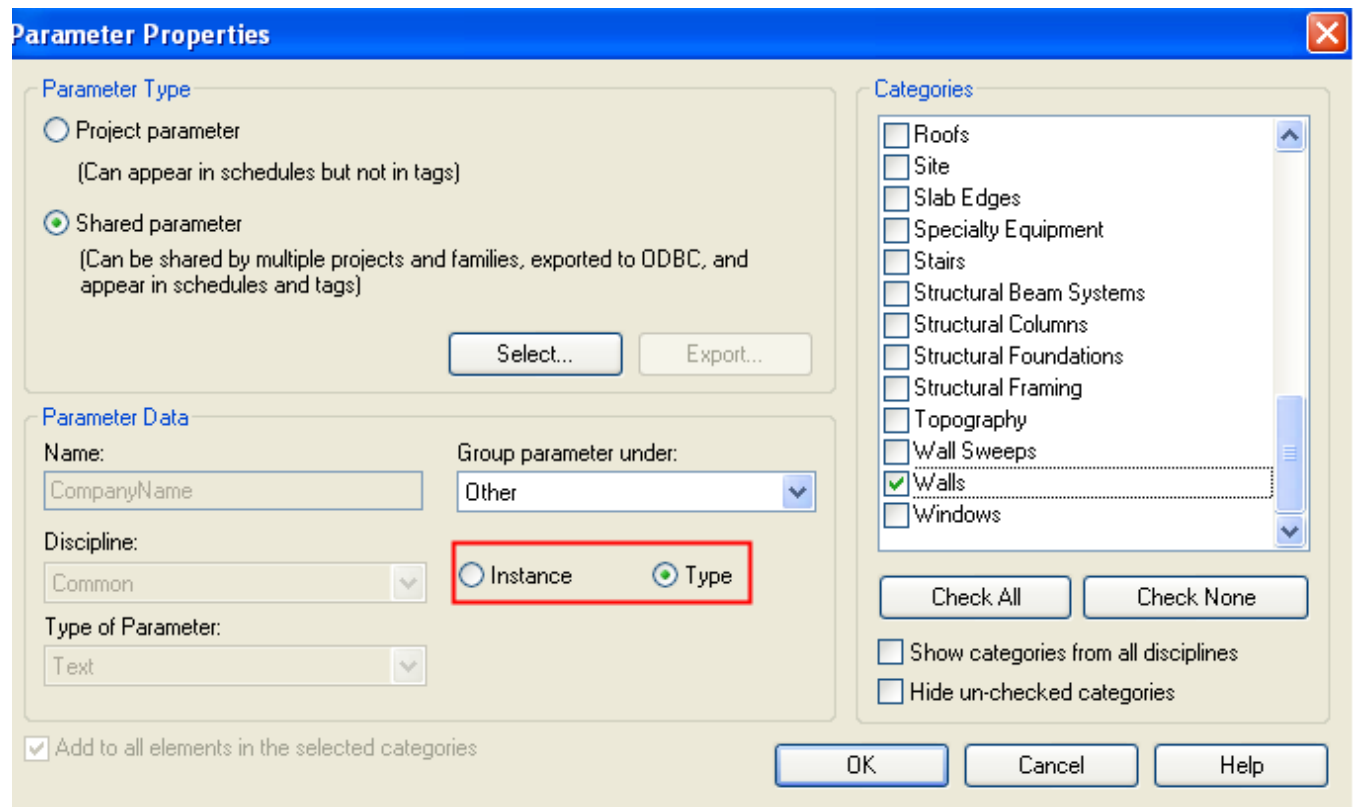


Figure 162: Parameter Properties dialog box Type Binding

The following code segment demonstrates how to add parameter definitions using a shared parameter file. The following code performs the same actions as using the dialog box in the previous picture. Parameter definitions are created in the following order:

1. A shared parameter file is created.
2. A definition group and a parameter definition are created for the Walls type.
3. The definition is bound to the wall type parameter in the current document based on the wall category.

Code Region 20-7: Adding type parameter definitions using a shared parameter file

```
public bool SetNewParameterToTypeWall (Autodesk.Revit.Application app, DefinitionFile
myDefinitionFile)
{
    // Create a new group in the shared parameters file
    DefinitionGroups myGroups = myDefinitionFile.Groups;
    DefinitionGroup myGroup = myGroups.Create("MyParameters");
```

```

// Create a type definition
Definition myDefinition_CompanyName =
    myGroup.Definitions.Create("CompanyName", ParameterType.Text);

// Create a category set and insert category of wall to it
CategorySet myCategories = app.Create.NewCategorySet();
// Use BuiltInCategory to get category of wall
Category myCategory = app.ActiveDocument.Settings.Categories.get_Item(
    BuiltInCategory.OST_Walls);
myCategories.Insert(myCategory);

//Create an object of TypeBinding according to the Categories
TypeBinding typeBinding = app.Create.NewTypeBinding(myCategories);

// Get the BindingMap of current document.
BindingMap bindingMap = app.ActiveDocument.ParameterBindings;

// Bind the definitions to the document
bool typeBindOK = bindingMap.Insert(myDefinition_CompanyName, typeBinding,
    BuiltInParameterGroup.PG_TEXT);
return typeBindOK;
}

```

20.3.2 Instance Binding

The Autodesk.Revit.Parameters.InstanceBinding object indicates binding between a parameter definition and a parameter in certain category instances. The following diagram illustrates Instance Binding in the Walls category.

Once bound, the parameter appears in all property dialog boxes for the instance. Changing the parameter in any one instance does not change the value in any other instance.

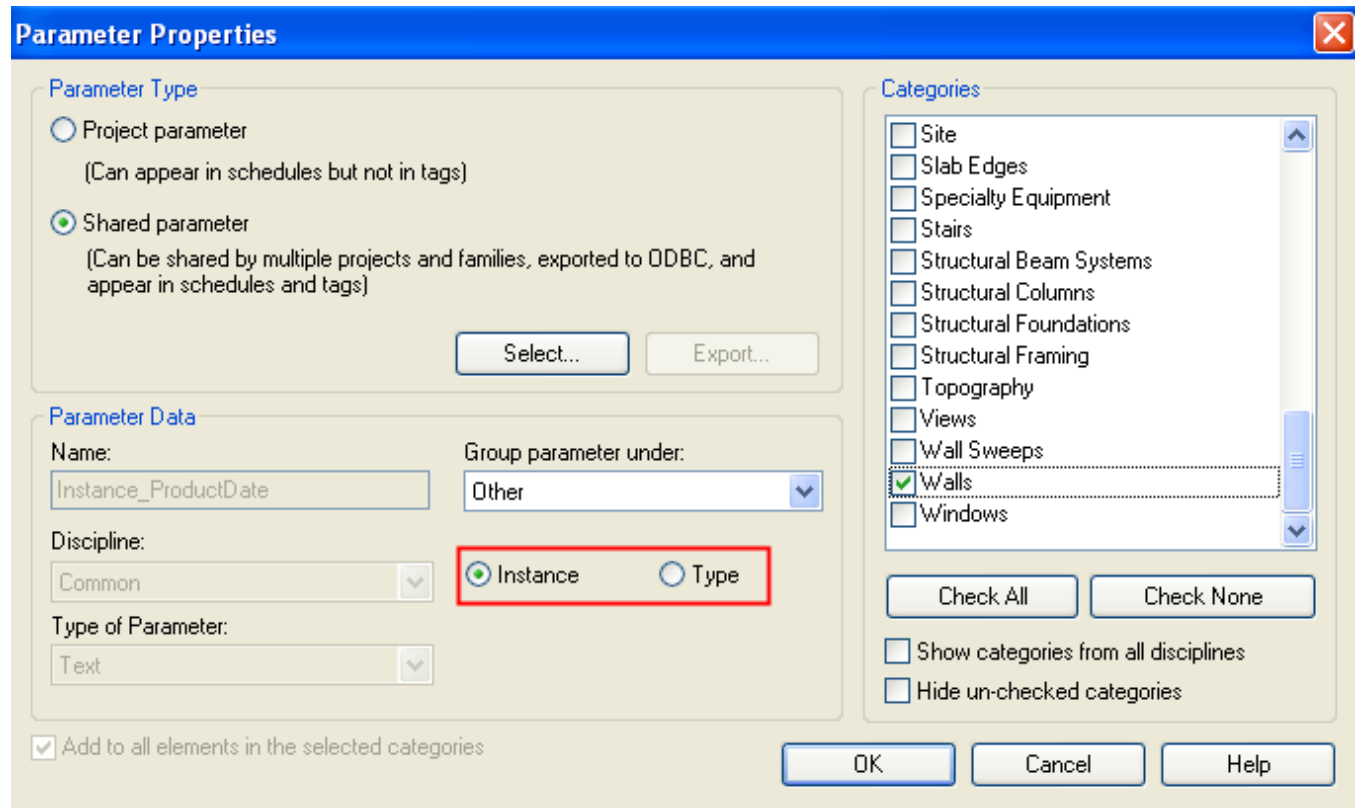


Figure 163: Parameter Properties dialog box Instance Binding

The following code sample demonstrates how to add parameter definitions using a shared parameter file. Parameter definitions are added in the following order:

1. A shared parameter file is created
2. A definition group and a definition for all Walls instances is created
3. Definitions are bound to each wall instance parameter in the current document based on the wall category.

Code Region 20-8: Adding instance parameter definitions using a shared parameter file

```
public bool SetNewParameterToInsanceWall(Autodesk.Revit.Application app, DefinitionFile
myDefinitionFile)
{
    // create a new group in the shared parameters file
    DefinitionGroups myGroups = myDefinitionFile.Groups;
    DefinitionGroup myGroup = myGroups.Create("MyParameters1");

    // create an instance definition in definition group MyParameters
    Definition myDefinition_ProductDate =
        myGroup.Definitions.Create("Instance_ProductDate", ParameterType.Text);

    // create a category set and insert category of wall to it
    CategorySet myCategories = app.Create.NewCategorySet();
    // use BuiltInCategory to get category of wall
    Category myCategory = app.ActiveDocument.Settings.Categories.get_Item(
        BuiltInCategory.OST_Walls);
    myCategories.Insert(myCategory);
}
```

```
//Create an instance of InstanceBinding
InstanceBinding instanceBinding = app.Create.NewInstanceBinding(myCategories);

// Get the BingdingMap of current document.
BindingMap bindingMap = app.ActiveDocument.ParameterBindings;

// Bind the definitions to the document
bool instanceBindOK = bindingMap.Insert(myDefinition_ProductDate,
                                         instanceBinding, BuiltInParameterGroup.PG_TEXT);

return instanceBindOK;
}
```

21 Transactions

Sometimes it is convenient (and safer) to group multiple operations together and treat them as a single operation. For example, in multiple operations creating a model object, if one operation fails, the document can be left in an inconsistent state. You can also use a transaction to group multiple operations, and then prompt the user to cancel or continue, where cancelling aborts the transaction. A transaction is a mechanism that lets you group multiple operations, and easily roll back all operations in the transaction if one of them fails.

The following rules apply to transactions:

- A transaction is successful if and only if all operations within the transaction execute successfully.
- Any operation that fails causes the entire transaction to be aborted.
- If a transaction is aborted, all operations within that transaction are cancelled.

With few exceptions, all changes in persistent data in the document are grouped into Transactions. A Transaction corresponds to a single Undo action which is the main use of transactions in Revit. This chapter introduces the two uses for Transaction and the limits that you must consider.

21.1 Usage

A transaction wraps a set of operations:

- The `Document.BeginTransaction()` method starts the transaction
- The `Document.EndTransaction()` method finishes the transaction when all operations are successful.
- The `Document.AbortTransaction()` method rolls back the whole transaction if an exception or a required abort operation occurs.

21.1.1 Atomic User Actions

The main transaction function in Revit is to initiate atomic user actions. All changes to persistent data are grouped into Transactions. A Transaction corresponds to a single undo action. Some changes affecting the display but not the model content can persist after saving the file but these situations are not managed by transactions. Transient Elements temporarily added to the document by editors are not persistent data. Unless the elements become non-transient, their addition and removal is not managed by transactions.

Examples of atomic user actions include the following:

- Creating a wall, door, window, and so on.
- Cutting/copying/pasting
- Moving/Rotating
- Deleting
- Choosing OK or Apply in a Properties dialog box.

Examples of non-atomic actions include the following:

- Pressing the left mouse button to start a pan. (A display change.)
- Creating a poly-wall. (More than one transaction.)
- Dragging the end of a wall when creating a wall command. (Less than one transaction.)
- Selecting (without persistent effect.)

A transaction typically contains several smaller changes, including a direct user change and the resultant effects of regeneration. However, reversing a subset of these can leave the document in an inconsistent state. Therefore, changes are grouped and rollback to the document state is permitted only to certain defined points in the change history. The following sample program demonstrates the transaction API function:

Code Region 21-1: Using transactions

```
// Get the document
Autodesk.Revit.Document document = application.ActiveDocument;

// Create a geometry line in revit application
XYZ Point1 = new XYZ(0, 0, 0);
XYZ Point2 = new XYZ(10, 0, 0);
XYZ Point3 = new XYZ(10, 10, 0);
XYZ Point4 = new XYZ(0, 10, 0);

// Transient Elements,so they are not managed by transactions
document.BeginTransaction();
Line geomLine1 = application.Create.NewLine(Point1, Point2, true);
Line geomLine2 = application.Create.NewLine(Point4, Point3, true);
Line geomLine3 = application.Create.NewLine(Point1, Point4, true);

// Create a geometry plane in revit application
XYZ origin = new XYZ(0, 0, 0);
XYZ normal = new XYZ(1, 1, 0);
Plane geomPlane = application.Create.NewPlane(normal, origin);

// Create a sketch plane in current document
SketchPlane sketch = document.Create.NewSketchPlane(geomPlane);

// non-transient elements,so they are grouped into transaction as atomic user actions
document.BeginTransaction();
// Create a ModelLine element using the created geometry line and sketch plane
ModelLine line1 = document.Create.NewModelCurve(geomLine1, sketch) as ModelLine;
ModelLine line2 = document.Create.NewModelCurve(geomLine2, sketch) as ModelLine;
ModelLine line3 = document.Create.NewModelCurve(geomLine3, sketch) as ModelLine;

if (DialogResult.OK == MessageBox.Show("Click OK to call EndTransaction, otherwise
AbortTransaction.", "Revit", MessageBoxButtons.OKCancel))
{
    document.EndTransaction();
}
else
{
    document.AbortTransaction();
}
```

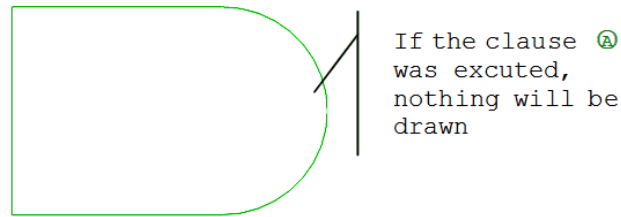


Figure 164: Transaction result

The transaction timeline for this sample is as follows:

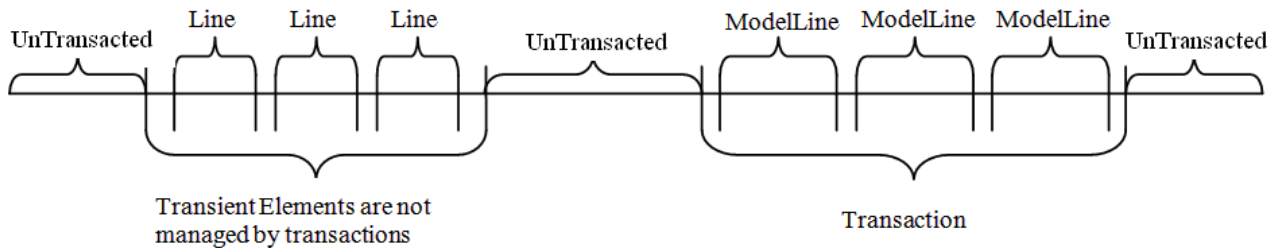


Figure 165: Transaction timeline

21.1.2 Getting Created Element Geometry and AnalyticalModel

Transactions also trigger the assignment of the Geometry and AnalyticalModel properties for Elements. You cannot access these properties unless the Element is created inside a transaction, and the transaction completes successfully.

For more details about the AnalyticalModel property, refer to the [AnalyticalModel](#) section in the [Revit Structure](#) chapter. For more details about the Geometry property, refer to the [Geometry](#) chapter.

The following sample program demonstrates how a transaction populates these properties:

Code Region 21-2: Transaction populating Geometry and AnalyticalModel properties

```
public void TransactionDuringElementCreation(Autodesk.Revit.Application application, Level
level)
{
    Autodesk.Revit.Document document = application.ActiveDocument;

    // Build a location line for the wall creation
    XYZ start = new XYZ(0, 0, 0);
    XYZ end = new XYZ(10, 10, 0);
    Autodesk.Revit.Geometry.Line geomLine = application.Create.NewLineBound(start, end);

    document.BeginTransaction();

    // Create a wall using the location line
    Wall wall = document.Create.NewWall(geomLine, level, true);
    document.EndTransaction();

    // If there was no transaction to group the creation of the wall above,
```

```
// then you could not get the value of Geometry and AnalyticalModel.
Autodesk.Revit.Geometry.Options options = application.Create.NewGeometryOptions();
Autodesk.Revit.Geometry.Element geoelem = wall.get_Geometry(options);
Autodesk.Revit.Structural.AnalyticalModel analyticalmodel = wall.AnalyticalModel;
}
```

The transaction timeline for this sample is as follows:

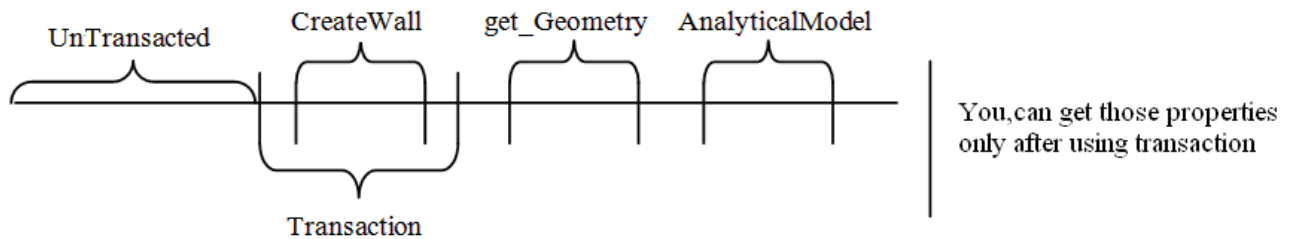


Figure 166: Transaction timeline

21.2 Boundaries

Because you are in charge of starting, ending, and aborting transactions, it is important to be aware of transaction boundaries. A transaction boundary is the period between starting and ending or aborting a transaction.

It is best to confine your boundary to a small scope. For example, if you start a transaction in a function, try to end or abort it before you return from that function since you might not have knowledge of the transaction outside of the function. It is not necessary to follow this rule if you want to maintain global managers for your transaction activity, but you should be responsible for aborting or ending all transactions you start.

The following sample program demonstrates transaction boundaries:

Code Region 21-3: Transaction boundaries

```
public bool TransactionBoundaries(Autodesk.Revit.Document document)
{
    bool result = true;
    try
    {
        document.BeginTransaction();
        //... Some operation

        // End the transaction before returning from the function
        document.EndTransaction();
    }
    catch
    {
        // Abort transaction when exception occurred
        document.AbortTransaction();
        result = false;
    }
}
```



```
return result;  
}
```

Most applications use transaction management for operations on objects. The operations are committed at the end of the outermost transaction.

- A Revit command boundary is the boundary of your transaction as far as you can stretch.
- No active transactions exist when a command ends. If it does, Revit aborts (or crashes).

Generally speaking, it is a good idea to start a transaction when one of your functions is invoked and end the transaction when you return from that function. Most actions in Revit refer to this idea using the `Document.BeginTransaction` method and `Document.EndTransaction` method.

22 Events

Events are messages 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 ("pre" event) or has just happened ("post" event), and take some action related to that event.

Revit provides access to events at both the Application level (such as `ApplicationClosing` or `DocumentOpened`) and the Document level (such as `DocumentClosing` and `DocumentPrinting`).

22.1 Application Events

The following are some of the events that are available at the Application level:

- `ApplicationClosing` – notification when the Revit application is about to be closed
- `DialogBoxShowing` – notification when Revit is about to show a dialog or message box
- `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
- `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
- `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
- `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

22.2 Document Events

The following events are available at the Document level:

- DocumentClosing – notification when Revit is about to close 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

22.3 Registering Events

Taking advantage of events is a two step process. First, you must have a function that will handle the event notification. This function must take two parameters, the first is an Object that denotes the “sender” of the event notification, the second is an event-specific object that contains event arguments specific to that event. For example, to register the DocumentSavingAs event, your event handler must take a second parameter that is a DocumentSavingAsEventArgs object.

The second part of using an event is registering the event with Revit. This is done in the OnStartup() function through the ControlledApplication parameter.

The following example registers the DocumentOpened event, and when that event is triggered, this application will set the address of the project.

Code Region 22-1: Registering Application.DocumentOpened

```
public class Application_DocumentOpened : IExternalApplication
{
    public IExternalApplication.Result OnStartup(ControlledApplication application)
    {
        try
        {
            // Register event.
            application.DocumentOpened += new EventHandler
                <Autodesk.Revit.Events.DocumentOpenedEventArgs>(application_DocumentOpened);
        }
        catch (Exception)
        {
            return IExternalApplication.Result.Failed;
        }

        return IExternalApplication.Result.Succeeded;
    }

    public IExternalApplication.Result OnShutdown(ControlledApplication application)
```

```

{
    // remove the event.
    application.DocumentOpened -= application_DocumentOpened;
    return IExternalApplication.Result.Succeeded;
}

public void application_DocumentOpened(object sender, DocumentOpenedEventArgs args)
{
    // get document from event args.
    Document doc = args.Document;

    doc.ProjectInformation.Address =
        "United States - Massachusetts - Waltham - 1560 Trapelo Road";
}
}

```

22.4 Canceling Events

Events that are triggered before an action has taken place (i.e. DocumentSaving) are often cancellable. 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, you can check for certain criteria in the document and cancel the Save or Save As action.

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 22-2: Canceling an Event

```

private void CheckProjectStatusInitial(Object sender, DocumentSavingAsEventArgs args)
{
    Document doc = args.Document;
    ProjectInfo proInfo = doc.ProjectInformation;

    // Project information is only available for project document.
    if (null != proInfo)
    {
        if (string.IsNullOrEmpty(proInfo.Status))
        {
            // cancel the save as process.
            args.Cancel = true;
            MessageBox.Show("Status project parameter is not set. Save is aborted.");
        }
    }
}

```

23 Revit Architecture

This chapter covers API functionality that is specific to Revit Architecture, namely:

- Functionality related to rooms (Element.Room, RoomTag, etc.)
- Limited energy analysis functions

23.1 Rooms

The following sections cover information about the room class, its parameters, and how to use the room class in the API.

23.1.1 Room, Area, and Tags

The Room class is used to represent rooms and elements such as room schedule and area plans. The properties and create function for different rooms, areas, and their corresponding tags in the API are listed in the following table:

Element	Class	Category	Boundary	Location	Can Create
Room in Plan View	Room	OST_Rooms	Has if in an enclosed region	LocationPoint	NewRoom() and NewRooms() except for NewRoom(Phase)
Room in Schedule View	Room	OST_Rooms	Null	Null	NewRoom(Phase)
Area	Room	OST_Areas	Always has	LocationPoint	No
Room Tag	RoomTag	OST_RoomTags		LocationPoint	Creation.Document.NewRoomTag()
Area Tag	FamilySymbol	OST_AreaTags		LocationPoint	No

Table 47: Room, Area, and Tags relationship

Note: Room.Name is the combination of the room name and room number. Use the ROOM_NAME BuiltInParameter to get the room name. In the following picture, the Room.Name returns "Master Bedroom 2".

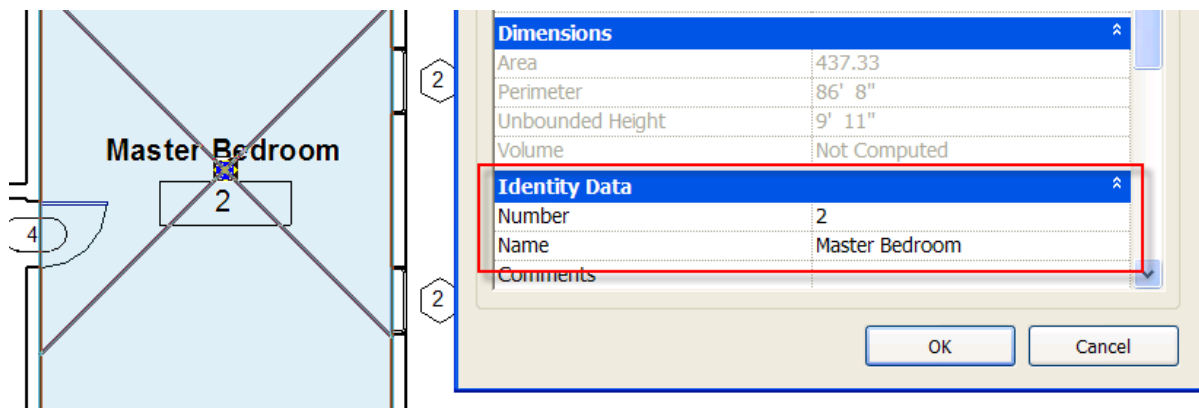


Figure 167: Room name and number

Note: As an Annotation Element, the specific view is available using RoomTag.View. Never try to set the RoomTag.Name property because the name is automatically assigned; otherwise an exception is thrown.

23.1.2 Creating a room

The following code illustrates the simplest way to create a room at a certain point in a specific level:

Code Region 23-1: Creating a room

```
Room CreateRoom(Autodesk.Revit.Document document, Level level)
{
    // Create a UV structure which determines the room location
    UV roomLocation = new UV(0, 0);

    // Create a new room
    Room room = document.Create.NewRoom(level, roomLocation);
    if (null == room)
    {
        throw new Exception("Create a new room failed.");
    }

    return room;
}
```

Rooms can be created in a room schedule then inserted into a plan circuit.

- The Document.NewRoom(Phase) method is used to create a new room, not associated with any specific location, and insert it into an existing schedule. Make sure the room schedule exists or create a room schedule in the specified phase before you make the call.
- The Document.NewRoom(Room room, PlanCircuit circuit) method is used to create a room from a room in a schedule and a PlanCircuit.
 - The input room must exist only in the room schedule, meaning that it does not display in any plan view.
 - After invoking the method, a model room with the same name and number is created in the view where the PlanCircuit is located.

For more details about PlanCircuit, see the [Plan Topology](#) section in this chapter.

The following code illustrates the entire process:

Code Region 23-2: Creating and inserting a room into a plan circuit

```
Room InsertNewRoomInPlanCircuit(Autodesk.Revit.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)
{
    // The input room must exist only in the room schedule,
    // meaning that it does not display in any plan view.
    newRoom2 = document.Create.NewRoom(newScheduleRoom, planCircuit);
    // a model room with the same name and number is created in the view
    // where the PlanCircuit is located
    if (null != newRoom2)
    {
        // Give the user some information
        MessageBox.Show("Room placed in Plan Circuit successfully.", "Revit");
    }
}

return newRoom2;
}

```

You can also create rooms in certain levels in batches. The `Document.NewRooms()` method can create rooms for every enclosed region in a given level, to a given phase, or based on a list of `RoomCreationData` objects.

Once a room has been created and added to a location, it can be removed from the location (but still remains in available in the project) by using the `Room.Unplace()` method. It can then be placed in a new location.

23.1.3 Room Boundary

Rooms have boundaries that create an enclosed region where the room is located.

- Boundaries include the following elements:
 - Walls
 - Model lines
 - Columns
 - Roofs

23.1.3.1 Retrieving Room Boundaries

The boundary around a room is contained in its Boundary property. The Room.Boundary property is null when the room is not in an enclosed region or only exists in the schedule. The Room.BoundaryArray is an array of BoundarySegment objects.

The following figure shows a room boundary selected in the Revit UI:

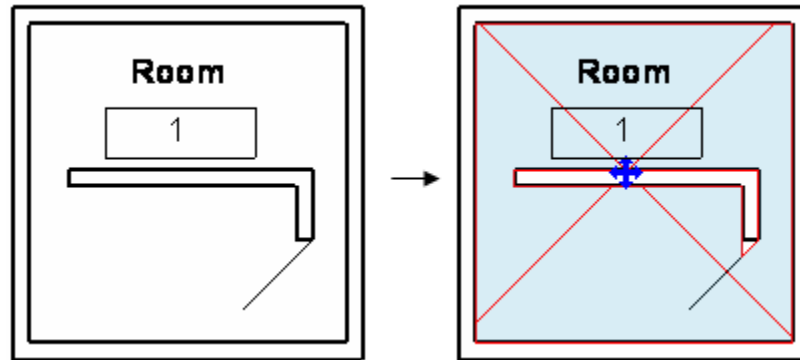


Figure 168: Room boundary

The Room.BoundaryArray segment array size depends on the enclosed region topology. Each BoundarySegmentArray makes a circuit or a continuous line in which one segment joins the next. The following pictures provide several examples. In the following pictures, all walls are Room-Bounding and the model lines category is OST_AreaSeparationLines. If an element is not Room-Bounding, it is excluded from the elements to make the boundary.

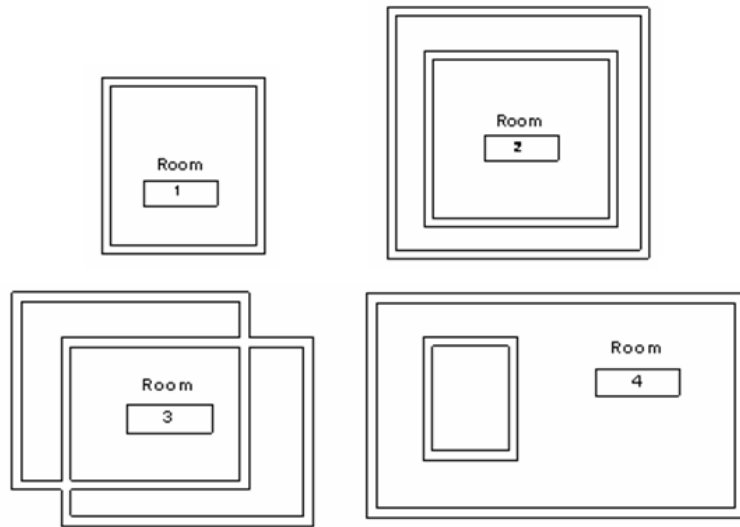


Figure 169: Rooms 1, 2, 3, 4

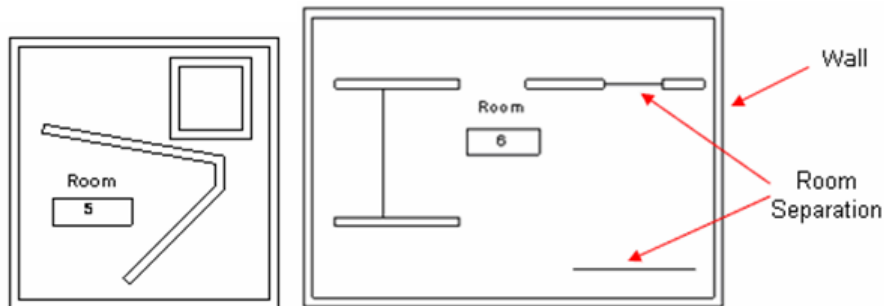


Figure 170: Room 5, 6

The following table provides the Room.Boundary.Size results for the previous rooms:

Room	Room.Boundary.Size
Room 1	1
Room 2	
Room 3	
Room 4	2
Room 5	3
Room 6	

Table 48: Room.Boundary.Size

Note: Walls joined by model lines are considered continuous line segments. Single model lines are ignored.

After getting BoundarySegmentArray, get the BoundarySegment by iterating the array.

23.1.3.2 BoundarySegment

The segments that make the region are represented by the BoundarySegment class; its Element property returns the corresponding element with the following conditions:

- For a ModelCurve element, the category must be BuiltInCategory.OST_AreaSeparationLines meaning that it represents a Room Separator.
- For other elements such as wall, column, and roof, if the element is a room boundary, the Room Bounding parameter (BuiltInParameter.WALL_ATTR_ROOM_BOUNDING) must be true as in the following picture.

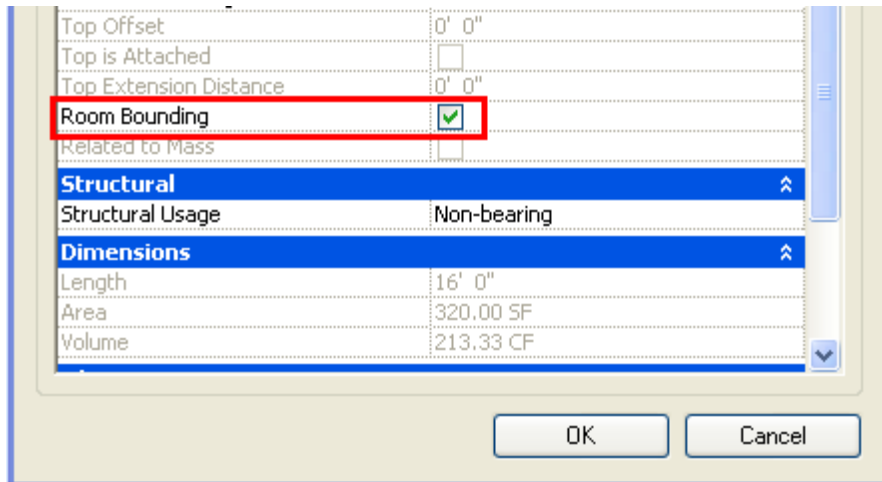


Figure 171: Room Bounding property

The WALL_ATTR_ROOM_BOUNDING BuiltInParameter is set through the API:

Code Region 23-3: Setting room bounding

```
public void SetRoomBounding(Wall wall)
{
    Parameter parameter = wall.get_Parameter(BuiltInParameter.WALL_ATTR_ROOM_BOUNDING);
    parameter.Set(1);    //set "Room Bounding" to true
    parameter.Set(0);    //set "Room Bounding" to false
}
```

Notice how the roof forms the BoundarySegment for a room in the following pictures. The first picture shows Level 3 in the elevation view. The room is created in the Level 3 floor view. The latter two pictures show the boundary of the room and the house in 3D view.

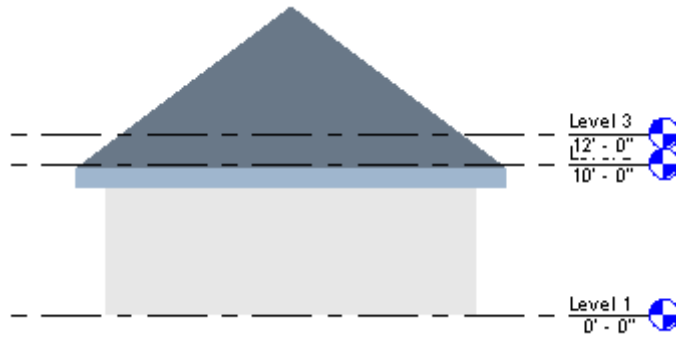


Figure 172: Room created in level 3 view

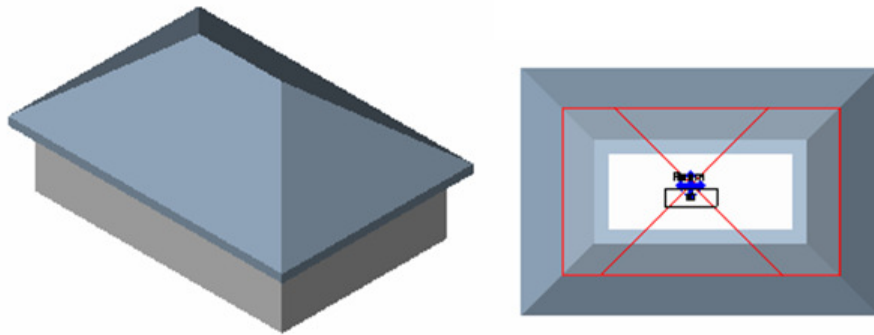


Figure 173: Room boundary formed by roof

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.

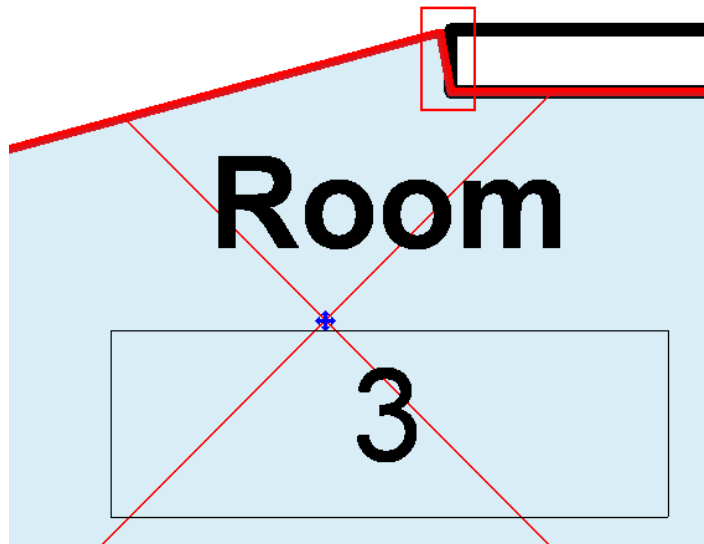


Figure 174: Wall end edge

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.

23.1.3.3 Boundary and Transaction

When you call `Room.Boundary` after creating an Element using the API such as a wall, the wall can change the room boundary. You must make sure the data is updated.

The following illustrations show how the room changes after a wall is created using the Revit Platform API.

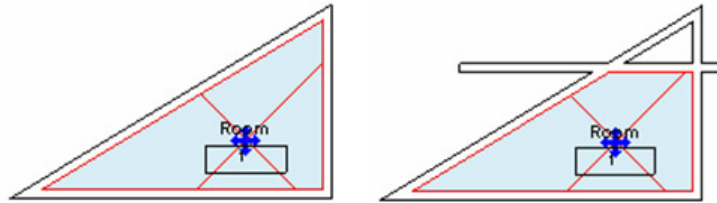


Figure 175: Added wall changes the room boundary

To update the room boundary data, use the transaction mechanism in the following code:

Code Region 23-4: Using a transaction to update room boundary

```
public void UpdateRoomBoundary(Autodesk.Revit.Application application, Room room, Level
level)
{
    Document document = application.ActiveDocument;

    //Get the size before creating a wall
    int size = room.Boundary.get_Item(0).Size;
    string message = "Room boundary size before wall: " + size;

    //Prepare a line
    XYZ startPos = new XYZ(-10, 0, 0);
    XYZ endPos = new XYZ(10, 0, 0);
    Line line = application.Create.NewLine(startPos, endPos, true);

    //Create a new wall and enclose the creating into a single transaction
    document.BeginTransaction();
    Wall wall = document.Create.NewWall(line, level, false);
    document.EndTransaction();

    //Get the new size
    size = room.Boundary.get_Item(0).Size;
    message += "\nRoom boundary size after wall: " + size;
    MessageBox.Show(message, "Revit");
}
```

For more details, see the [Transaction](#) chapter.

23.1.4 Plan Topology

The level plan that rooms lie in have a topology made by elements such as walls and room separators. The `PlanTopology` and `PlanCircuit` classes are used to present the level topology.

- Get the `PlanTopology` object from the `Document` object using the `Level`. In each plan view, there is one `PlanTopology` corresponding to every phase.

- The same condition applies to BoundarySegment, except room separators and Elements whose Room Bounding parameter is true can be a side (boundary) in the PlanCircuit.

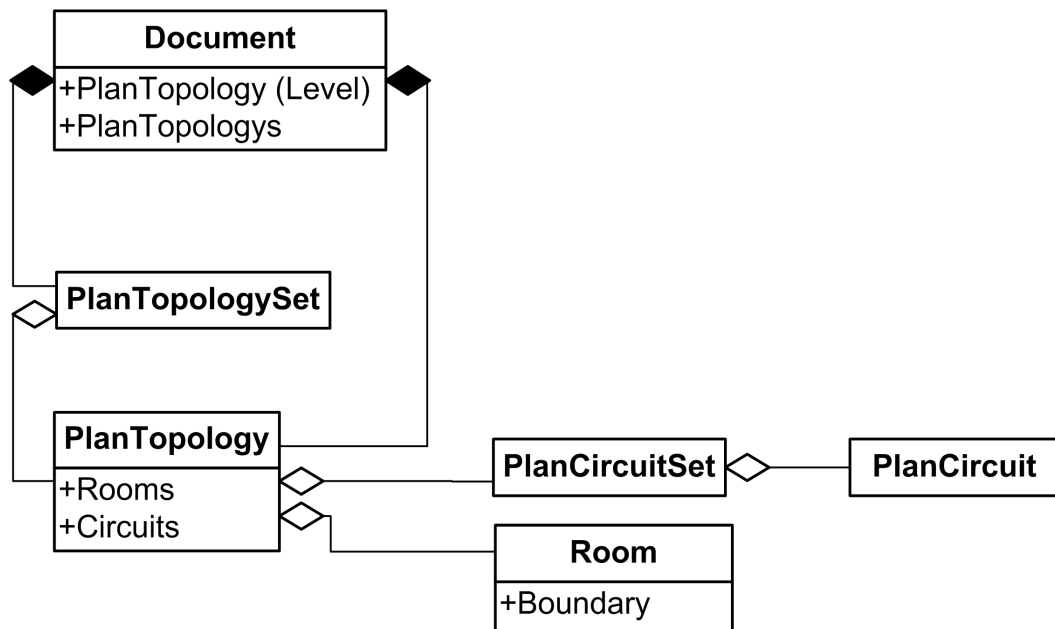


Figure 176: Room and Plan Topology diagram

The `PlanCircuit.SideNum` property returns the circuit side number, which is different from the `BoundarySegmentArray.Size`.

- The `BoundarySegmentArray.Size` recognizes the bottom wall as two walls if there is a branch on the wall.
- `PlanCircuit.SideNum` always sees the bottom wall in the picture as one regardless of the number of branches.

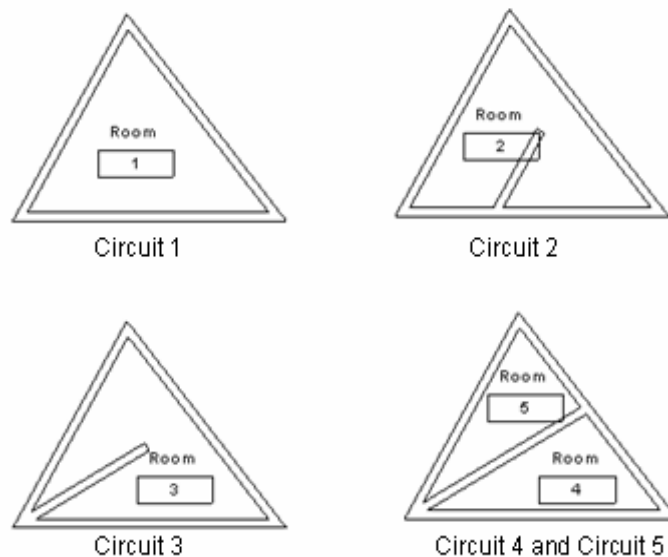


Figure 177: Compare room boundary with PlanCircuit

Circuit	Circuit.SideNum	BoundarySegmentArray.Size for Room

Circuit	Circuit.SideNum	BoundarySegmentArray.Size for Room
Circuit 1	3	3 (Room1)
Circuit 2	4 + 2 = 6	4 + 3 = 7 (Room2)
Circuit 3	3 + 2 = 5	3 + 3 = 6 (Room3)
Circuit 4	3	3 (Room4)
Circuit 5	3	3 (Room5)

Table 49: Compare Room Boundary with PlanCircuit

23.1.5 Room and FamilyInstance

Doors and Windows are special family instances related to Room. Only doors are discussed here since the only difference is that windows have no handle to flip.

The following characteristics apply to doors:

- Door elements can exist without a room.
- In the API (and only in the API), a Door element has two additional properties that refer to the regions on the two opposite sides of a door: ToRoom and FromRoom
- If the region is a room, the property's value would be a Room element.
- If the region is not a room, the property will return null. Both properties may be null at the same time.
- The region on the side into which a door opens, will be ToRoom. The room on the other side will be FromRoom.
- Both properties get dynamically updated whenever the corresponding regions change.

In the following pictures, five doors are inserted into walls without flipping the facing. The table lists the FromRoom, ToRoom, and Room properties for each door. The Room property belongs to all Family Instances.

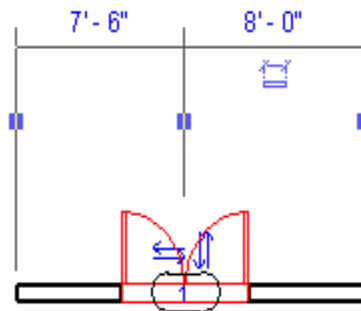


Figure 178: Door 1

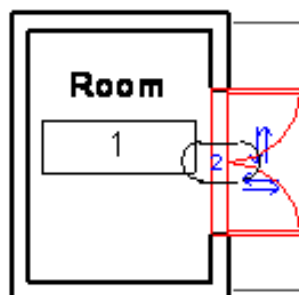


Figure 179: Door 2

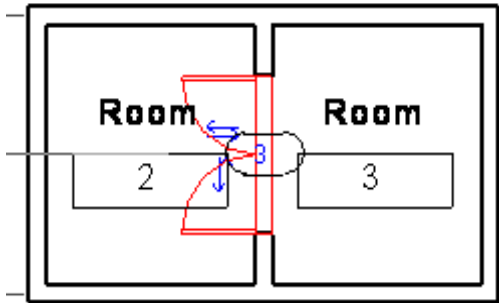


Figure 180: Door 3

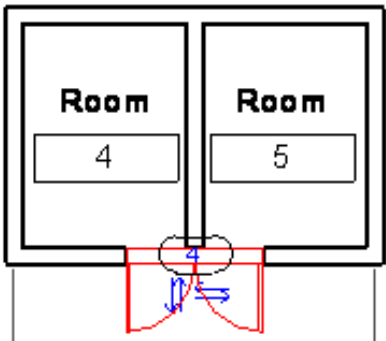


Figure 181: Door 4

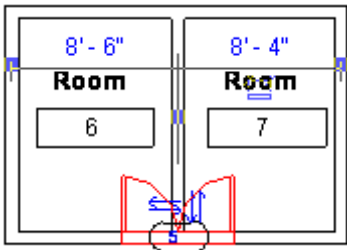


Figure 182: Door 5

Door	FromRoom	ToRoom	Room
Door 1	null	null	null
Door 2	Room 1	null	null
Door 3	Room 3	Room 2	Room 2
Door 4	Room 4	null	null
Door 5	null	Room 6	Room 6

Table 50: Door Properties

All family instances have the Room property, which is the room where an instance is located in the last project phase. Windows and doors face into a room. Change the room by flipping the door or window facing, or by calling `FamilyInstance.FlipFromToRoom()`. For other kinds of instances, such as beams and columns, the Room is the room that has the same boundary as the instance.

The following code illustrates how to get the Room from the family instance. It is necessary to check if the result is null or not.

Code Region 23-5: Getting a room from the family instance

```

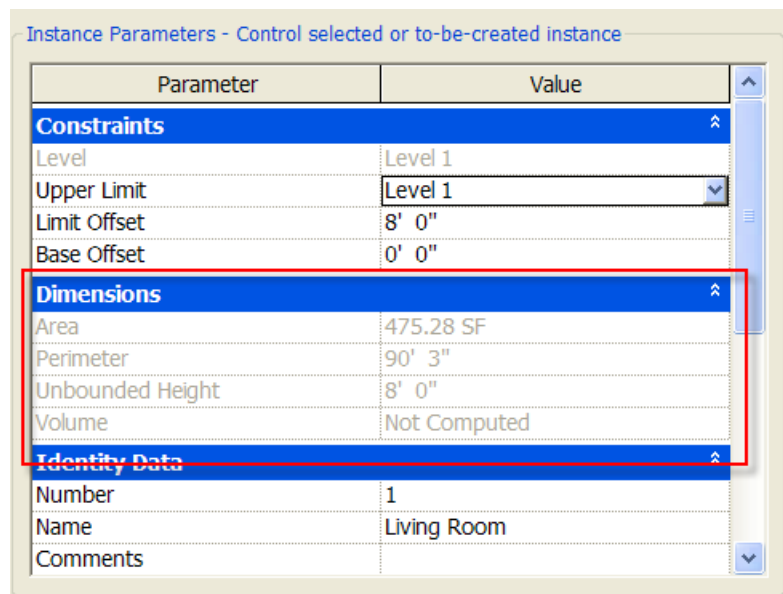
public void GetRoomInfo(FamilyInstance familyInstance)
{
    Room room = familyInstance.Room;
    room = familyInstance.FromRoom; //for door and window family only
    room = familyInstance.ToRoom;   //for door and window family only
    if (null != room)
    {
        //use the room...
    }
}

```

23.1.6 Other Room Properties

The Room class has several other properties you can use to get information about the object. Rooms have these read-only dimension properties:

- Area
- Perimeter
- UnboundedHeight
- Volume
- ClosedShell



This example displays the dimension information for a selected room. Note that the volume calculations setting must be enabled, or the room volume is returned as 0.

Code Region 23-6: Getting a room's dimensions

```

public void GetRoomDimensions(Document doc, Room room)
{
    String roominfo = "Room dimensions:\n";
}

```



```
// turn on volume calculations:

doc.Settings.VolumeCalculationSetting.VolumeCalculationOptions.VolumeComputationEnable =
                                                                    true;

roominfo += "Vol: " + room.Volume + "\n";
roominfo += "Area: " + room.Area + "\n";
roominfo += "Perimeter: " + room.Perimeter + "\n";
roominfo += "Unbounded height: " + room.UnboundedHeight + "\n";
MessageBox.Show(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.

23.2 Energy Data

The gbXMLParamElem object represents the gbXML Parameters in the Revit project. To view the parameters, from the Revit UI, select Project Information from the Project Settings panel on the Manage tab. The Project Information dialog box appears.

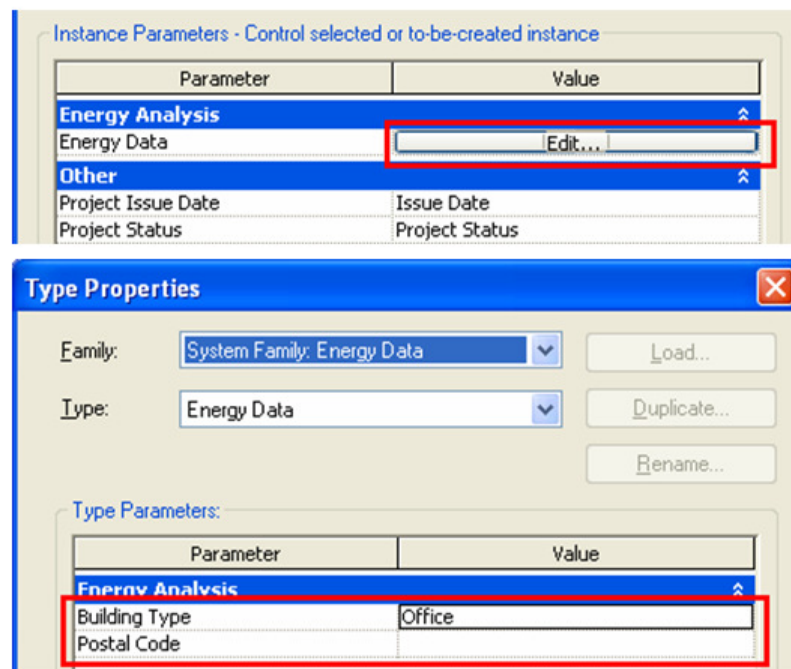


Figure 183: gbXMLParamElem setting

The gbXMLParamElem object is derived from the Element base object. It is unique in each project, similar to ProjectInformation. Though gbXMLParamElem 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. In the Revit English version, Name returns Energy Data with the following Parameters:

- The BuildingType property is used to get or set the Project Information Building Type.
- The ZIPCode property is used to get or set the Project Information ZIP Code.

The following code sample uses the gbXMLParamElem class. The result appears in a message box after invoking the command. The following sample illustrates how to create a new grid with a line.

Code Region 23-7: Using the gbXMLParamElem class

```
public void GetInfo_gbXMLParamElem(Document document)
{
    // The gbXMLParamElem can be retrieved from ProjectInformation only
    gbXMLParamElem xmlPara = document.ProjectInformation.gbXMLSettings;

    if (null != xmlPara)
    {
        string message = "gbXMLParamElem : ";
        message += "\nBuildingType : " + xmlPara.BuildingType;
        message += "\nPostal Code : " + xmlPara.PostalCode;
        MessageBox.Show(message, "Revit", MessageBoxButtons.OK);
    }
}
```

24 Revit Structure

Certain API features that only exist in Revit Structure products are discussed in the following sections:

- Structural Model Elements - Discusses specific Elements and their properties that only exist in the Revit Structure product.
- AnalyticalModel - Discusses analytical model-related classes such as AnalyticalModel, RigidLink, and SupportData.
- Loads - Discusses Load Settings and three kinds of Loads.
- Your Analysis Link - Provides suggestions for API users who want to link the Revit Structure product to certain Structural Analysis applications.

This chapter contains some advanced topics. If you are not familiar with the Revit Platform API, read the basic chapters first, such as [Getting Started](#), [Elements Essentials](#), [Parameter](#), and so on.

24.1 Structural Model Elements

Structural Model Elements are, literally, elements that support a structure such as columns, rebar, trusses, and so on. This section discusses how to manipulate these elements.

24.1.1 Column, Beam, and Brace

Structural column, beam, and brace elements do not have a specific class such as the StructuralColumn class but they are in the FamilyInstance class form.

Note: Though the StructuralColumn, StructuralBeam, and StructuralBrace classes do not exist in the current API, they are used in this chapter to indicate the FamilyInstance objects corresponding to structural columns, beams, and braces.

Though Structural column, beam, and brace are all FamilyInstance objects in the API, they are distinguished by the StructuralType property.

Code Region 24-1: 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;
    }

    MessageBox.Show(message, "Revit");
}

```

You can filter out FamilySymbol objects corresponding to structural columns, beams, and braces in using categories. The category for Structural beams and braces is BuiltInCategory.OST_StructuralFraming.

Code Region 24-2: Using BuiltInCategory.OST_StructuralFraming

```

public void GetBeamAndColumnSymbols(Document document)
{
    FamilySymbolSet columnTypes = new FamilySymbolSet();
    FamilySymbolSet framingTypes = new FamilySymbolSet();
    ElementIterator elemItr = document.get_Elements(typeof(Family));
    elemItr.Reset();
    while (elemItr.MoveNext())
    {
        Family tmpFamily = elemItr.Current as Family;
        Category category = tmpFamily.FamilyCategory;
        if (null != category)
        {
            if ((int)BuiltInCategory.OST_StructuralColumns == category.Id.Value)
            {
                foreach (FamilySymbol tmpSymbol in tmpFamily.Symbols)
                {
                    columnTypes.Insert(tmpSymbol);
                }
            }
            else if ((int)BuiltInCategory.OST_StructuralFraming == category.Id.Value)
            {
                foreach (FamilySymbol tmpSymbol in tmpFamily.Symbols)
                {
                    framingTypes.Insert(tmpSymbol);
                }
            }
        }
    }

    string message = "Column Types: ";
    FamilySymbolSetIterator fsItr = columnTypes.ForwardIterator();
    fsItr.Reset();
    while (fsItr.MoveNext())
    {
        FamilySymbol familySybmol = fsItr.Current as FamilySymbol;
        message += "\n" + familySybmol.Name;
    }
}

```

```

    MessageBox.Show(message, "Revit");
}

```

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.

24.1.2 AreaReinforcement and PathReinforcement:

Find the `AreaReinforcementCurves` for `AreaReinforcement` by iterating the `Curves` property which is an `ElementArray`. In edit mode, `AreaReinforcementCurves` are the purple curves (red when selected) in the Revit UI. From the `Element Properties` dialog box you can see `AreaReinforcementCurve` parameters. Parameters such as Hook Types and Orientation are editable only if the `Override Area Reinforcement Setting` parameter is true.

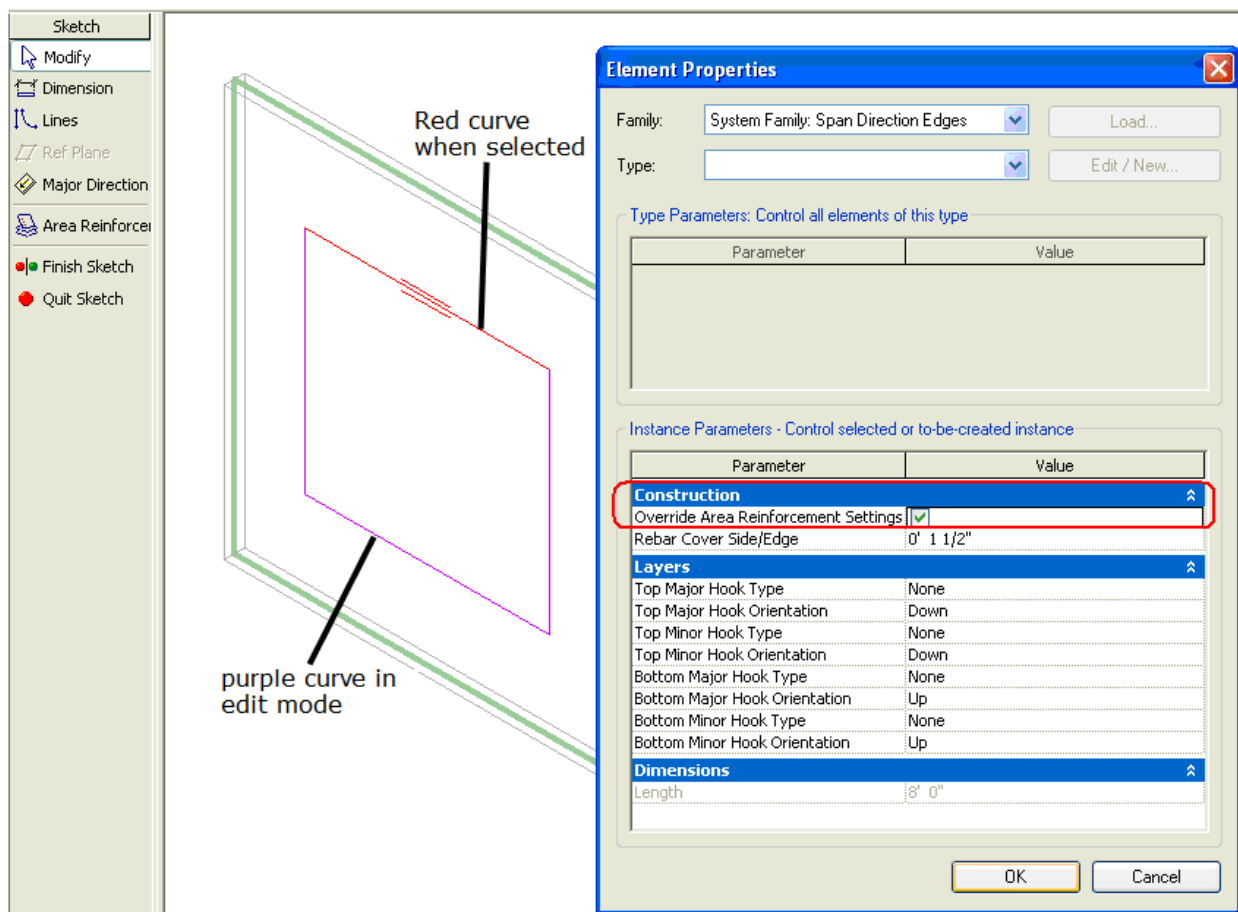


Figure 184: AreaReinforcementCurve in edit mode

There is no way to control the direction except by using the `NewAreaReinforcement()` method (the last XYZ direction input parameter).

Code Region 24-3: NewAreaReinforcement()

```

public AreaReinforcement NewAreaReinforcement(
    Element host, CurveArray curves, XYZ direction);

```

Although the `AreaReinforcement.Curves` property returns a set of `AreaReinforcementCurves`, `PathReinforcementCurves` returns a `ModelCurve`. There is no way to flip the `PathReinforcement` except by using the `NewPathReinforcement()` method (the last input parameter).

Code Region 24-4: `NewPathReinforcement()`

```
public PathReinforcement NewPathReinforcement(
    Element host, CurveArray curves, bool flip);
```

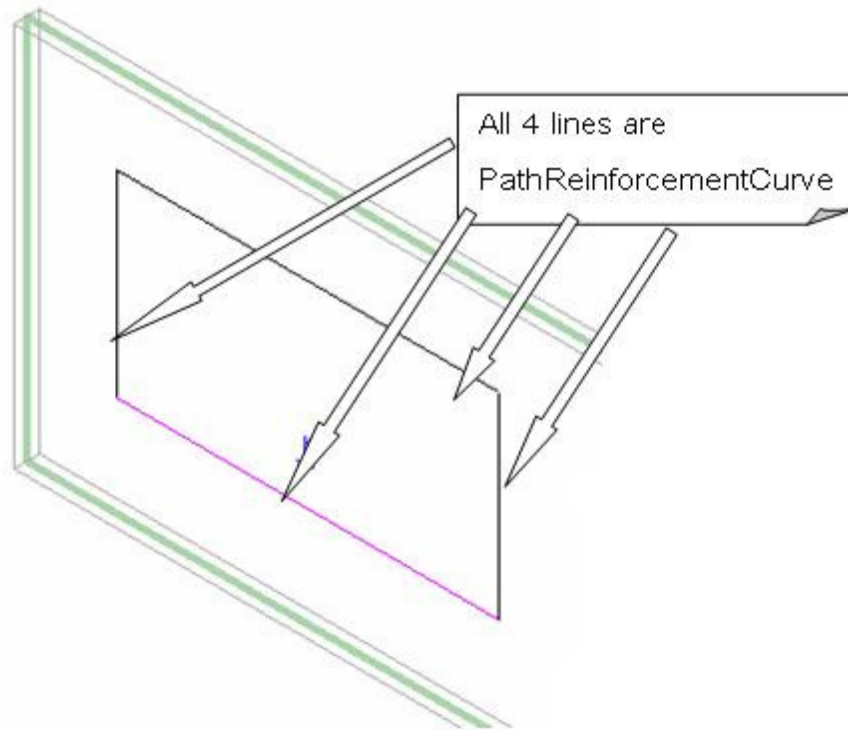


Figure 185: `PathReinforcement.PathReinforcementCurve` in edit mode

When using `NewAreaReinforcement()` and `NewPathReinforcement()` methods to create objects, you must decide on which host `Element` face it will lay. Currently `AreaReinforcement` and `PathReinforcement` are only created on the `PlanarFace` retrieved from the `Wall` or `Floor` object. After removing the faces from the `Wall` or `Floor` geometry, you can filter the `PlanarFace` out as follows:

- Downcast the `Face` to `PlanarFace`:

Code Region 24-5: Downcasting Face to `PlanarFace`

```
PlanarFace planarFace = face as PlanarFace;
```

- If it is a `PlanarFace`, get its `Normal` and `Origin`:

Code Region 24-6: Getting `Normal` and `Origin`

```
private void GetPlanarFaceInfo(Face face)
{
    PlanarFace planarFace = face as PlanarFace;
    if (null != planarFace)
    {
        XYZ origin = planarFace.Origin;
        XYZ normal = planarFace.Normal;
    }
}
```

```

        XYZ vector = planarFace.get_Vector(0);
    }
}

```

- Filter out the right face based on normal and origin. For example:
 - For a general vertical Wall, the face is located using the following factors:
 - The face is vertical; (normal.Z == 0.0)
 - Parallel face directions are opposite:
 - (normal1.X = - normal2.X; normal1.Y = - normal2.Y)
 - Normal must be parallel to the location line.
 - For a general Floor without slope, the factors are:
 - The face is horizontal; (normal.X == 0.0 && normal.Y == 0.0)
 - Judge the top and bottom face; (distinguish 2 faces by normal.Z)

For more details about retrieving an Element's Geometry, refer to the [Geometry](#) chapter.

24.1.3 BeamSystem

BeamSystem provides you with full access and edit ability. You can get and set all of its properties, such as BeamSystemType, BeamType, Direction, and Level. BeamSystem.Direction is not limited to one line of edges. It can be set to any XYZ coordinate on the same plane with the BeamSystem.

Note: You cannot change the StructuralBeam AnalyticalModel after the Elevation property is changed in the UI or by the API. In the following picture the analytical model lines stay in the original location after BeamSystem Elevation is changed to 10 feet.

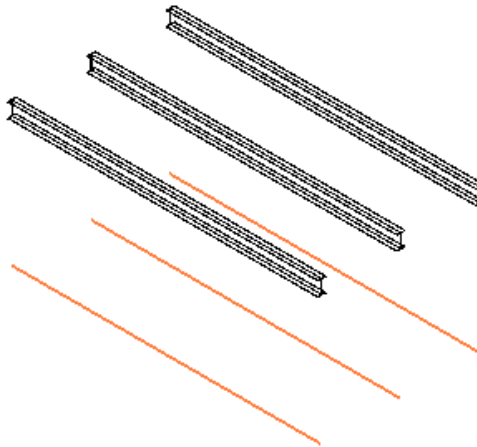


Figure 186: Change BeamSystem elevation

24.1.4 Truss

The Truss class represents all types of trusses in Revit. The TrussType property indicates the type of truss.

Code Region 24-7: Creating a truss over two columns

```

Truss CreateTruss(Autodesk.Revit.Document document, FamilyInstance column1,
FamilyInstance column2)
{
    //sketchPlane

```

```

XYZ origin = new XYZ(0, 0, 0);
XYZ xDirection = new XYZ(1, 0, 0);
XYZ yDirection = new XYZ(0, 1, 0);
Plane plane = document.Application.Create.NewPlane(xDirection, yDirection,
origin);
SketchPlane sketchPlane = document.Create.NewSketchPlane(plane);

//new base Line - use line that spans two selected columns
AnalyticalModelFrame frame1 = column1.AnalyticalModel as AnalyticalModelFrame;
XYZ centerPoint1 = (frame1.Curve as Line).get_EndPoint(0);

AnalyticalModelFrame frame2 = column2.AnalyticalModel as AnalyticalModelFrame;
XYZ centerPoint2 = (frame2.Curve as Line).get_EndPoint(0);
XYZ startPoint = new XYZ(centerPoint1.X, centerPoint1.Y, 0);
XYZ endPoint = new XYZ(centerPoint2.X, centerPoint2.Y, 0);
Autodesk.Revit.Geometry.Line baseLine = null;

try
{
    baseLine = document.Application.Create.NewLineBound(startPoint, endPoint);
}
catch (System.ArgumentException)
{
    throw new Exception("Selected columns are too close to create truss.");
}

// use the active view for where the truss's tag will be placed; View used in
// NewTruss should be plan or elevation view parallel to the truss's base line
Autodesk.Revit.Elements.View view = document.ActiveView;

// Get a truss type for the truss
TrussType trussType = null;
foreach (TrussType currentType in document.TrussTypes)
{
    if (null != currentType)
    {
        trussType = currentType;
        break;
    }
}

Truss truss = null;
if (null != trussType)
{
    truss = document.Create.NewTruss(trussType, sketchPlane, baseLine, view);
}
else
{
    throw new Exception("No truss types found in document.");
}

return truss;
}

```

24.1.5 Rebar

The Rebar class represents rebar used to reinforce suitable elements, such as concrete beams, columns, slabs or foundations.

You can create rebar objects using one of two `Document.Create.NewRebar()` method overloads.

Name	Description
<pre>public Rebar NewRebar(RebarStyle style, RebarBarType rebarType, RebarHookType startHook, RebarHookType endHook, Element host, XYZ norm, CurveArray curves, int startHookOrient, int endHookOrient, bool useExistingShapeIfPossible, bool createNewShape);</pre>	Creates a new instance of a Rebar element within the project.
<pre>public Rebar NewRebar(RebarShape rebarShape, RebarBarType rebarType, Element host, XYZ origin, XYZ xVec, XYZ yVec);</pre>	Creates a new Rebar, as an instance of a RebarShape. The instance will have the default shape parameters from the RebarShape, and its location is based on the bounding box of the shape in the shape definition. Hooks are removed from the shape before computing its bounding box. If appropriate hooks can be found in the document, they will be assigned arbitrarily.

The first version creates rebar from an array of curves describing the rebar, while the second creates a Rebar object based on a RebarShape and position.

When using the first `NewRebar()` method overload, the parameters `RebarBarType` and `RebarHookType` are available in the `RebarBarTypes` and `RebarHookTypes` property. The `RebarBarTypes` property and `RebarHookTypes` property are Document properties.

The following code illustrates how to create Rebar with a specific layout.

Code Region 24-8: Creating rebar with a specific layout

```
Rebar CreateRebar(Autodesk.Revit.Document document, FamilyInstance column, RebarBarType
barType, RebarHookType hookType)
{
    // Define the rebar geometry information - Line rebar
    LocationPoint location = column.Location as LocationPoint;
    XYZ origin = location.Point;
    XYZ normal = new XYZ(1, 0, 0);
    XYZ rebarLineEnd = new XYZ(origin.X, origin.Y, origin.Z + 9);
    Line rebarLine = document.Application.Create.NewLineBound(origin, rebarLineEnd);

    // Create the line rebar
    CurveArray curves = new CurveArray();
    curves.Append(rebarLine);

    Rebar rebar =
        document.Create.NewRebar(Autodesk.Revit.Structural.Enums.RebarStyle.Standard,
                                barType, hookType, hookType, column, origin, curves,
                                RebarHookOrientation.Right, RebarHookOrientation.Left,
                                true, true);
```

```

if (null != rebar)
{
    // set specific layout for new rebar
    Parameter paramLayout = rebar.get_Parameter(BuiltInParameter.REBAR_ELEM_LAYOUT_RULE);
    paramLayout.Set(1); // 1 = Fixed Number
    Parameter parNum = rebar.get_Parameter(BuiltInParameter.REBAR_ELEM_QUANTITY_OF_BARS);
    parNum.Set(10);
    rebar.ArrayLength = 1.5;
}

return rebar;
}

```

Note: For more examples of creating rebar elements, see the Reinforcement and NewRebar sample applications included with the Revit SDK.

The following table lists the integer value for the Parameter REBAR_ELEM_LAYOUT_RULE:

Value	0	1	2	3	4
Description	None	Fixed Number	Maximum Spacing	Number with Spacing	Minimum Clear Spacing

Table 51: Rebar Layout Rule

In the NewRebar() method input parameters, the following rules apply:

- All curves must lie on the same plane as origin.

The Rebar.ScaleToBox() method provides a way to simultaneously set all the shape parameters. The behavior is similar to the UI for placing Rebar.

24.1.5.1 RebarHostData and RebarCoverType

Clear cover is associated with individual faces of valid rebar hosts. You can access the cover settings of a host through the Autodesk.Revit.Elements.RebarHostData object. A simpler, less powerful mechanism for accessing the same settings is provided through parameters.

Cover is defined by a named offset distance, modeled as an element Autodesk.Revit.Symbols.RebarCoverType.

24.1.6 BoundaryConditions

There are three types of BoundaryConditions:

- Point
- Curve
- Area

The type and pertinent geometry information is retrieved using the following code:

Code Region 24-9: Getting boundary condition type and geometry

```

public void GetInfo_BoundaryConditions(BoundaryConditions boundaryConditions)
{
    string message = "BoundaryConditions : ";
}

```

```

Parameter param =
    boundaryConditions.get_Parameter(BuiltInParameter.BOUNDARY_CONDITIONS_TYPE);
switch (param.AsInteger())
{
    case 0:
        XYZ point = boundaryConditions.Point;
        message += "\nThis BoundaryConditions is a Point Boundary Conditions.";
        message += "\nLocation point: (" + point.X + ", "
            + point.Y + ", " + point.Z + ")";
        break;
    case 1:
        message += "\nThis BoundaryConditions is a Line Boundary Conditions.";
        Curve curve = boundaryConditions.get_Curve(0);
        // Get curve start point
        message += "\nLocation Line: start point: (" + curve.get_EndPoint(0).X + ", "
            + curve.get_EndPoint(0).Y + ", " + curve.get_EndPoint(0).Z + ")";
        // Get curve end point
        message += "; end point: (" + curve.get_EndPoint(1).X + ", "
            + curve.get_EndPoint(1).Y + ", " + curve.get_EndPoint(1).Z + ")";
        break;
    case 2:
        message += "\nThis BoundaryConditions is an Area Boundary Conditions.";
        for (int i = 0; i < boundaryConditions.NumCurves; i++)
        {
            Autodesk.Revit.Geometry.Curve areaCurve = boundaryConditions.get_Curve(i);
            // Get curve start point
            message += "\nCurve start point: (" + areaCurve.get_EndPoint(0).X + ", "
                + areaCurve.get_EndPoint(0).Y + ", " + areaCurve.get_EndPoint(0).Z + ")";
            // Get curve end point
            message += "; Curve end point: (" + areaCurve.get_EndPoint(1).X + ", "
                + areaCurve.get_EndPoint(1).Y + ", " + areaCurve.get_EndPoint(1).Z + ")";
        }
        break;
    default:
        break;
}

MessageBox.Show(message, "Revit", MessageBoxButtons.OK);
}

```

24.1.7 Other Structural Elements

Some Element derived classes exist in Revit Architecture and Revit Structure products. In this section, methods specific to Revit Structure are introduced. For more information about these classes, see the corresponding parts in the [Walls, Floors, Roofs and Openings](#) and [Family](#) Instances chapters.

24.1.7.1 Slab

Both Slab (Structural Floor) and Slab Foundation are represented by the Floor class and are distinguished by the IsFoundationSlab property.

The Slab Span Directions are represented by the IndependentTag class in the API and are available as follows:

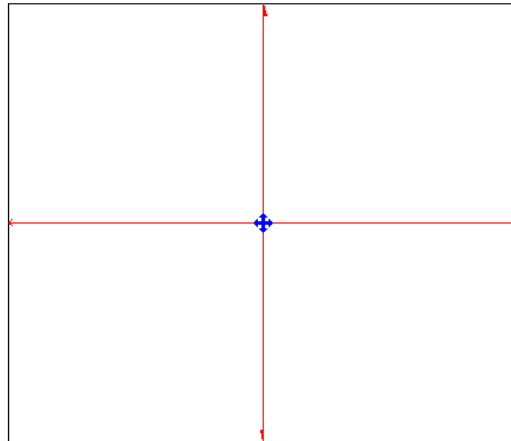


Figure 187: Slab span directions

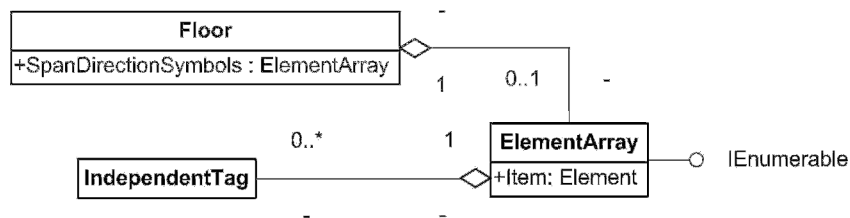


Figure 188: SpanDirectionSymbols diagram

When using NewSlab() to create a Slab, Span Directions are not automatically created. There is also no way to create them directly.

The Slab compound structure layer Structural Deck properties are exposed by the following properties:

- CompoundStructuralLayer.DeckUsage
- DeckProfile

The properties are outlined in the following dialog box:

Edit Assembly

Family: Floor
 Type: 3" LW Concrete on 2" Metal Deck
 Total thickness: 0' 5" (Default)

Layers

	Function	Material	Thickness	Wraps	Variable
1	Core Boundary	Layers Above	0' 0"		
2	Structure [1]	Concrete - Cast	0' 5"	☐	☐
3	Structural Deck	Metal - Deck	0' 0"	☐	☐
4	Core Boundary	Layers Below	0' 0"		

Insert Delete Up Down

Structural Deck Properties

Deck Profile: Form Deck_Non-Composite : 2" x 1" ☐
 Deck Usage: Bound Layer Above ☐

<< Preview OK Cancel Help

Figure 189: Floor CompoundStructuralLayer properties

24.2 Analytical Model

In Revit Structure, an analytical model is the engineering description of a structural physical model. The following structural elements have elemental analytical models:

- Column
- Beam
- Slab
- Brace
- Wall

In the API, an Element object's AnalyticalModel type is determined by the class type, Category, and other properties. Depending on the element's family, the AnalyticalModel may not exist.

If the AnalyticalModel value does not apply to an element's family, the AnalyticalModel property is null. Check the value before using this class. The following table lists the structural elements and AnalyticalModel types in the API:

Object	Element Type	Category (BuiltInCategory)	AnalyticalModel Type
Continuous Wall Foundation	ContFooting	OST_StructuralFoundation	AnalyticalModel3D
Structural Wall	Wall	OST_Walls	AnalyticalModelWall
Slab	Floor	OST_Floors	AnalyticalModelFloor
Beam	StructuralBeam	OST_StructuralFraming	AnalyticalModelFrame
Brace	StructuralBrace	OST_StructuralFraming	AnalyticalModelFrame
Column	StructuralColumn	OST_StructuralColumns	AnalyticalModelFrame
Isolated Footing	FamilyInstance	OST_StructuralFoundation	AnalyticalModelLocation
In-Place Member	FamilyInstance	Corresponding Category	AnalyticalModel3D

Table 52: Elements and Corresponding AnalyticalModel Type

Use the following code to get the proper AnalyticalModel for different elements:

Code Region 24-10: Getting AnalyticalModel

```

public AnalyticalModel GetAnalyticalModel(Element e)
{
    AnalyticalModel analyticalModel = null;
    ContFooting aConFooting = e as ContFooting;
    if (null != aConFooting)
    {
        // get AnalyticalModel3d from ContFooting
        AnalyticalModel3D model3D =
            aConFooting.AnalyticalModel as AnalyticalModel3D;
        analyticalModel = model3D;
    }
    Wall aWall = e as Wall;
    if (null != aWall)
    {
        // get AnalyticalModelWall from Structural Wall
        AnalyticalModelWall modelWall =
            aWall.AnalyticalModel as AnalyticalModelWall;
        analyticalModel = modelWall;
    }
    Floor aFloor = e as Floor;
    if (null != aFloor)
    {
        // get AnalyticalModelFloor from Structural Floor
        AnalyticalModelFloor modelFloor =
            aFloor.AnalyticalModel as AnalyticalModelFloor;
        analyticalModel = modelFloor;
    }
    FamilyInstance familyInst = e as FamilyInstance;
    if (null != familyInst)
    {

```

```

// get AnalyticalModelFrame from structural column, beam and brace
if (familyInst.StructuralType == StructuralType.Beam ||
    familyInst.StructuralType == StructuralType.Brace ||
    familyInst.StructuralType == StructuralType.Column)
{
    AnalyticalModelFrame modelFrame =
        familyInst.AnalyticalModel as AnalyticalModelFrame;
    analyticalModel = modelFrame;
}
// get AnalyticalModelLocation from structural footing
else if (familyInst.StructuralType == StructuralType.Footing)
{
    AnalyticalModelLocation modelLocation =
        familyInst.AnalyticalModel as AnalyticalModelLocation;
    analyticalModel = modelLocation;
}
}
return analyticalModel;
}

```

- AnalyticalModelProfile objects are available in the AnalyticalModelFrame Profile property; you can get it from the AnalyticalModel property directly.
- The In-place member AnalyticalModel property returns an AnalyticalModel3D object.

24.2.1.1 Wall AnalyticalModel

Every overridden NewWall method has a structural Boolean input parameter that indicates if the wall is structural in nature. Check for one of the following conditions:

- The wall family contains AnalyticalModel and the structural input parameter is true.
- WALL_ATTR_EXTERIOR is Foundation.

If either of these conditions applies, downcast the created Wall's AnalyticalModel property to AnalyticalModelWall. In this case, the StructuralUsage property of the wall will be Bearing. Without the AnalyticalModelWall, the StructuralUsage property is NonBearing.

24.2.1.2 Floor AnalyticalModel

If the Floor parameter FLOOR_PARAM_IS_STRUCTURAL is true, the Floor AnalyticalModel is present and the Floor's AnalyticalModel can be downcast to an AnalyticalModelFloor.

Note: Use the Transaction method to get the newly created Element's AnalyticalModel. For example, after you create a wall using the NewWall method, the AnalyticalModel property returns null. Using the Transaction method, you can get both the AnalyticalModel and the wall geometry created by the API. For more details, refer to the [Transaction](#) chapter.

24.2.2 RigidLink

The AnalyticalModelFrame.RigidLink property only works for StructuralBeam when it links to StructuralColumn. The difference between the AnalyticalModelFrame properties Curve and Curves is that Curves includes Curve as well as the StructuralBeam RigidLink Curve if it is present.

Note: You cannot create a rigid link directly since it is not an independent object. You can create it using the manual command.

- StructuralColumn has the rigid link parameter (STRUCTURAL_ANALYTICAL_RIGID_LINK) with the ParameterType, YesNo.
- If this parameter is set to 1 and the StructuralBeam is in the proper location relative to the StructuralColumn, the rigid link is created automatically.

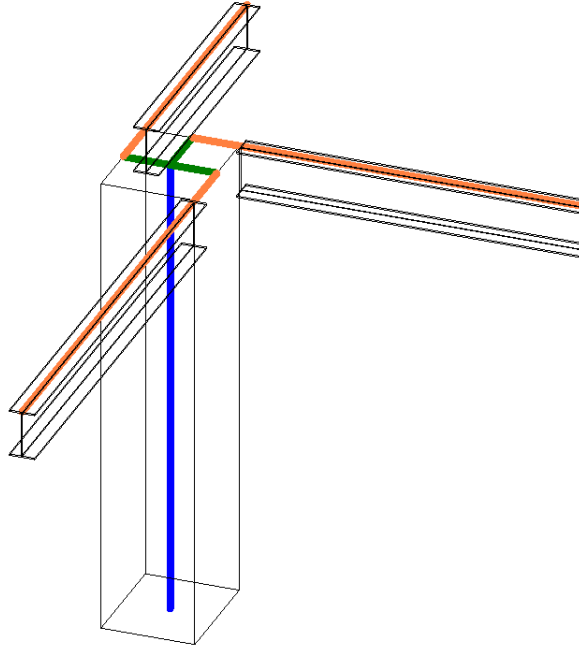


Figure 190: RigidLink

24.2.3 SupportData

SupportData provides information to support other Structural Elements. The following examples illustrate how to use the SupportData property in different conditions.

24.2.3.1 Floor and StructuralBeam SupportData

When drawing a slab in sketch mode, select Pick Supports on the design bar. As shown in the following picture, a slab has three support beams. By iterating the slab SupportData InfoArray, you get the three Beams as well as the Line_Support SupportType.

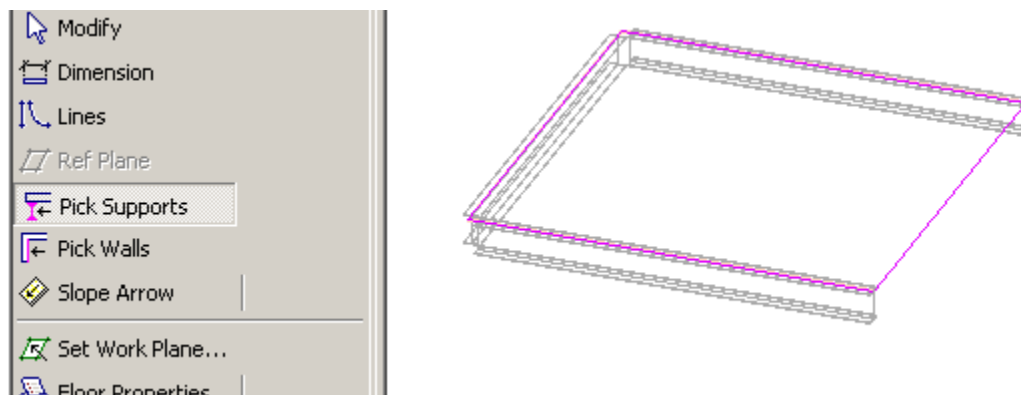


Figure 191: Floor and StructuralBeam SupportData

24.2.3.2 Floor and Wall SupportData

After drawing a slab by picking walls as the support, you cannot get Walls from Floor SupportData. Instead, Floor is available in Wall SupportData. Get the support curve from the AnalyticalModelWall property Curves.

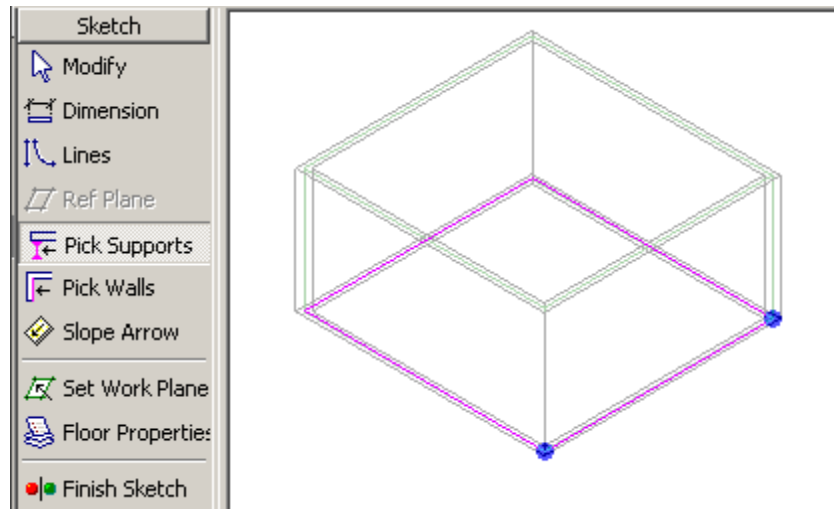


Figure 192: Floor and Wall SupportData

24.2.3.3 Structural Column, Beam and Brace SupportData

In the following picture, the horizontal beam has three Point_Supports--two StructuralColumns and one StructuralBrace. The brace has three Point_Supports-- two StructuralColumns and one StructuralBeam. Neither Column has a support Element.

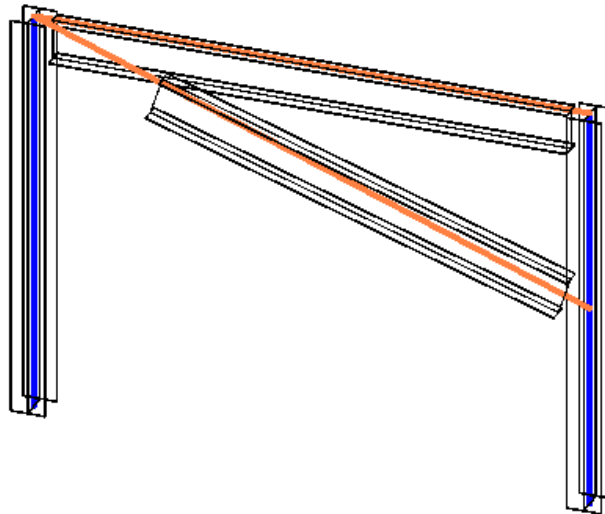


Figure 193: StructuralElements SupportData

24.2.3.4 BeamSystem and Wall SupportData

Though you can pick walls as supports when you draw a BeamSystem, its support information is not directly available because the BeamSystem does not have the AnalyticalModel property. The solution is to call the GetAllBeams method, to retrieve the AnalyticalSupportInfo for the Beams.

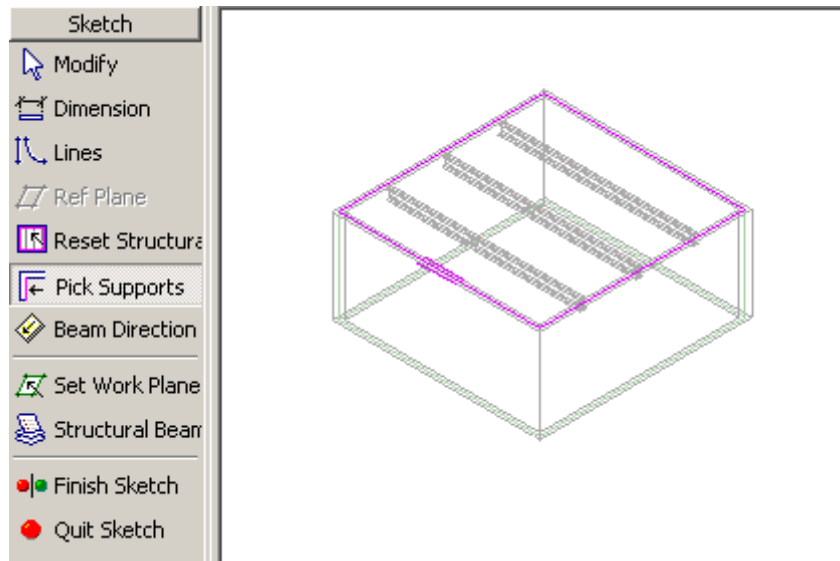


Figure 194: BeamSystem and Wall SupportData

24.2.3.5 ContFooting and Wall SupportData

For a Wall with a continuous Foundation, the Wall has a Line_Support with ContFooting available. The support curves are available using the AnalyticalModel3D.Curves. In the following sample, there are two Arcs in the Curve.

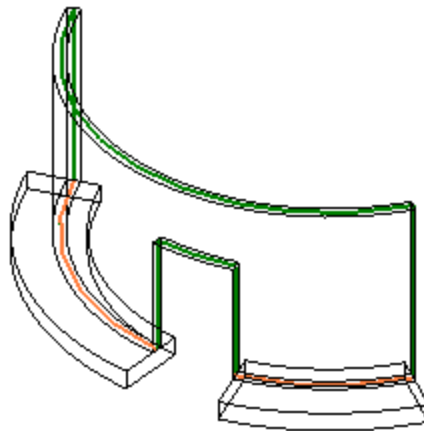


Figure 195: ContFooting and Wall SupportData

24.2.3.6 Isolated Foundation and StructuralColumn SupportData

StructuralColumns can have an Isolated Footing as a Point_Support. In this condition, the Footing can move with the supported StructuralColumn. The FamilyInstance with the OST_StructuralFoundation category is available from InfoArray. Generally, the support point is the bottom point of the AnalyticalModelFrame Curve property. It is also available after you get the FamilyInstance, Isolated Footing, and the AnalyticalModelLocation Point property.

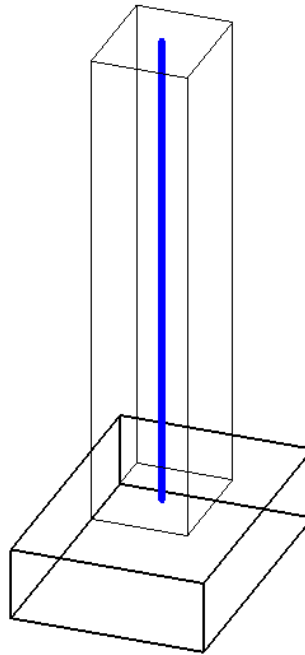


Figure 196: Isolated Foundation (FamilyInstance) and StructuralColumn SupportData

24.3 Loads

The following sections identify load settings and discuss load limitation guidelines.

24.3.1 Load Settings

All functionality on the Setting dialog box Load Cases and Load Combinations tabs can be accessed by the API.

The following properties are available from the corresponding LoadCase BuiltInParameter:

Property	BuiltInParameter
Case Number	LOAD_CASE_NUMBER
Nature	LOAD_CASE_NATURE
Category	LOAD_CASE_CATEGORY

Table 53 Load Case Properties and Parameters

The LOAD_CASE_CATEGORY parameter returns an ElementId. The following table identifies the mapping between Category and ElementId Value.

Load Case Category	BuiltInCategory
Dead Loads	OST_LoadCasesDead
Live Loads	OST_LoadCasesLive
Wind Loads	OST_LoadCasesWind
Snow Loads	OST_LoadCasesSnow
Roof Live Loads	OST_LoadCasesRoofLive
Accidental Loads	OST_LoadCasesAccidental

Load Case Category	BuiltInCategory
Temperature Loads	OST_LoadCasesTemperature
Seismic Loads	OST_LoadCasesSeismic

Table 54: Load Case Category

The following Document creation methods create corresponding subclasses:

- NewLoadUsage() creates LoadUsage
- NewLoadNature() creates LoadNature
- NewLoadCase() creates LoadCase
- NewLoadCombination() creates LoadCombination.
- NewPointLoad() creates PointLoad (an overload allows you to specify the load host element as a Reference). You can optionally specify a load type and sketch plane.
- NewLineLoad() creates LineLoad (overloads allow you to specify the host element as a Reference or an Element). You can optionally specify a load type and sketch plane.
- NewAreaLoad() creates AreaLoad (overloads allow you to specify the host element as a Reference or an Element). You can optionally specify a load type and sketch plane.

Because they are all Element subclasses, they can be deleted using Document.Delete().

Code Region 24-11: NewLoadCombination()

```
public LoadCombination NewLoadCombination(string name,
    int typeInd, int stateInd, double[] factors, LoadCaseArray cases,
    LoadCombinationArray combinations, LoadUsageArray usages);
```

For the NewLoadCombination() method, the factor size must be equal to or greater than the sum size of cases and combinations. For example,

- If cases.Size is M, combinations.Size is N,
- Factors.Size should not be less than M+N. The first M factors map to M cases in order, and the last N factors map to N combinations.
- Check that LoadCombination does not include itself.

There is no Duplicate() method in the LoadCase and LoadNature classes. To implement this functionality, you must first create a new LoadCase (or LoadNature) object using the NewLoadCase() (or NewLoadNature()) method, and then copy the corresponding properties and parameters from an existing LoadCase (or LoadNature).

24.4 Analysis Link

With Revit Structure, an analytical model is automatically generated as you create the physical model. The analytical model is linked to structural analysis applications and the physical model is automatically updated from the results through the Revit Structure API. Some third-party software developers already provide bi-directional links to their structural analysis applications. These include the following:

- ADAPT-Builder Suite from ADAPT Corporation (www.adaptsoft.com/revitstructure/)
- Fastrak and S-Frame from CSC (www.cscworld.com)
- ETABS from CSI (www.csiberkeley.com/)

- RFEM from Dlubal (www.dlubal.com/FEA-Software-RFEM-integrates-with-Revit-Structure.aspx)
- Advance Design, VisualDesign, Arche, Effel and SuperSTRESS from GRAITEC (www.graitec.com/En/revit.asp)
- Scia Engineer from Nemetschek (www.scia-online.com/en/revit.html)
- GSA from Oasys Software (Arup) (www.oasys-software.com/products)
- ProDESK from Prokon Software Consultants (www.prokon.com/)
- RAM Structural System from Bentley (www.bentley.com/en-US/Products/RAM+Structural+System/)
- RISA-3D and RISAFloor from RISA Technologies (www.risatech.com/partner/revit_structure.asp)
- SOFiSTiK Structural Desktop Suite from SOFiSTiK (www.sofistik.com/revit-to-sofistik)
- SPACE GASS from SPACE GASS (www.spacegass.com/index.asp?resend=/revit.asp)
- Revit Structure STAAD.Pro interface from Structural Integrators (structuralintegrators.com/products/si_xchange.php)

The key to linking Revit Structure to other analysis applications is to set up the mapping relationship between the objects in different object models. That means the difficulty and level of the integration depends on the similarity between the two object models.

For example, during the product design process, design a table with at least the first two columns in the object mapping in the following table: one for Revit Structure API and the other for the structural analysis application, shown as follows:

Revit Structural API	Analysis Application	Import to Revit
StructuralColumn	Column	NewStructuralColumn
Property:		
...		
Location		Read-only;
Parameter:		
...		
Analyze as		Editable;
AnalyticalModel:		
...		
Profile		Read-only;
RigidLink		Read-only;
...		
Material:		
...		

Table 55: Revit and Analysis Application Object Mapping

25 Revit MEP

The Revit MEP portion of the Revit API provides read and write access to HVAC and Piping data in a Revit model including:

- Traversing ducts, pipes, fittings, and connectors in a system
- Adding, removing, and changing ducts, pipes, and other equipment
- Getting and setting system properties
- Determining if the system is well-connected

25.1 MEP Element Creation

Elements related to duct and pipe systems can be created using the following methods available in the Autodesk.Revit.Creation.Document class:

- NewDuct
- NewFlexDuct
- NewPipe
- NewFlexPipe
- NewMechanicalSystem
- NewPipingSystem
- NewCrossFitting
- NewElbowFitting
- NewTakeoffFitting
- NewTeeFitting
- NewTransitionFitting
- NewUnionFitting

25.1.1 Creating Pipes and Ducts

There are 3 ways to create new ducts, flex ducts, pipes and flex pipes. They can be created between two points, between two connectors, or between a point and a connector.

The following code creates a new pipe between two points. New flex pipes, ducts and flex ducts can all be created similarly.

Code Region 25-1: Creating a new Pipe

```
public Pipe CreateNewPipe(Document document)
{
    // find a pipe type
    PipeType pipeType = null;
    Filter filter = document.Application.Create.Filter.NewTypeFilter(typeof(PipeType));
    List<Autodesk.Revit.Element> pipeTypes = new List<Autodesk.Revit.Element>();
    document.get_Elements(filter, pipeTypes);
    if (pipeTypes.Count > 0)
    {
        pipeType = pipeTypes[0] as PipeType;
    }
}
```

```

Pipe pipe = null;
if (null != pipeType)
{
    // create pipe between 2 points
    XYZ p1 = new XYZ(0, 0, 0);
    XYZ p2 = new XYZ(10, 0, 0);

    pipe = document.Create.NewPipe(p1, p2, pipeType);
}

return pipe;
}

```

After creating a pipe, you might want to change the diameter. The Diameter property of Pipe is read-only. To change the diameter, get the RBS_PIPE_DIAMETER_PARAM built-in parameter.

Code Region 25-2: Changing pipe diameter

```

public void ChangePipeSize(Pipe pipe)
{
    Parameter parameter = pipe.get_Parameter(BuiltInParameter.RBS_PIPE_DIAMETER_PARAM);

    string message = "Pipe diameter: " + parameter.AsValueString();

    parameter.Set(0.5); // set to 6"

    message += "\nPipe diameter after set: " + parameter.AsValueString();

    MessageBox.Show(message, "Revit");
}

```

Another common way to create a new duct or pipe is between two existing connectors, as the following example demonstrates. In this example, it is assumed that 2 elements with connectors have been selected in Revit MEP, one being a piece of mechanical equipment and the other a duct fitting with a connector that lines up with the SupplyAir connector on the equipment.

Code Region 25-3: Adding a duct between two connectors

```

public Duct CreateDuctBetweenConnectors(Document document)
{
    // prior to running this example
    // select some mechanical equipment with a supply air connector
    // and an elbow duct fitting with a connector in line with that connector
    Connector connector1 = null, connector2 = null;
    ConnectorSetIterator csi = null;
    ElementSet selection = document.Selection.Elements;
    // First find the selected equipment and get the correct connector
    foreach (Element e in selection)
    {
        if (e is FamilyInstance)

```

```

    {
        FamilyInstance fi = e as FamilyInstance;
        Family family = fi.Symbol.Family;
        if (family.FamilyCategory.Name == "Mechanical Equipment")
        {
            csi = fi.MEPModel.ConnectorManager.Connectors.ForwardIterator();
            while (csi.MoveNext())
            {
                Connector conn = csi.Current as Connector;
                if (conn.Direction == FlowDirectionType.Out &&
                    conn.DuctSystemType == DuctSystemType.SupplyAir)
                {
                    connector1 = conn;
                    break;
                }
            }
        }
    }

    // next find the second selected item to connect to
    foreach(Element e in selection)
    {
        if (e is FamilyInstance)
        {
            FamilyInstance fi = e as FamilyInstance;
            Family family = fi.Symbol.Family;
            if (family.FamilyCategory.Name != "Mechanical Equipment")
            {
                csi = fi.MEPModel.ConnectorManager.Connectors.ForwardIterator();
                while (csi.MoveNext())
                if (null == connector2)
                {
                    Connector conn = csi.Current as Connector;

                    // make sure to choose the connector in line with the first connector
                    if (Math.Abs(conn.Origin.Y - connector1.Origin.Y) < 0.001)
                    {
                        connector2 = conn;
                        break;
                    }
                }
            }
        }
    }

    Duct duct = null;
    if (null != connector1 && null != connector2)
    {

```



```

// find a duct type
DuctType ductType = null;
Filter filter = document.Application.Create.Filter.NewTypeFilter(typeof(DuctType));
List<Autodesk.Revit.Element> ductTypes = new List<Autodesk.Revit.Element>();
document.get_Elements(filter, ductTypes);
for (int n=0; n<ductTypes.Count; n++)
{
    ductType = ductTypes[n] as DuctType;
    // make sure it is one of the rectangular duct types
    if (ductType.Name.Contains("Mitered Elbows") == true )
    {
        break;
    }
}

if (null != ductType)
{
    duct = document.Create.NewDuct(connector1, connector2, ductType);
}

return duct;
}

```

Below is the result of running this code after selecting a VAV Unit – Parallel Fan Powered and a rectangular elbow duct fitting.

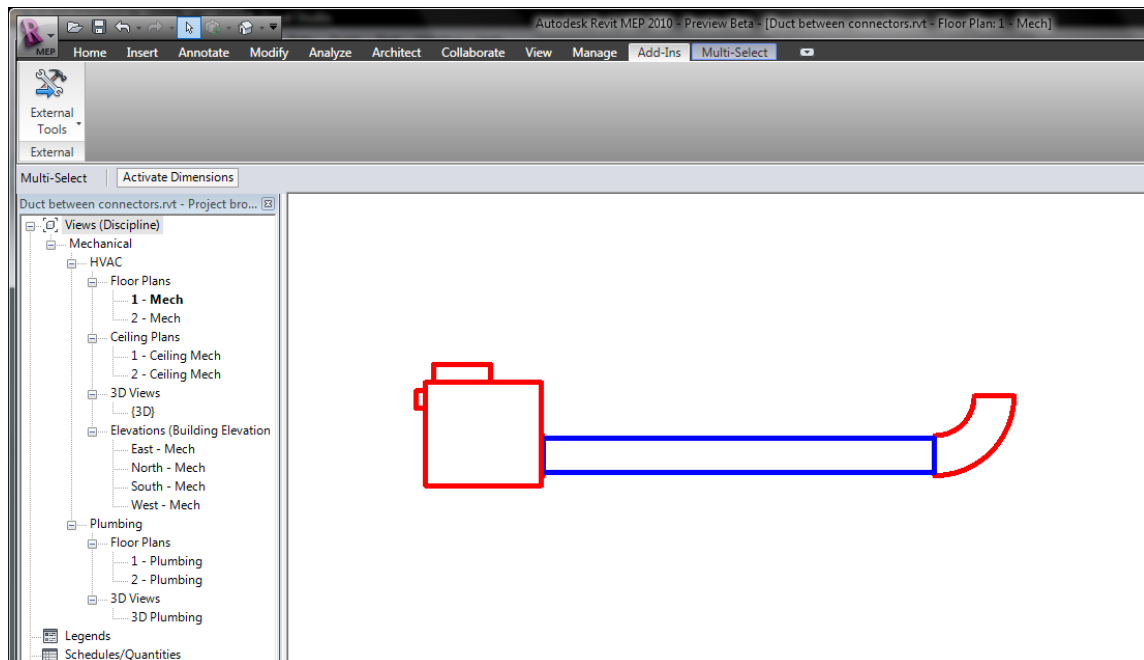


Figure 197: New duct added between selected elements

25.1.2 Creating a new system

New piping and mechanical systems can be created using the Revit API. `NewPipingSystem()` and `NewMechanicalSystem()` both take a `Connector` that is the base equipment connector, such as a hot water heater for a piping system, or a fan for a mechanical system. They also take a `ConnectorSet` of connectors that will be added to the system, such as faucets on sinks in a piping system. The last piece of information required to create a new system is either a `PipeSystemType` for `NewPipingSystem()` or a `DuctSystemType` for `NewMechanicalSystem()`.

In the following sample, a new `SupplyAir` duct system is created from a selected piece of mechanical equipment (such as a fan) and all selected `Air Terminals`.

Code Region 25-4: Creating a new mechanical system

```
// create a connector set for new mechanical system
ConnectorSet connectorSet = new ConnectorSet();
// Base equipment connector
Connector baseConnector = null;

// Select a Parallel Fan Powered VAV and some Supply Diffusers
// prior to running this example
ConnectorSetIterator csi = null;
ElementSet selection = document.Selection.Elements;
foreach (Element e in selection)
{
    if (e is FamilyInstance)
    {
        FamilyInstance fi = e as FamilyInstance;
        Family family = fi.Symbol.Family;
        // Assume the selected Mechanical Equipment is the base equipment for new system
        if (family.FamilyCategory.Name == "Mechanical Equipment")
        {
            //Find the "Out" and "SupplyAir" connector on the base equipment
            if (null != fi.MEPModel)
            {
                csi = fi.MEPModel.ConnectorManager.Connectors.ForwardIterator();
                while (csi.MoveNext())
                {
                    Connector conn = csi.Current as Connector;
                    if (conn.Direction == FlowDirectionType.Out &&
                        conn.DuctSystemType == DuctSystemType.SupplyAir)
                    {
                        baseConnector = conn;
                        break;
                    }
                }
            }
        }
        else if (family.FamilyCategory.Name == "Air Terminals")
        {
            // add selected Air Terminals to connector set for new mechanical system

```

```

        csi = fi.MEPModel.ConnectorManager.Connectors.ForwardIterator();
        csi.MoveNext();
        connectorSet.Insert(csi.Current as Connector);
    }
}

MechanicalSystem mechanicalSys = null;
if (null != baseConnector && connectorSet.Size > 0)
{
    // create a new SupplyAir mechanical system
    mechanicalSys = document.Create.NewMechanicalSystem(baseConnector, connectorSet,
                                                         DuctSystemType.SupplyAir);
}

```

25.2 Connectors

As shown in the previous section, new pipes and ducts can be created between two connectors. Connectors are associated with a domain – ducts, piping or electrical – which is obtained from the Domain property of a Connector. Connectors are present on mechanical equipment as well as on ducts and pipes.

To traverse a system, you can examine connectors on the base equipment of the system and determine what is attached to the connector by checking the IsConnected property and then the AllRefs property. When looking for a physical connection, it is important to check the ConnectionType of the connector. There are both physical and logical connectors in Revit, but only the physical connectors are visible in the application. The following image shows the two types of physical connectors – end connections and curve connectors.

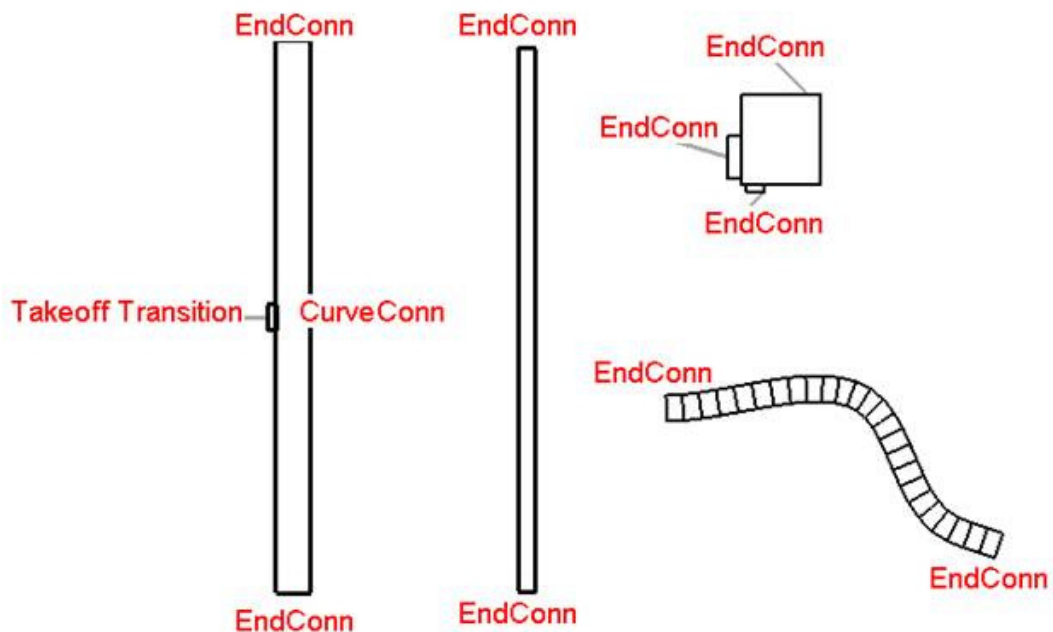


Figure 198: Physical connectors

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 25-5: Determine what is attached to a connector

```
public void GetElementAtConnector(Connector connector)
{
    MEPSystem mepSystem = connector.MEPSSystem;
    if (null != mepSystem)
    {
        string message = "Connector is owned by: " + connector.Owner.Name;

        if (connector.IsConnected == true)
        {
            ConnectorSet connectorSet = connector.AllRefs;
            ConnectorSetIterator csi = connectorSet.ForwardIterator();
            while (csi.MoveNext())
            {
                Connector connected = csi.Current as Connector;
                if (null != connected)
                {
                    // look for physical connections
                    if (connected.ConnectorType == ConnectorType.EndConn ||
                        connected.ConnectorType == ConnectorType.CurveConn ||
                        connected.ConnectorType == ConnectorType.PhysicalConn)
                    {
                        message += "\nConnector is connected to: " + connected.Owner.Name;
                        message += "\nConnection type is: " + connected.ConnectorType;
                    }
                }
            }
        }
        else
        {
            message += "\nConnector is not connected to anything.";
        }

        MessageBox.Show(message, "Revit");
    }
}
```

The following dialog box is the result of running this code example on the connector from a piece of mechanical equipment.

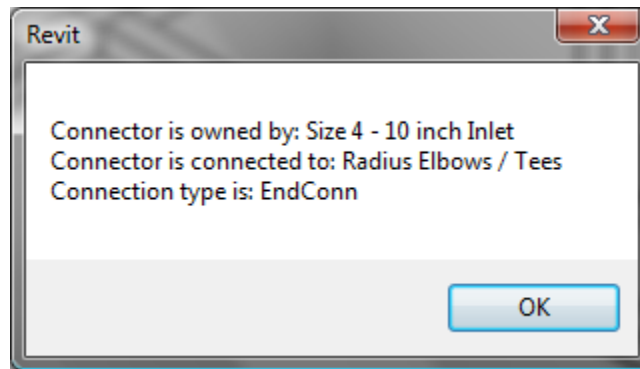


Figure 199: Connector Information

25.3 Family Creation

When creating mechanical equipment in a Revit family document, you will need to add connectors to allow the equipment to connect to a system. Duct, electrical and pipe connectors can all be added similarly, using a reference plane where the connector will be placed and a system type for the connector.

The functions provided in FamilyItemFactory for MEP Connectors are:

- NewDuctConnector
- NewElectricalConnector
- NewPipeConnector

The following code demonstrates how to add two pipe connectors to faces on an extrusion and set some properties on them.

Code Region 25-6: Adding a pipe connector

```
public void CreatePipeConnectors(Document document, Extrusion extrusion)
{
    // get the faces of the extrusion
    Options geoOptions = document.Application.Create.NewGeometryOptions();
    geoOptions.View = document.ActiveView;
    geoOptions.ComputeReferences = true;

    List<PlanarFace> planarFaces = new List<PlanarFace>();
    Autodesk.Revit.Geometry.Element geoElement = extrusion.get_Geometry(geoOptions);
    foreach (GeometryObject geoObject in geoElement.Objects)
    {
        Solid geoSolid = geoObject as Solid;
        if (null != geoSolid)
        {
            foreach (Face geoFace in geoSolid.Faces)
            {
                if (geoFace is PlanarFace)
                {
                    planarFaces.Add(geoFace as PlanarFace);
                }
            }
        }
    }
}
```

```

    }
}
}
if (planarFaces.Count > 1)
{
    // Create the Supply Hydronic pipe connector
    PipeConnector connSupply =
        document.FamilyCreate.NewPipeConnector(planarFaces[0].Reference,
                                                PipeSystemType.SupplyHydronic);

    Parameter param = connSupply.get_Parameter(BuiltInParameter.CONNECTOR_RADIUS);
    param.Set(1.0); // 1' radius
    param = connSupply.get_Parameter(BuiltInParameter.RBS_PIPE_FLOW_DIRECTION_PARAM);
    param.Set(2);

    // Create the Return Hydronic pipe connector
    PipeConnector connReturn =
        document.FamilyCreate.NewPipeConnector(planarFaces[1].Reference,
                                                PipeSystemType.ReturnHydronic);

    param = connReturn.get_Parameter(BuiltInParameter.CONNECTOR_RADIUS);
    param.Set(0.5); // 6" radius
    param = connReturn.get_Parameter(BuiltInParameter.RBS_PIPE_FLOW_DIRECTION_PARAM);
    param.Set(1);
}
}

```

The following illustrates the result of running this example using in a new family document created using a Mechanical Equipment template and passing in an extrusion 2'x2'x1'. Note that the connectors are placed at the centroid of the planar faces.

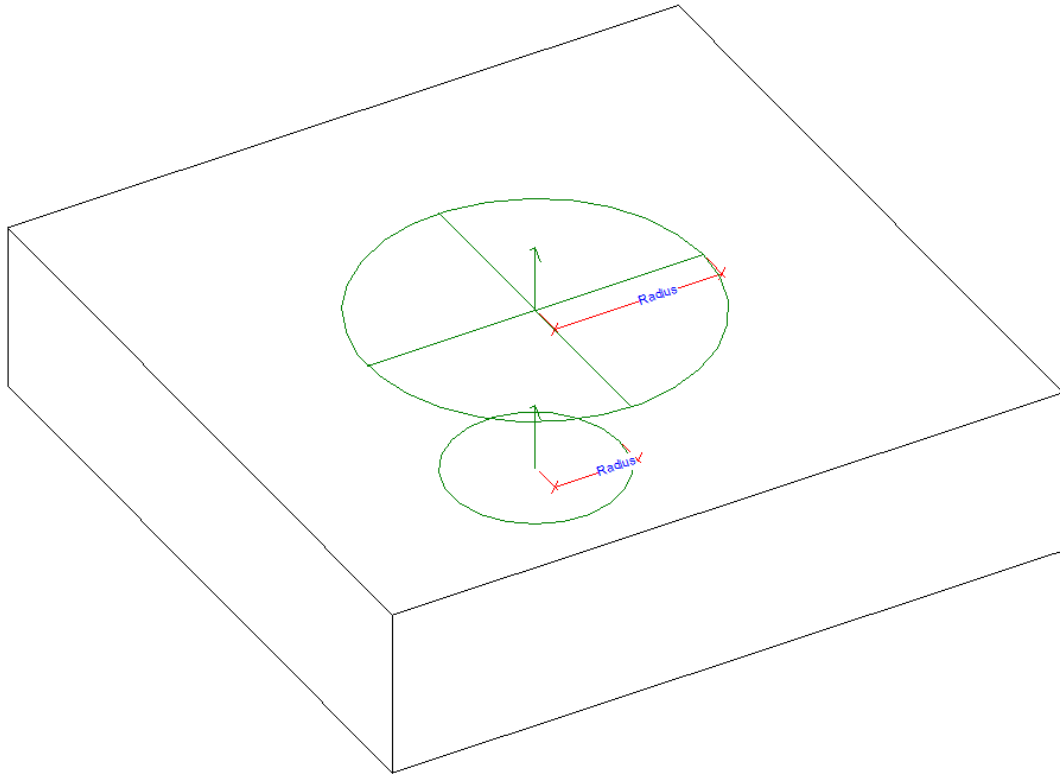


Figure 200: Two connectors created on an extrusion

A. Glossary

Array

Arrays hold a series of data elements, usually of the same size and data type. Individual elements are accessed by their position in the array. The position is provided by an index, which is also called a subscript. The index usually uses a consecutive range of integers, but the index can have any ordinal set of values.

BIM

Building Information Modeling is the creation and use of coordinated, internally consistent, computable information about a building project in design and construction. In a BIM application the graphics are derived from the information and are not the original information itself like in general CAD applications.

Class

In object-oriented programming (OOP), classes are used to group related Properties (variables) and Methods (functions) together. A typical class describes how those methods operate upon and manipulate the properties. Classes can be standalone or inherited from other classes. In the latter, a class from which others are derived is usually referred to as a Base Class.

Events

Events are messages or functions that are called when an event occurs within an application. For example when a model is saved or opened.

Iterator

An iterator is an object that allows a programmer to traverse through all elements in a collection (an array, a set, etc.), regardless of its specific implementation.

Method

A method is a function or procedure that belongs to a class and operates or accesses the class data members. In procedural programming, this is called a function.

Namespace

A namespace is an organizational unit used to group similar and/or functionally related classes together.

Overloading

Method overloading is when different methods (functions) of the same name are invoked with different types and/or numbers of parameters passed.

Properties

Properties are data members of a class accessible to the class user. In procedural programming this is called a variable. Some properties are read only (support Get() method) and some are modifiable (support Set() method).

Revit Families

A Family is a collection of objects called types. A family groups elements with a common set of parameters, identical use, and similar graphical representation. Different types in a family can have different values of some or all parameters, but the set of parameters - their names and their meaning - are the same.

Revit Parameters

There are a number of Revit parameter types.

- Shared Parameters can be thought of as user-defined variables.
- System Parameters are variables that are hard-coded in Revit.
- Family parameters are variables that are defined when a family is created or modified.

Revit Types

A Type is a member of a Family. Each Type has specific parameters that are constant for all instances of the Type that exist in your model; these are called Type Properties. Types have other parameters called Instance parameters, which can vary in your model.

Sets

A set is a collection (container) of values without a particular order and no repeated values. It corresponds with the mathematical concept of set except for the restriction that it has to be finite.

Element ID

Each element has a corresponding ID. It is identified by an integer value. It provides a way of uniquely identifying an Element within an Autodesk Revit project. It is only unique for one project, but not unique across separate Autodesk Revit projects.

Element UID

Each element has a corresponding UID. It is a string identifier that is universally unique. That means it is unique across separate Autodesk Revit projects.

B. FAQ

General Questions

Q: How do I reference an element in Revit?

A: Each element has an ID. The ID that is unique in the model is used to make sure that you are referring to the same element across multiple sessions of Revit.

Q: Can a model only use one shared parameter file?

A: Shared parameter files are used to hold bits of information about the parameter. The most important piece of information is the GUID (Globally Unique Identifier) that is used to insure the uniqueness of a parameter in a single file and across multiple models.

Revit can work with multiple shared parameter files but you can only read parameters from one file at a time. It is then up to you to choose the same shared parameter file for all models or a different one for each model.

In addition, your API application should avoid interfering with the user's parameter file. Ship your application with its own parameter file containing your parameters. To load the parameter(s) into a Revit file:

- The application must remember the user parameter file name.
- Switch to the application's parameter file and load the parameter.
- Then switch back to the user's file.

Q: Do I need to distribute the shared parameters file with the model so other programs can use the shared parameters?

A: No. The shared parameters file is only used to load shared parameters. After they are loaded the file is no longer needed for that model.

Q: Are shared parameter values copied when the corresponding element is copied?

A: Yes. If you have a shared parameter that holds the unique ID for an element in your database, append the Revit element Unique ID or add another shared parameter with the Revit element unique ID. Do this so that you can check it and make sure you are working with the original element ID and not a copy.

Q: Are element Unique IDs (UID) universally unique and can they ever change?

A: The element UIDs are universally unique, but element IDs are only unique within a model. For example, if you copy a wall from one Revit project to another one, the UID of the wall is certain to change to maintain universal uniqueness, but the ID of the wall may not change.

Q: Revit takes a long time to update when my application sends data back to the model. What do I need to do to speed it up?

A: You can try using the `SuspendUpdating` command. See the `FrameBuilder` example in the SDK.

Q: What do I do if I want to add shared parameters to elements that do not have the ability to have shared parameters bound to them? For example, Grids or Materials.

A: If an element type does not have the ability to add shared parameters, you need to add a project parameter. This does make it a bit more complicated when it is time to access the shared parameter associated with the element because it does not show up as part of the element's parameter list. By using tricks like making the project shared parameter a string and including the element ID in the shared parameter you can associate the data with an element by first parsing the string.

Q: How do I access the saved models and content BMP?

A: The Preview.dll is a shell plugin which is an object that implements the IExtractImage interface. IExtractImage is an interface used by the Windows Shell Folders to extract the images for a known file type.

For more information, review the information at <http://windowssdk.msdn.microsoft.com/en-us/library/ms645964.aspx>

CRevitPreviewExtractor implements standard API functions:

```
STDMETHOD(GetLocation)(LPWSTR pszPathBuffer,
                      DWORD cchMax,
                      DWORD *pdwPriority,
                      const SIZE *prgSize,
                      DWORD dwRecClrDepth,
                      DWORD *pdwFlags);

STDMETHOD(Extract)(HBITMAP*);
```

It registers itself in the registry.

Revit Structure Questions

Q: Sometimes the default end releases of structural elements render the model unstable. How can I deal with this situation?

A: The Analytical Model Check feature introduced in Revit Structure R3 can find some of these issues. When importing the analytical model, you are asked if you want to retain the release conditions from RST (Revit Structure) or if you want to set all beams and columns to fixed. When re-importing the model to RST, always update the end releases and do not overwrite the end releases on subsequent export to analysis programs.

Q: I am rotating the beam orientation so they are rotated in the weak direction. For example, the I of a W14X30 is rotated to look like an H by a 90 degree rotation. How is that rotation angle accessed in the API?

Because the location is a LocationCurve not a LocationPoint I do not have access to the Rotation value so what is it I need to check? I have a FamilyInstance element to check so what do I do with it?

A: Take a look at the RotateFramingObject example in the SDK. It has examples of how to get and change the beam braces and columns rotation angle.

Q: How do I add new concrete beam and column sizes to a model?

A: Take a look at the FrameBuilder sample code in the SDK

Q: How do I view the true deck layer?

A: There is an example in the SDK called DeckProperties that provides information about how to get the layer information for the deck. The deck information is reported in exactly the same way as it is in the UI. The deck dimension parameters are shown as follows.

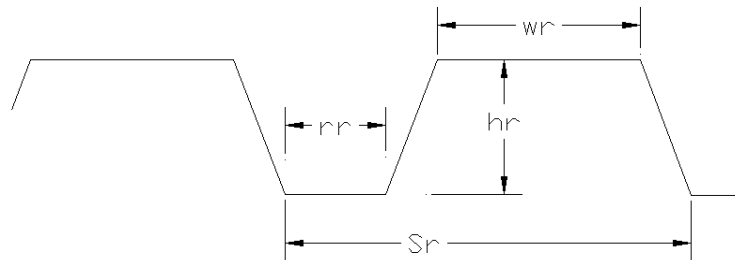


Figure 201: Deck dimension parameters

Q: How do I tell when I have a beam with a cantilever?

A: There is no direct way in the Revit database to tell if a beam has a cantilever. However, one or more of the following options can give you a good guess at whether a section is a cantilever:

4. There are two parameters called Moment Connection Start and Moment Connection End. If the value set for these two is not None then you should look and see if there is a beam that is co-linear and also has the value set to something other than None. Also ask the user to make sure to select Cantilever Moment option rather than Moment Frame option.

Trace the connectivity back beyond the element approximately one or two elements.

Look at element release conditions.

Q: How do I model a foundation with end overhangs under a wall?

A: The wall foundation capability in Revit Structure does not support an overhang at its ends. The best solution is to use the foundation slab feature in the GUI or API. You can create them using the Autodesk.Revit.Create.NewFloor methods. For the STRUCTURAL_FLOOR_ANALYZES_AS, you can use AA_Mat or AA_SlabOnGrade for foundation or AA_Slab if it is a floor.

Q: When exporting a model containing groups to an external program, the user receives the following error at the end of the export:

"Changes to group "Group 1" are allowed only in group edit mode. Use the Edit Group command to make the change to all instances of the group. You may use the "Ungroup" option to proceed with this change by ungrouping the changed group instances."

A: Currently the API does not permit changes to group members. You can programmatically ungroup, make the change, regroup and then swap the other instances of the old group to the new group to get the same effect.

C. Hello World for VB.NET

Directions for creating the sample application for Visual Basic .NET are available in the following sections. The sample application was created using Microsoft Visual Studio.

Create a New Project

The first step in writing a VB.NET program with Visual Studio is to choose a project type and create a new project.

1. From the File menu, select New> Project....
2. In the Project types frame, click Visual Basic.
3. In the Templates frame, click Class Library. The application assumes that your project location is: D:\Sample.
4. In the Name field, type HelloWorld as the project name.
5. Click OK.

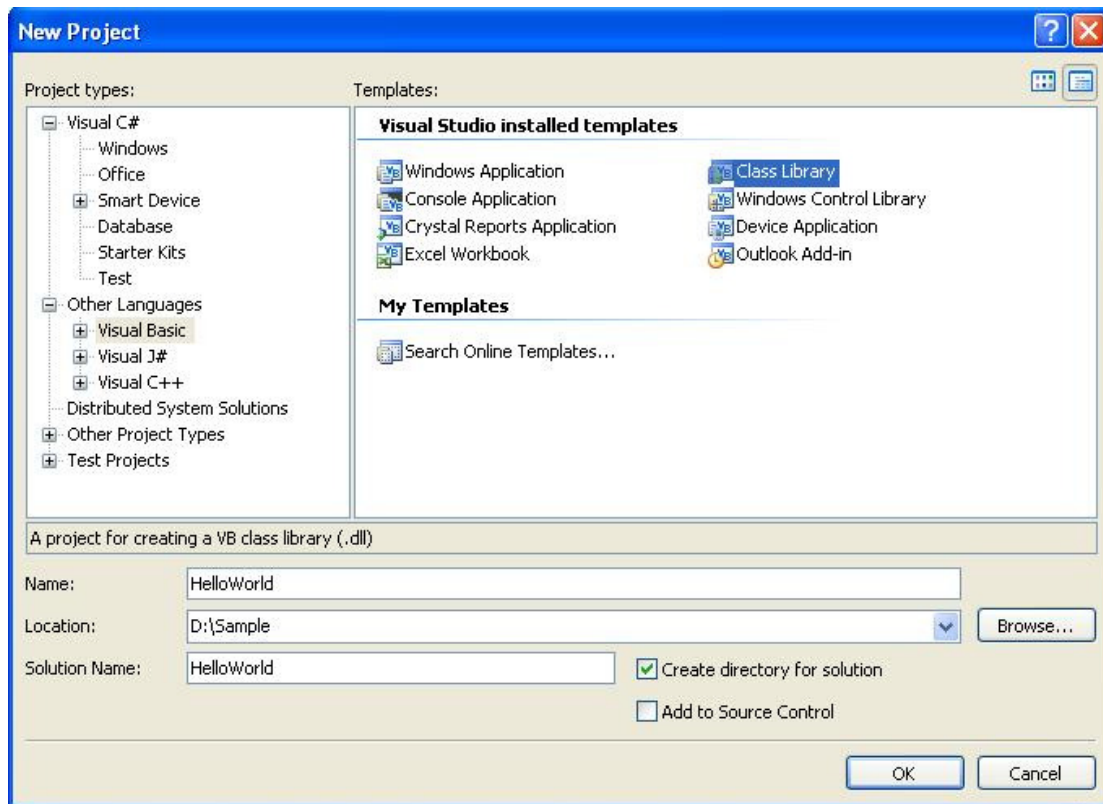


Figure 202: New Project dialog box

Add Reference and Namespace

VB.NET uses a process similar to C#. After you create the Hello World project, complete the following steps:

1. Right-click the project name in the Solution Explorer to display a context menu.
2. From the context menu, select Properties to open the Properties dialog box.
3. In the Properties dialog box, click the References tab. A list of references and namespaces appears.

4. Click the Add button to open the Add Reference dialog box.
5. In the Add Reference dialog box, click the Browse tab. Locate the folder where Revit is installed and click the RevitAPI.dll. For example the installed folder location might be C:\Program Files\Autodesk Revit Architecture 2010\Program\RevitAPI.dll.
6. Click OK to add the reference and close the dialog box.

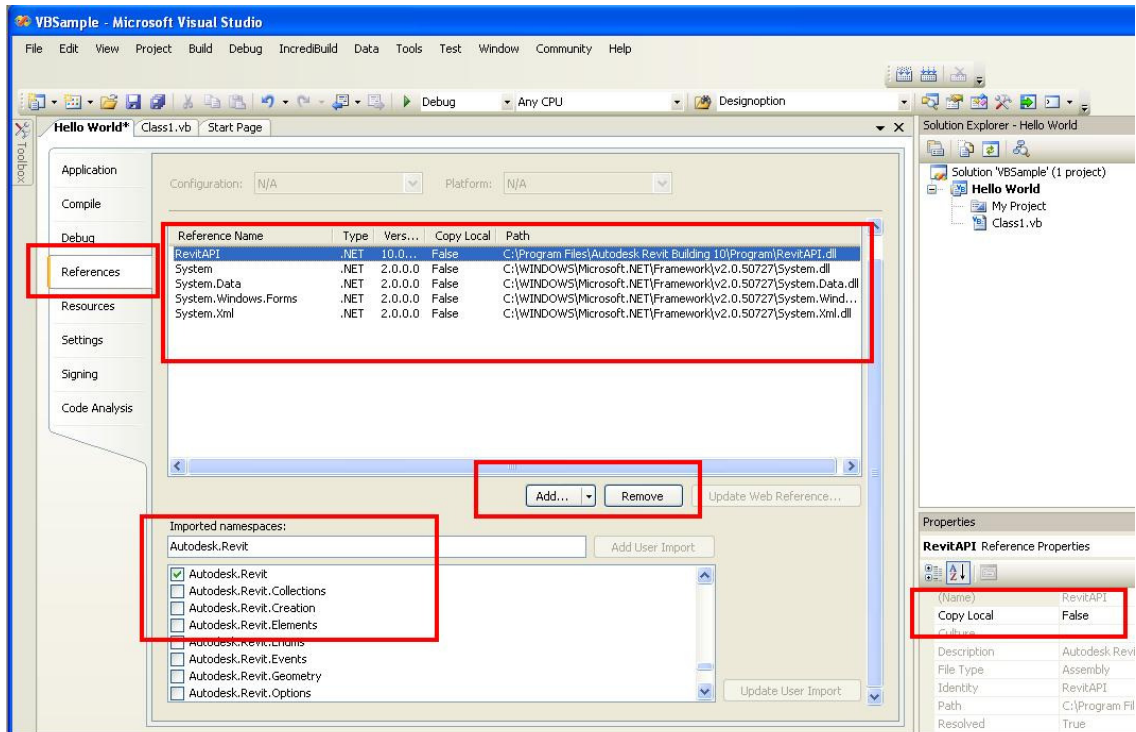


Figure 203: Add references and import Namespaces

After adding the reference, you must import the namespaces used in the project. For this example, import the Autodesk.Revit namespace

To complete the process, click RevitAPI in the Reference frame to highlight it. Set Copy Local to False in the property frame.

Change the Class Name

To change the class name, complete the following steps:

1. In the Solution Explorer, right-click Class1.vb to display a context menu.
2. From the context menu, select Rename. Rename the file HelloWorld.vb.
3. In the Solution Explorer, double-click HelloWorld.vb to open it for editing.

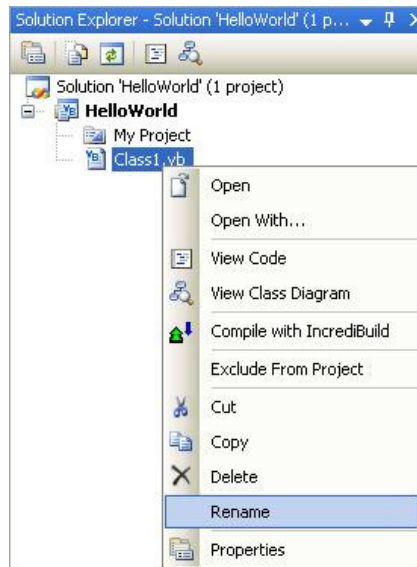


Figure 204: Change the class name

Add Code

When writing the code in VB.NET, you must pay attention to key letter capitalization.

Code Region 25-7: Hello World in VB.NET

```
Imports System
Imports System.Windows.Forms

Imports Autodesk.Revit

Public Class HelloWorld
    Implements Autodesk.Revit.IExternalCommand

    Public Function Execute(ByVal commandData As Autodesk.Revit.ExternalCommandData, _
        ByRef message As String, ByVal elements As Autodesk.Revit.ElementSet) As Autodesk.Revit.IExternalCommand.Result _
        Implements Autodesk.Revit.IExternalCommand.Execute

        MsgBox("Hello World")
        Return IExternalCommand.Result.Succeeded

    End Function
End Class
```

Modify the Revit.ini File

After you add the code, you must build the file. The HelloWorld.dll file appears in the project output directory. If you want to invoke the application in Revit, modify the Revit.ini file to register it.

1. To edit the Revit.ini file, open it for editing in Notepad. The Revit.ini file is usually located in the Revit installation directory on your computer. For example: C:\Program Files\Autodesk Revit Structure 2010\Program.
2. Add the following to the end of the existing code:

Code Region 25-8: Revit.ini ExternalCommands Section

```
[ExternalCommands]
ECCount=1
ECClassName1= HelloWorld.HelloWorld
ECAssembly1= D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll
ECName1= HelloWorld
ECDescription1=Implementation of HelloWorld within Autodesk Revit
```

Note: ECAssembly1 is the path to the assembly, D:\Sample\HelloWorld\bin\Debug\HelloWorld.dll in this particular example.

Refer to the [Add-In Integration](#) chapter for more details about the Revit.ini file.

Build the Program

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

Debug the Program

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

1. In the Solution Explorer window, right-click the HelloWorld project to display a context menu.
2. From the context menu, click Properties. The Properties window appears.
3. Click the Debug tab.
4. In the Debug window Start Action section, click Start external program and browse to the Revit.exe file. By default, the file is located at the following path, C:\Program Files\Autodesk Revit Structure 2010\Program\Revit.exe.

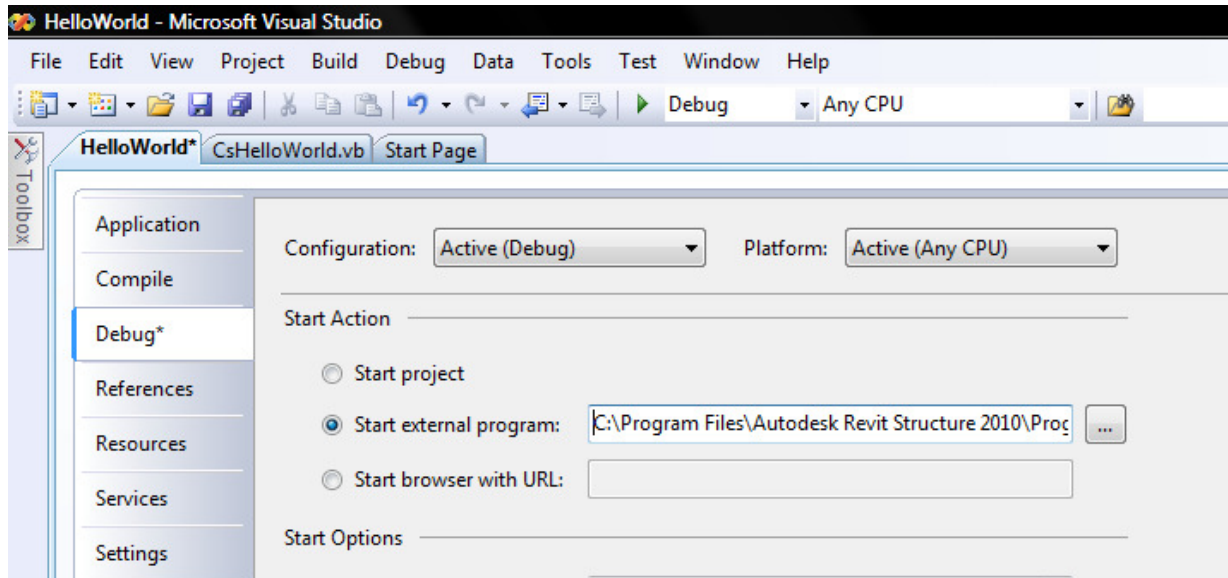


Figure 205: Set Debug environment

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

Code Region 25-9: MsgBox("Hello World")

```
MsgBox ("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.

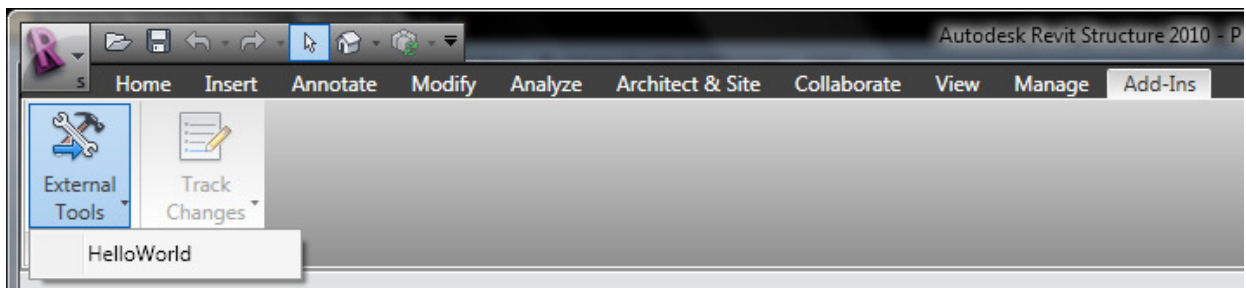


Figure 206: HelloWorld External Tools command

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



Figure 207: System message

D. Material Properties Internal Units

Parameter Name in Dialog	API Parameter Name	Variable Type	Internal Database Units	Description
Steel / Generic				
Behavior	Behavior	bool	Isotropic, Orthotropic	Currently exposed for Generic Only
Young Modulus X	YoungModulusX	double	UT_Stress	
Young Modulus Y	YoungModulusY	double	UT_Stress	
Young Modulus Z	YoungModulusZ	double	UT_Stress	
Poisson ratio X	PoissonModulusX	double	UT_Number	
Poisson ratio Y	PoissonModulusY	double	UT_Number	
Poisson ratio Z	PoissonModulusZ	double	UT_Number	
Shear modulus X	ShearModulusX	double	UT_Stress	
Shear modulus Y	ShearModulusY	double	UT_Stress	
Shear modulus Z	ShearModulusZ	double	UT_Stress	
Thermal Expansion coefficient X	ThermalExpansionCoefficientX	double	UT_TemperaExp	
Thermal Expansion coefficient Y	ThermalExpansionCoefficientY	double	UT_TemperaExp	
Thermal Expansion coefficient Z	ThermalExpansionCoefficientZ	double	UT_TemperaExp	
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

Material Properties Internal Units

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_TemperaExp	
Thermal Expansion coefficient Y	ThermalExpansionCoefficientY	double	UT_TemperaExp	
Thermal Expansion coefficient Z	ThermalExpansionCoefficientZ	double	UT_TemperaExp	
Unit Weight	UnitWeight	double	UT_UnitWeight	
Damping ratio	DampingRatio	double	UT_Number	
Concrete compression	ConcreteCompression	double	UT_Stress	Concrete compression stress capacity. F'c for US codes.
Lightweight	LightWeight	Bool		If true then lightweight concrete is defined.
Shear strength modification	ShearStrengthReduction	double	UT_Number	When Lightweight = True then this value is available. It is the reduction of the concrete compression capacity for shear. Concrete Shear stress Capacity = ConcreteCompression / ShearStrengthReduction
Wood				
Young Modulus	PoissonModulus	double	UT_Stress	
Poisson ratio	ShearModulus	double	UT_Number	
Shear modulus	ShearModulus	double	UT_Stress	
Thermal Expansion coefficient	ThermalExpansionCoefficient	double	UT_TemperaExp	

Material Properties Internal Units

Unit Weight	UnitWeight	double	UT_UnitWeight	
Species	Species	AString		
Grade	Grade	AString		
Bending	Bending	double	UT_Stress	
Compression parallel to grain	CompressionParallel	double	UT_Stress	
Compression perpendicular to grain	CompressionPerpendicular	double	UT_Stress	
Shear parallel to grain	ShearParallel	double	UT_Stress	
Tension perpendicular to grain	ShearPerpendicular	double	UT_Stress	

#	Unit	Dimension	Internal Representation
1.	UT_Number	No dimension	Simple number.
2.	UT_Stress	$(\text{Mass}) \times (\text{Length}^{-1}) \times (\text{Time}^{-2})$	$\text{Kg}(\text{mass})/(\text{Foot} \times \text{Sec}^2)$
3.	UT_TemporalExp	$(\text{Temperature}^{-1})$	$(1/^\circ\text{C})$
4.	UT_UnitWeight	$(\text{Mass}) \times (\text{Length}^{-2}) \times (\text{Time}^{-2})$	$\text{Kg}(\text{mass})/(\text{Foot}^2 \times \text{Sec}^2)$

E. Concrete Section Definitions

Concrete-Rectangular Beam

Family Types

Name: 12 x 24

Parameter	Value	Formula
Materials and Finishes		
Beam Material (default)	Concrete - Cast-in-Place Concrete	=
Structural		
Angle (default)	0.000°	=
Dimensions		
Length (default)	5' 0"	=
b	1' 0"	=
h	2' 0"	=
Identity Data		
Assembly Code	B10	=
Keynote		=
Model		=
Manufacturer		=
Type Comments		=
URL		=
Description		=
Cost		=

Family Types

New...

Rename...

Delete

Parameters

Add...

Modify...

Remove

OK Cancel Apply Help

Figure 208: Concrete-Regangular Beam Parameters

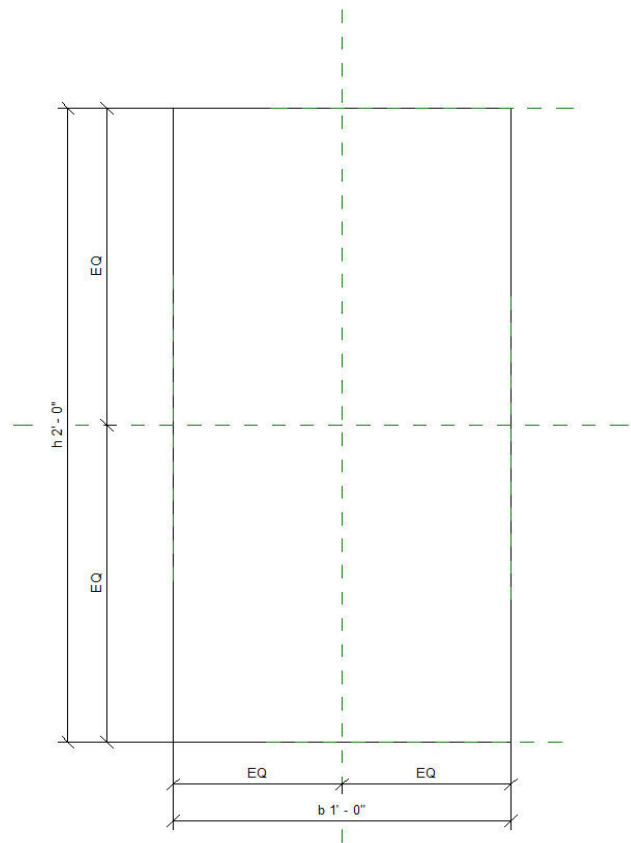


Figure 209: Concrete-Rectangular Beam Cross Section

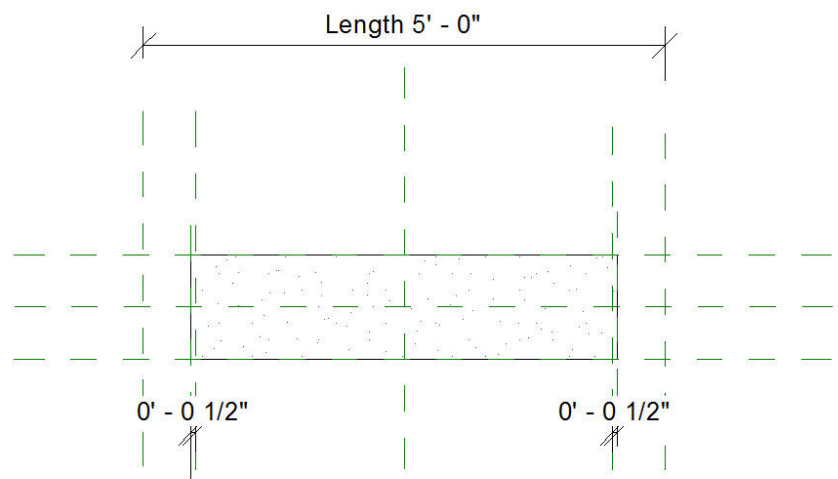


Figure 210: Concrete-Rectangular Beam

Precast-Rectangular Beam

Family Types [X]

Name: 12RB24

Parameter	Value	Formula
Construction		
Start Extension (default)	-0' 0 1/2"	=
End Extension (default)	-0' 0 1/2"	=
Materials and Finishes		
Beam Material (default)	Concrete - Precast Concrete - Normal W	=
Structural		
Angle (default)	0.000°	=
Dimensions		
Length (default)	5' 0"	=
b	1' 0"	=
h	2' 0"	=
Identity Data		
Assembly Code	B10	=
Keynote		=
Model		=
Manufacturer		=
Type Comments		=
URL		=
Description		=
Cost		=
Other		
Start Extension Calculation (default)	10' 0"	= 10' 0 1/2" + Start Ext
End Extension Calculation (default)	10' 0"	= 10' 0 1/2" + End Ext

Family Types: New... Rename... Delete

Parameters: Add... Modify... Remove

OK Cancel Apply Help

Figure 211: Precast-Rectangular Beam Properties

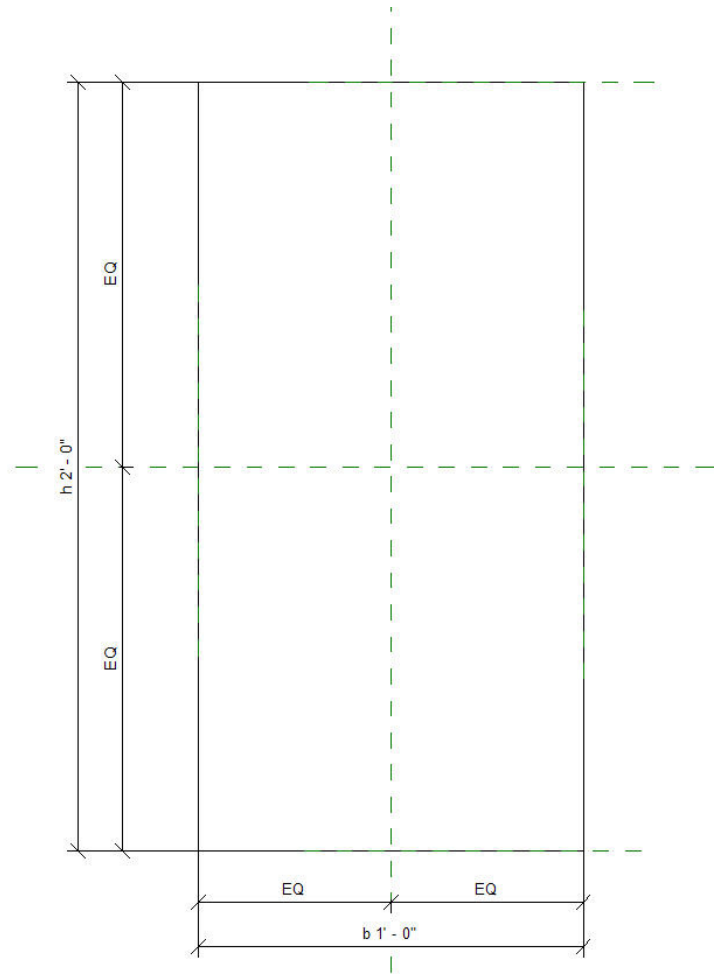


Figure 212: Precast-Rectangular Beam Cross Section

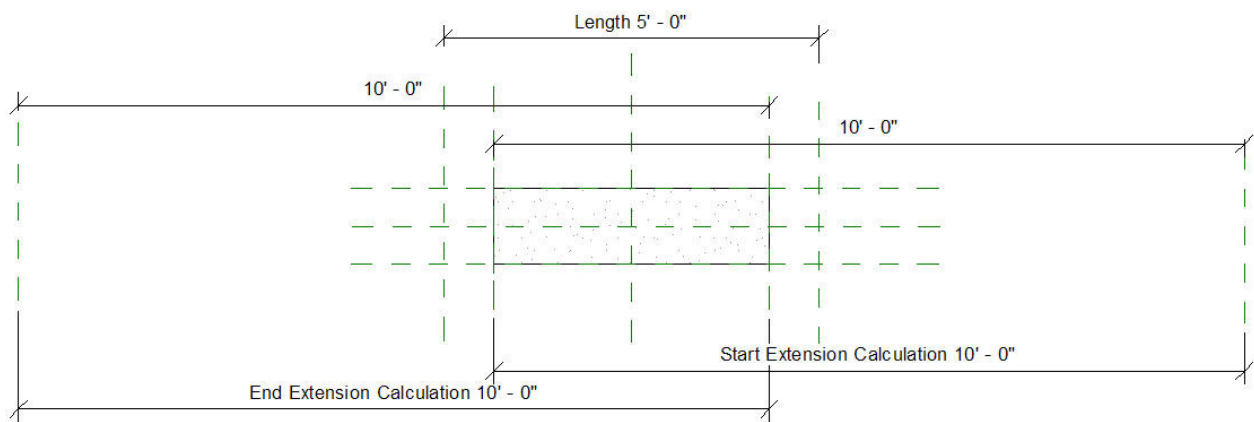


Figure 213: Precast-Rectangular Beam

Precast-L Shaped Beam

Family Types

Name: 18LB24

Parameter	Value	Formula
Construction		
Start Extension (default)	-0' 0 1/2"	=
End Extension (default)	-0' 0 1/2"	=
Materials and Finishes		
Beam Material (default)	Concrete - Precast Concrete - Normal W	=
Structural		
Angle (default)	0.000°	=
Dimensions		
Seat	0' 6"	=
Length (default)	5' 0"	=
b	1' 6"	=
h	2' 0"	=
h1	1' 2"	=
Identity Data		
Assembly Code	B10	=
Keynote		=
Model		=
Manufacturer		=
Type Comments		=
URL		=
Description		=
Cost		=
Other		
Start Extension Calculation (default)	10' 0"	= 10' 0 1/2" + Start Ext
End Extension Calculation (default)	10' 0"	= 10' 0 1/2" + End Ext

Family Types: New..., Rename..., Delete

Parameters: Add..., Modify..., Remove

OK Cancel Apply Help

Figure 214: Precast-L Shaped Beam Properties

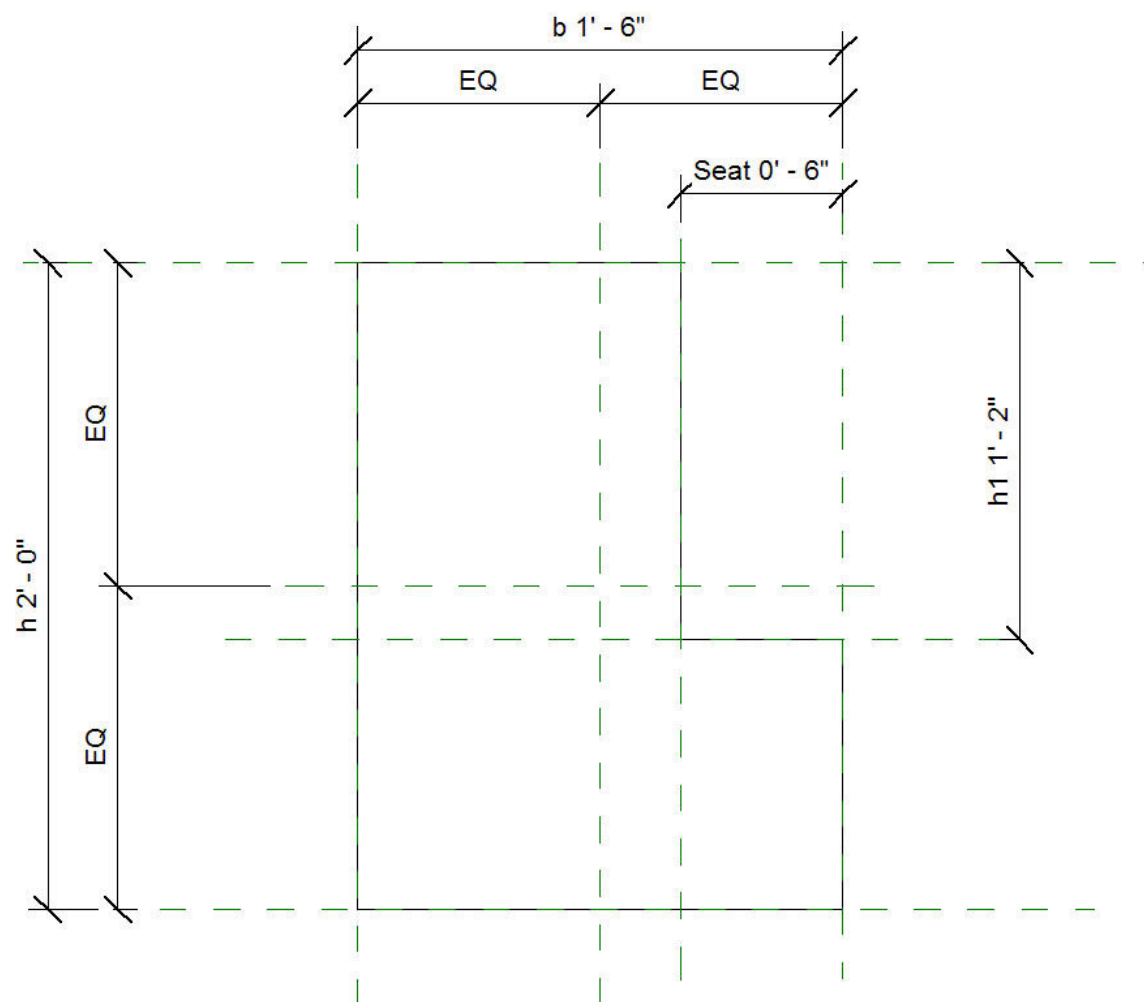


Figure 215: Precast-L Shaped Beam Cross Section

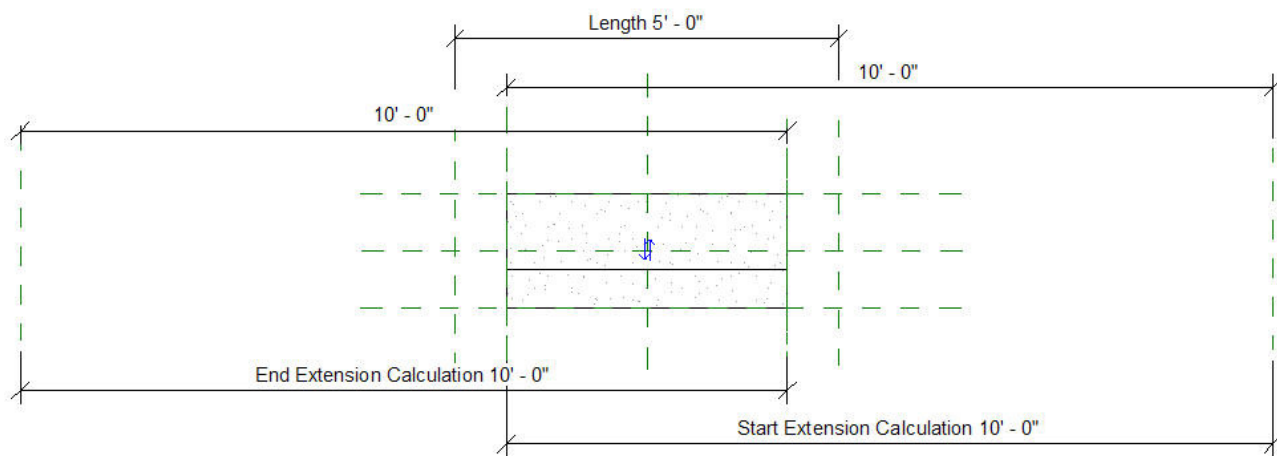


Figure 216: Precast-L Shaped Beam

Precast-Single Tee

Family Types

Name: 8' x 18"

Parameter	Value	Formula
Construction		
Start Extension (default)	-0' 0 1/2"	=
End Extension (default)	-0' 0 1/2"	=
Materials and Finishes		
Beam Material (default)	Concrete - Precast Concrete - Normal W	=
Structural		
Angle (default)	0.000°	=
Dimensions		
Depth	1' 6"	=
Length (default)	5' 0"	=
Slab Depth	0' 2"	=
Stem Width	0' 3 3/4"	=
Width	8' 0"	=
Identity Data		
Assembly Code	B10	=
Keynote		=
Model		=
Manufacturer		=
Type Comments		=
URL		=
Description		=
Cost		=
Other		
Start Extension Calculation (default)	10' 0"	= 10' 0 1/2" + Start Ext
End Extension Calculation (default)	10' 0"	= 10' 0 1/2" + End Ext

Family Types

New...
Rename...
Delete

Parameters

Add...
Modify...
Remove

OK

Cancel

Apply

Help

Figure 217: Precast-Single Tee Properties

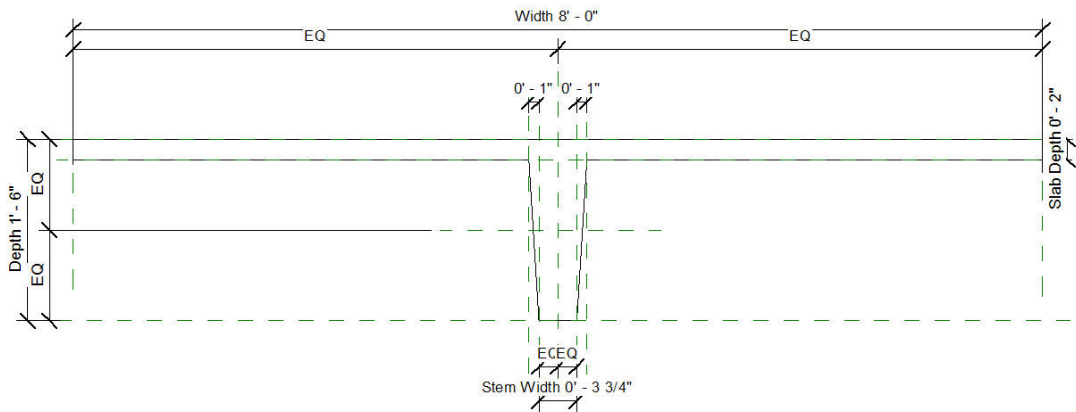


Figure 218: Precast-Single Tee Cross Section

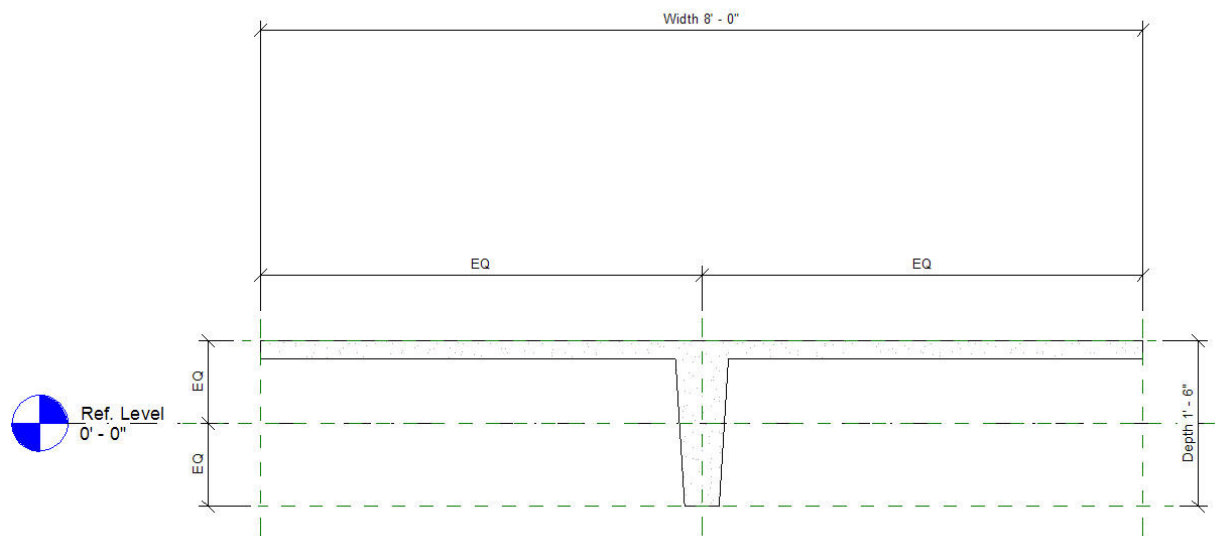


Figure 219: Precast-Single Tee Cross Section 2

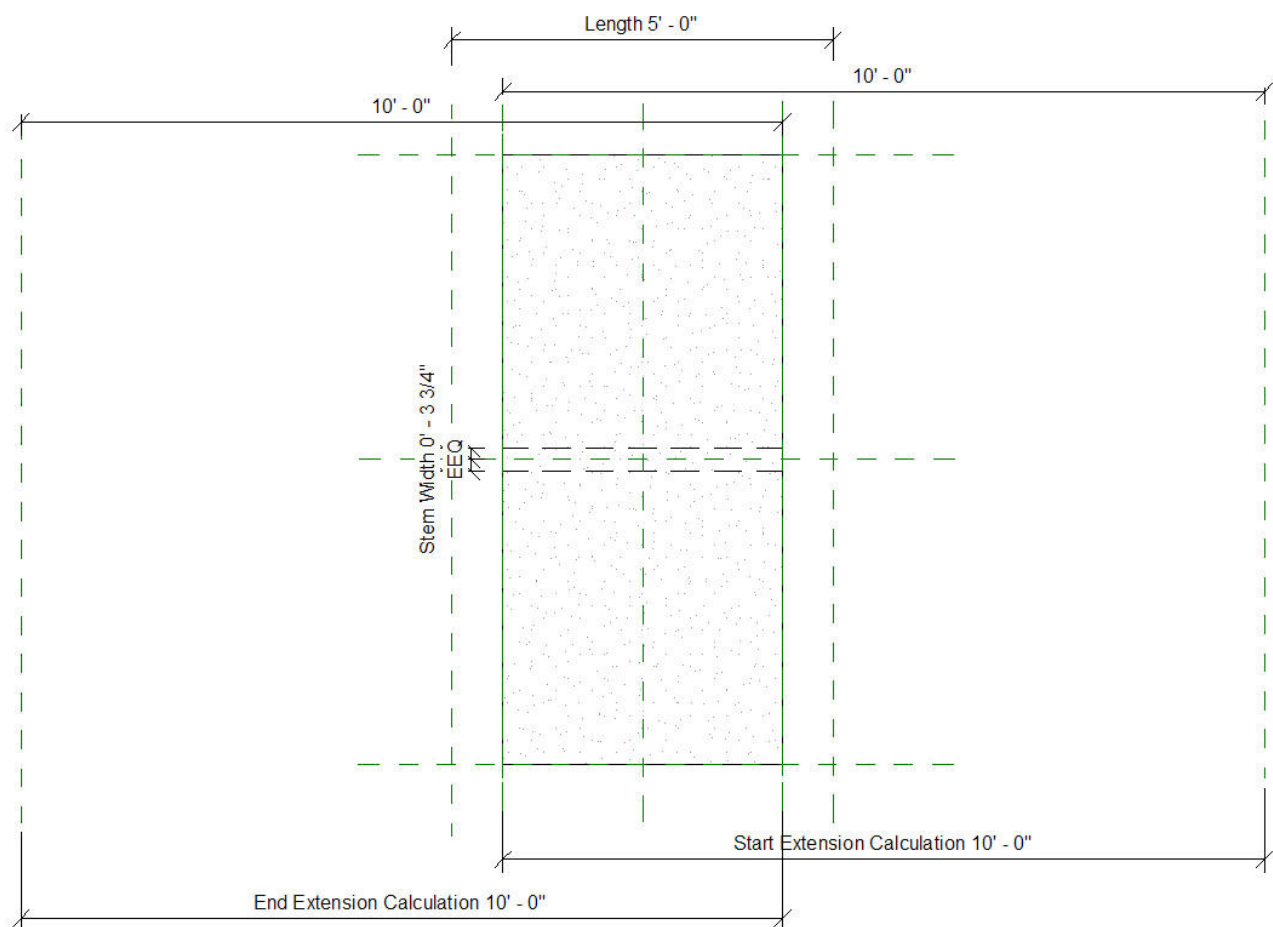


Figure 220: Precast-Single Tee

Precast-Inverted Tee

Family Types X

Name:

Parameter	Value	Formula
Construction		
Start Extension (default)	-0' 0 1/2"	=
End Extension (default)	-0' 0 1/2"	=
Materials and Finishes		
Beam Material (default)	Concrete - Precast Concrete - Normal W	=
Structural		
Angle (default)	0.000°	=
Dimensions		
Seat	0' 6"	=
Length (default)	5' 0"	=
b	2' 0"	=
h	2' 4"	=
h1	1' 4"	=
Identity Data		
Assembly Code	B10	=
Keynote		=
Model		=
Manufacturer		=
Type Comments		=
URL		=
Description		=
Cost		=
Other		
Start Extension Calculation (default)	10' 0"	=10' 0 1/2" + Start Ext
End Extension Calculation (default)	10' 0"	=10' 0 1/2" + End Ext

Family Types

New...

Rename...

Delete

Parameters

Add...

Modify...

Remove

OK Cancel Apply Help

Figure 221: Precast-Inverted Tee Properties

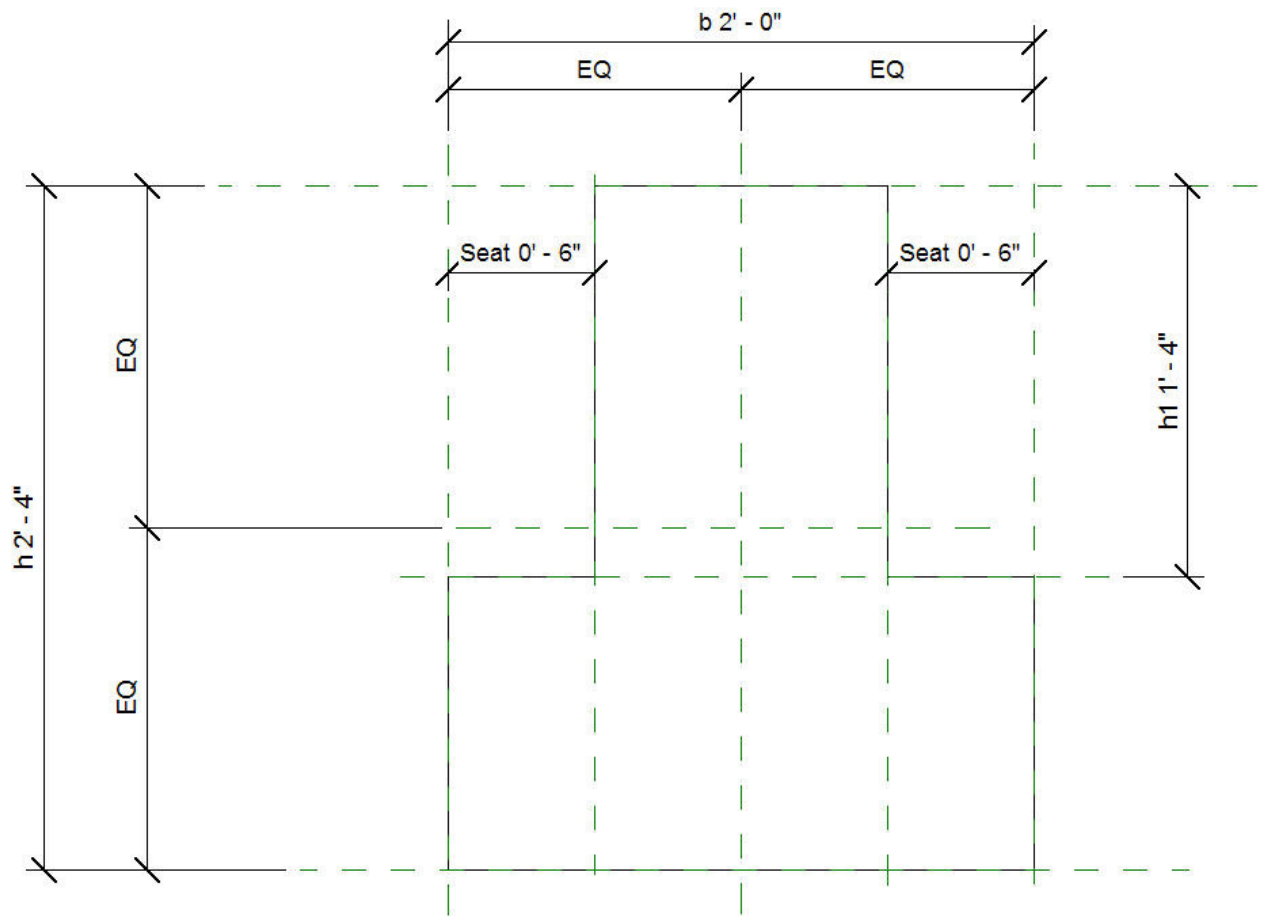


Figure 222: : Precast-Inverted Tee Cross Section

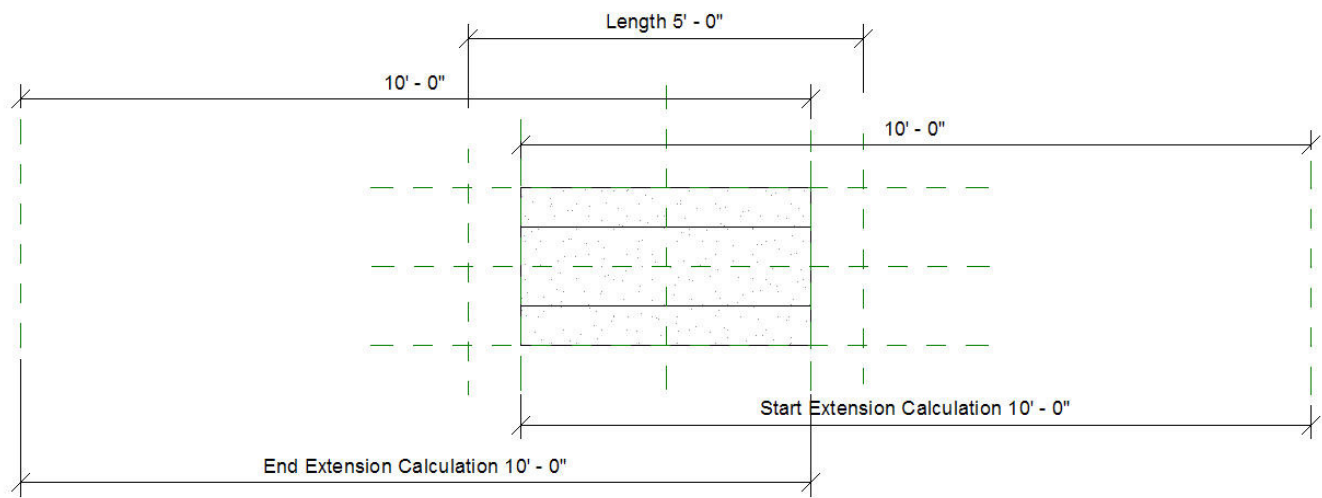


Figure 223: Precast-Inverted Tee

Precast-Double Tee

Family Types

Name: 8'x18"

Parameter	Value	Formula
Construction		
Start Extension (default)	-0' 0 1/2"	=
End Extension (default)	-0' 0 1/2"	=
Materials and Finishes		
Beam Material (default)	Concrete - Precast Concrete - Normal W	=
Structural		
Angle (default)	0.000°	=
Dimensions		
Depth	1' 6"	=
Length (default)	5' 0"	=
Slab Depth	0' 2"	=
Stem Width	0' 3 3/4"	=
Tee Width	4' 0"	=
Width	8' 0"	=
Identity Data		
Assembly Code	B10	=
Keynote		=
Model		=
Manufacturer		=
Type Comments		=
URL		=
Description		=
Cost		=
Other		
Start Extension Calculation (default)	10' 0"	=10' 0 1/2" + Start Ext
End Extension Calculation (default)	10' 0"	=10' 0 1/2" + End Ext

Family Types: New... Rename... Delete

Parameters: Add... Modify... Remove

OK Cancel Apply Help

Figure 224: Precast-Double Tee

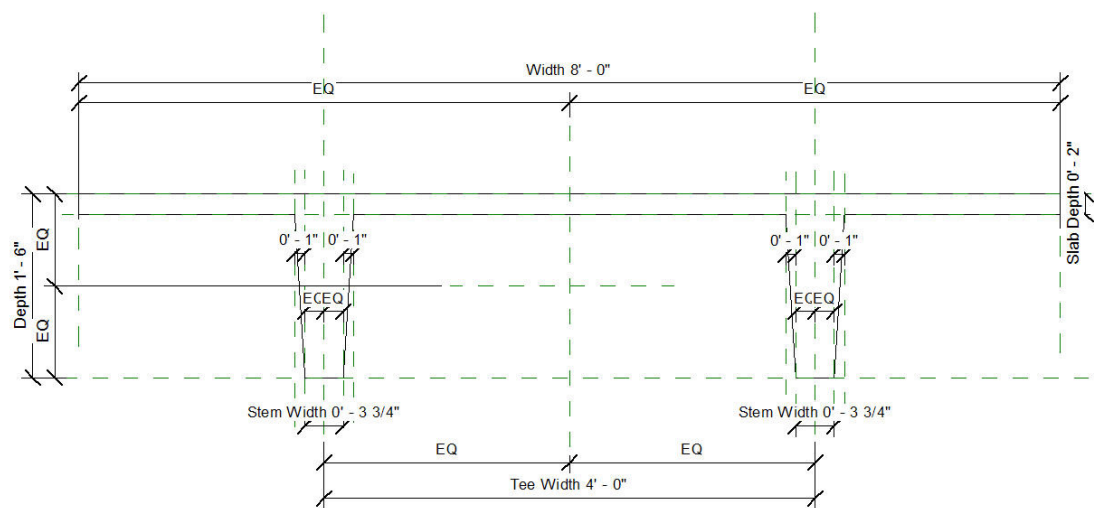


Figure 225: Precast-Double Tee Cross Section

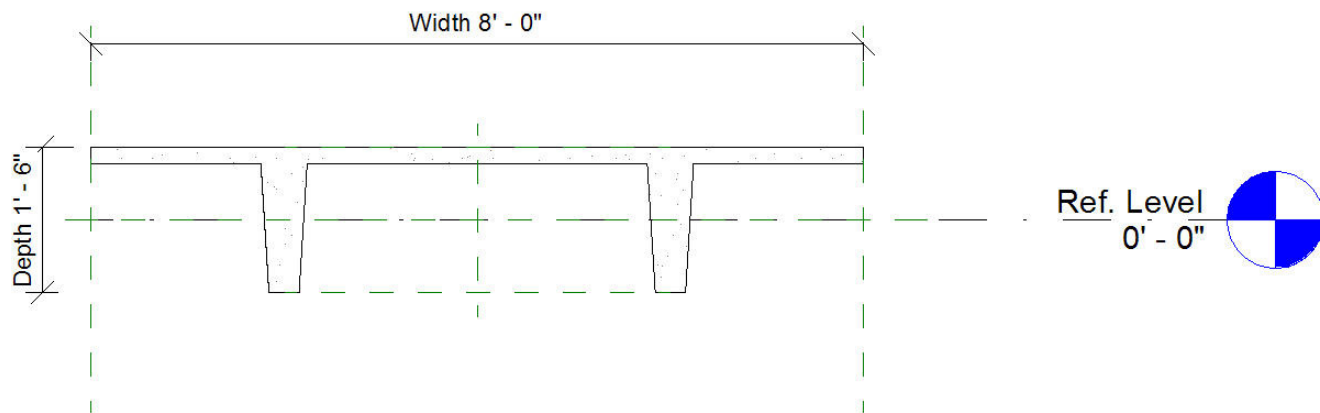


Figure 226: Precast-Double Tee Cross Section 2

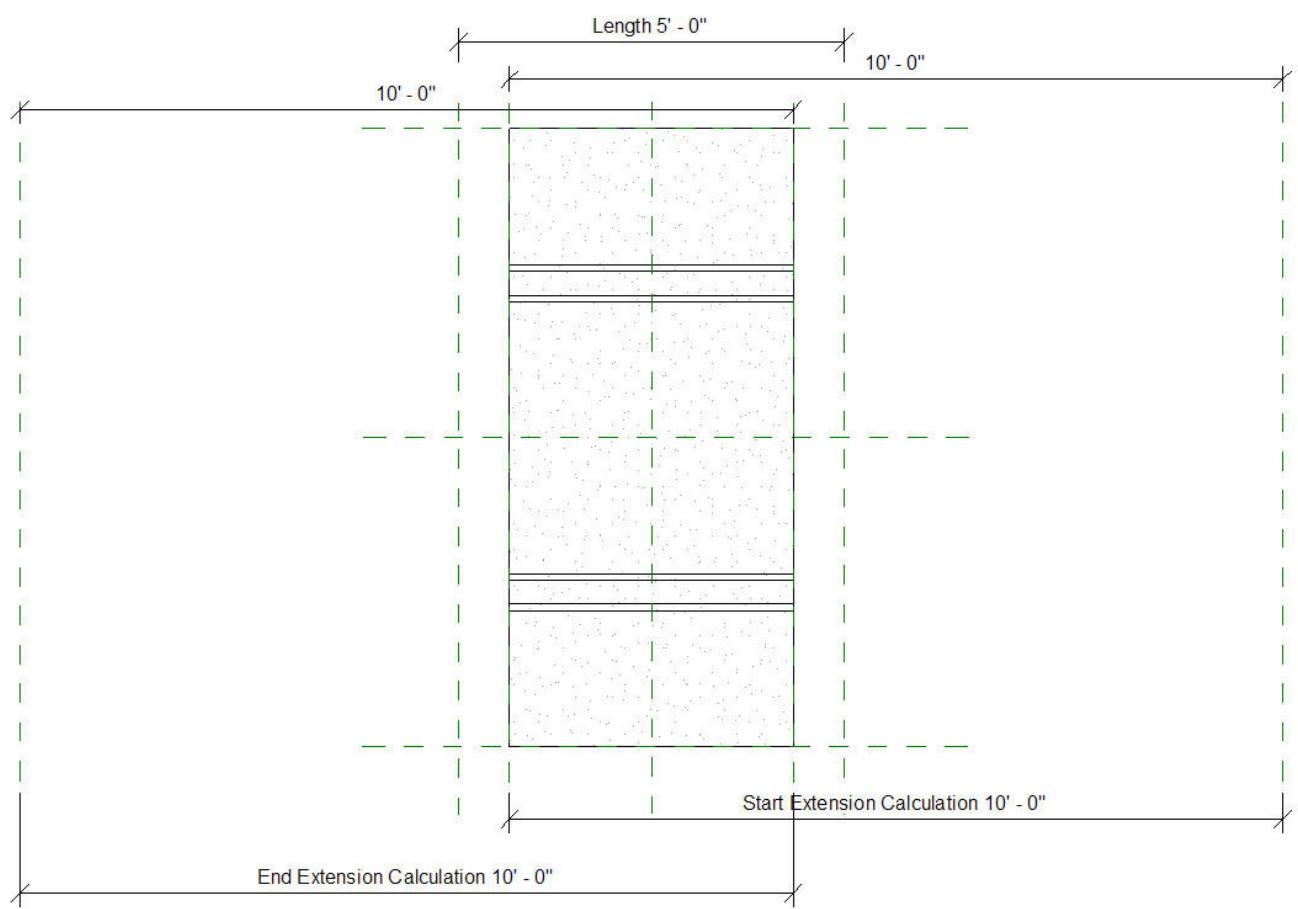


Figure 227: Precast-Double Tee

F. API Ribbon Layout Guidelines

Introduction

The following are aspects of the ribbon UI that can be modified by individual add-in developers. These guidelines must be followed to make your application's user interface (UI) compliant with standards used by Autodesk.

Terminology

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](#). Some minor modifications were added.

- **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.

Definitions

Ribbon

The horizontal tabbed user interface across the top of the application frame in Revit 2010 products.

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.

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.

Button

The button is the mechanism for launching a command. They can either be large (see Measure in Figure 228) or small (see Report Coordinates in Figure 228.) Both large and small buttons can either be a simple push button or a drop-down button (see below.)

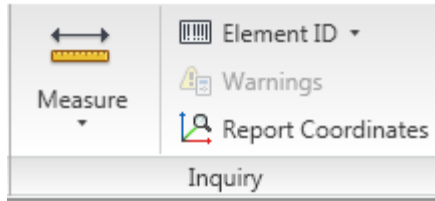


Figure 228: Inquiry panel in the 2010 Revit products

Menu button

The default first panel on the Add-Ins tab is the External Tools panel that contains one button titled "External Tools.". The External Tools menu-button is equivalent to the Tools > External Tools menu in Revit 2009. Any External Command registered in Revit.ini under [ExternalCommands] will appear in this menu button.

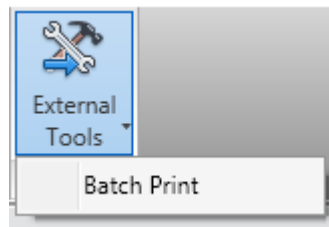


Figure 229: External Tools menu-button on Add-Ins tab

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 (see Figure 231 below.)

Vertical Separator

A vertical separator is a thin vertical line that can be added between controls on a panel (see to the right of Measure button in Figure 230 below.)

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 command's expected behavior.

Layout Guidelines

Number of Panels per Tab

Each API application SHOULD add only one panel to 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 from Revit 2010 provides an example to follow.

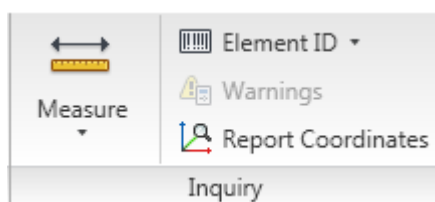


Figure 230: Inquiry panel in the 2010 Revit products

General layout

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 section 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 (see below.)

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.

Drop-down button:

- The top label **SHOULD** sufficiently describe the contents of the drop-down list. For example, in Figure 231, Roof, Curtain System, Wall and Floor are all ways of modeling by face.)
- Every item in the list **SHOULD** contain a large icon.

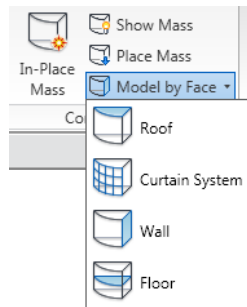


Figure 231: Model by Face drop-down button in Revit Architecture 2010.

Vertical separator

A vertical separator **MAY** be added between a control or sets of controls to create distinct groupings of commands within a panel. For example, in Figure 230, the Measure button deals with measuring elements while the items in the three button stack deal with querying data from the model. A panel **SHOULD** have no more than two separators.

Icons

For proper icon design, see the attached icon design guidelines.

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, and 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.

Note: Panel resizing and collapsing is handled automatically by the ribbon component.

Button Labels

These guidelines are for English language only.

1. MUST not have any punctuation (except hyphen, ampersand or forward slash, see below)
2. MUST be no longer than three words
3. MUST be no longer than 36 characters
4. MUST be Title Case; e.g. Show Mass
5. The ampersand '&' MUST be used instead of 'and'. A space should appear before and after the ampersand.
6. The forward slash '/' MUST be used instead of 'or'. No spaces should appear before and after the slash.
7. Only large buttons MAY have two line labels but MUST NOT have more than two lines. Labels for all other widgets MUST fit on a single line.
8. Button labels MUST NOT contain ellipses (...).
9. 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, regardless of what they are.

Panel Labels

These guidelines are English-only. All rules from the Button Labels section apply to Panel Labels in addition to the following:

1. 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.
2. Applications MUST NOT use panel names that use the abbreviations 'misc.' or 'etc.'
3. Panel labels SHOULD NOT include the term 'add-ins' since it is redundant with the tab label.
4. Panel labels MAY include the name of the third party product or provider.

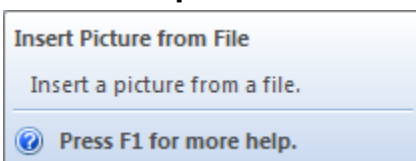
Tooltips

The following are guidelines for writing tooltip text. Write concisely. There is limited space to work with.

1. Make every word count. These guidelines are particularly important for localizing tooltip text to other languages.
2. Use simple sentences. The "Verb-Object-Adverb" format is recommended.
3. Use only one space between sentences.
4. Avoid repetitive text. The content in the tooltip should be unique and add value.
5. Don't include lengthy step-by-step procedures in tooltips. These belong in Help.

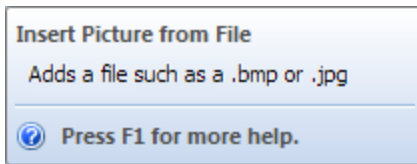
6. Use terminology consistently.
7. Don't use gerunds (verb forms used as nouns) because they can be confused with participles (verb forms used as adjectives). In the example, "Drawing controls", drawing could be used as a verb or a noun. A better example is "Controls for drawing."
8. Focus on the quality and understandability of the tooltip. Is the description clear? Is it helpful?
9. Use strong and specific verbs that describe a specific action (such as "tile") rather than weak verbs (such as "use to...").
10. Write in the active voice (for example, "Moves objects between model space and paper space").
11. Use the descriptive style instead of the imperative style ("Opens an existing drawing file" vs. "Open an existing drawing file").
12. 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.
13. Make the tooltip description easily recognizable by using the third person singular (for example – "Specifies the current color" instead of "Specify the current color".)
14. Don't use slang, jargon, or hard to understand acronyms.
15. Make sure that your use of conjunctions doesn't 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".
16. 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.
17. 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"?
18. 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".
19. Be careful using metaphors. Metaphors can be subtle. Metaphors are often discussed in the context of icons that are not culturally appropriate or understood across cultures, but 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".
20. Avoid abbreviations. For example, the word "Number" has many common abbreviations: No., Nbr, Num, Numb. It is best to spell out terms.

Bad Example:



In this example, the tooltip content repeats the tooltip title verbatim and does not add value to the tooltip. Additionally, there are translation issues with this example. If the translator can't identify whether this string is a name/title or a descriptive sentence, it will be difficult for them to decide on the translation style.

Good Example:



An example of a more useful descriptive sentence might be “Adds a file such as a .bmp or .jpg”. This provides more detailed information and gives the user more insight into the feature.