**Enhancing Your Revit® Add-In**

Jeremy Tammik – Autodesk Inc.

**DE205-3** This session will highlight the hottest tips and tricks for enhancing your Revit add-in and delve into Revit element geometry in depth. Learn the best way to make your Revit application language independent; handle exceptions; optimize performance; combine different .NET languages; cache frequently used data; use profiling and benchmarking to avoid bottlenecks; and use the Revit geometry library and the geometrical structure of building elements. This class will benefit all Revit programmers and assumes basic knowledge of Revit programming.

**About the Speaker:**

Jeremy is a member of the AEC workgroup of the DevTech team, providing developer support, training, and conferences to the Autodesk Developer Network ADN. He originally joined Autodesk in 1988 as the technology evangelist responsible for European developer support. In this capacity, he wrote articles, consulted, lectured on AutoCAD application programming techniques, and supported AutoCAD application developers in Europe, the U.S., Australia, and Africa. He was a cofounder of ADGE, the AutoCAD Developer Group Europe, and a prolific author on AutoCAD application development. He left Autodesk in 1994 to work as an HVAC application developer, and then rejoined the company in 2005.

Jeremy graduated with a MA in mathematics and physics in Marburg, Germany, in 1984, and worked first as a teacher and translator of both computer and human languages, then as a C++ programmer on early GUI and multitasking projects. He is fluent in five European languages, vegetarian, has four kids, plays the flute, likes reading, travelling, theatre improvisation and carpentry, loves mountains, oceans, sports and especially climbing.

Jeremy writes a blog on the Revit API at <http://thebuildingcoder.typepad.com> and can be contacted at [jeremy.tammik@eur.autodesk.com](mailto:jeremy.tammik@eur.autodesk.com).

# Tips and Tricks Developing with the Revit API

Jeremy Tammik, Consulting Analyst

Developer Technical Services, EMEA

Autodesk SARL

[Introduction 3](#_Toc216052038)

[Revit API Coding Tips 3](#_Toc216052039)

[0 – Element Filters 4](#_Toc216052040)

[1 – Namespaces, Aliases, and Regions 5](#_Toc216052041)

[2 - Version and Flavour Checking 7](#_Toc216052042)

[3 - Command Results, Error Handling and Transactions 8](#_Toc216052043)

[4 - Language Independence 9](#_Toc216052044)

[5 - Data Caching 9](#_Toc216052045)

[6 - Journaling Mechanism 10](#_Toc216052046)

[7 - Assembly Information 11](#_Toc216052047)

[8 - Suspend Updating 12](#_Toc216052048)

[9 - Comparison of C#, VB and managed C++ 12](#_Toc216052049)

[10 – Creating a new Type 15](#_Toc216052050)

[Geometry 15](#_Toc216052051)

[Geometry Class Hierarchy 15](#_Toc216052052)

[Non-APIObject Classes 15](#_Toc216052053)

[Classes derived from APIObject 16](#_Toc216052054)

[Classes derived from GeometryObject 16](#_Toc216052055)

[Coverage, Creation, and Retrieval 17](#_Toc216052056)

[Geometry sample code 18](#_Toc216052057)

[Appendix 18](#_Toc216052058)

[Learning More 18](#_Toc216052059)

[Acronyms 19](#_Toc216052060)

## Introduction

The two main topics in this session are a collection of Revit API coding **tips**, and some more in-depth information on the Revit **geometry** namespace.

## Revit API Coding Tips

In this section, we discuss the following coding tips:

1. Use Filtering at all Times
2. Namespaces, Aliases, and Regions
3. Version Checking
4. Error Handling and Transactions
5. Language Independence
6. Data Caching
7. Journaling and Unit Testing
8. Assembly Information
9. Suspend Updating
10. Comparison C#, VB and C++

The numbering of these tips originally started at 1, but with the advent of the important API filtering functionality provided in Revit 2009, we added a note on that and put it in right up front.

Ah yes, another important and basic development tip that comes to mind and is not listed above, but will be demonstrated during this session and is explained in detail in <http://thebuildingcoder.typepad.com/blog/2008/09/debugging-a-rev.html>, is how to drive Revit and run your application in the Visual Studio debugger.

### 0 – Element Filters

In the Revit 2008 API, one bottleneck of implementing effective applications was the element access, such as retrieving specific building elements from the Revit database. This bottleneck could be worked around somewhat by ensuring the application made one single pass through all elements and selected absolutely everything it required in that single pass, which required some extra effort and planning.

Element access has been tremendously speeded up in the Revit 2009 API by the implementation of API filters.

To compare the two, we take a look at retrieving the first building level from a Revit document. Several of the later tips will create walls for test purposes. To do so, they need to determine a level to place it on. Demonstrate selecting the first level using 2008-style iteration over all document elements, and 2009-style element filtering and compare the time. Actually, the time used for the 2008 style iteration is much less in Revit 2009 than in 2008 due to related optimisations.

Here is the typical 2008-style iteration over all document elements:

public static Level GetFirstLevel\_2008( Document doc )

{

Level level = null;

ElementIterator iter = doc.Elements;

while( iter.MoveNext() )

{

level = iter.Current as Level;

if( null != level )

{

break;

}

}

return level;

}

Here is equivalent code using 2009-style element filtering:

public static Level GetFirstLevel( Application app, Document doc )

{

Level level = null;

Filter f1 = app.Create.Filter.NewCategoryFilter( BuiltInCategory.OST\_Levels );

Filter f2 = app.Create.Filter.NewTypeFilter( typeof( Level ) );

Filter f3 = app.Create.Filter.NewLogicAndFilter( f1, f2 );

ElementIterator iter = doc.get\_Elements( f3 );

while( iter.MoveNext() )

{

level = iter.Current as Level;

Debug.Assert( null != level, "expected filter to return non-null level" );

break;

}

return level;

}

The performance and optimal use of filters has been discussed by Guy Robinson in a note on 'Revit 2009 Performance' and in further detail in the Performance category in ‘The Building Coder’ Revit API blog:

* <http://forums.augi.com/showthread.php?t=82239>
* <http://thebuildingcoder.typepad.com/blog/performance/>

Guy has done performance analysis in real-world projects. Here are some of his comments:

1. If possible, have the user select elements in the active view and process these instead of walking the entire project. This will be significantly faster than any other method.

2. Unless there is an element you cannot get via a combination of filters never walk the tree using Document.Elements. In other words avoid at all costs this C# code:

ElementIterator itor = commandData.Application.ActiveDocument.Elements;

while(itor.MoveNext())

{

Element element = itor.Current as Element;

}

Unfortunately, new users will look to the SDK which in most cases still use this approach.

3. Understand and use filters exclusively where possible. The performance gains particularly on subsequent running of the command during a session will yield a massive improvement in performance. Not all filter options will give the same increase in performance and some do not perform as expected. Test against real world projects not blank projects. To understand what you can do, start with the filter sample included in the SDK. Then move on to writing simple test commands to evaluate different filter combinations and the resulting performance gains.

4. If you are writing localised commands the approach has been to get a language neutral category and test an element against this. However, filters make this redundant, which is a good thing. Never have the following line of code execute every time a command is run.

ElementId categoryId

= doc.Settings.Categories.get\_Item( BuiltInCategory.OST\_Doors ).Id;

This single line walks the whole element tree building a unique set of categories. So in one line you can at a minimum double the time a command would take to run. If you have to get a category do it once and persist the id somewhere either per session or across sessions.

Note that one of the most effective things to filter for is category. Also note that in a Boolean expression of filters, the most effective filters, i.e. the ones that reject as many elements as quickly as possible, should be placed first. This reduces the need to run the less effective and more time consuming filters on many elements that will be rejected by other filters later on anyway.

An easily accessible tool to explore all aspects of element filtering is the new ElementsFilter SDK sample.

### 1 – Namespaces, Aliases, and Regions

#### Namespace

Some type names are duplicated in different namespaces. For instance, Application is defined at least three times, and Element at least twice:

* Autodesk.Revit.Application
* Autodesk.Revit.Creation.Application
* System.Windows.Forms.Application
* Autodesk.Revit.Element
* Autodesk.Revit.Geometry.Element

The ambiguities caused by the duplicate type names must be resolved somehow. There are several different resolution possibilities.

1. No global 'using'

* Namespace prefix everything
* No ambiguities

This is a rather radical approach, and makes the code somewhat lengthy, cluttered, and unreadable. It also defeats the purpose of the ‘using’ statement. You can however combine this approach with alias statements to introduce shorter versions of the full namespace prefixes.

2. Global 'using'

* Use aliases to disambiguate individual types

This approach is somewhat more elegant and permits succinct code. One can add all required 'using' statements up front, and use individual aliases to disambiguate any conflicts.

The syntax of the ‘using’ statement differs between C#, VB and managed C++. All three flavours are presented in the code provided for tip #9.

#### Namespace Alias

Here is an example of using the first approach, creating aliases for Revit API namespaces and types:

using Rvt = Autodesk.Revit;

using RvtGeom = Autodesk.Revit.Geometry;

using RvtSymbs = Autodesk.Revit.Symbols;

using RvtElems = Autodesk.Revit.Elements;

Prefix every type with its namespace alias. This completely avoids ambiguity and makes code readable and maintainable. This approach ensures that you always know exactly which type comes from which namespace.

Rvt.Application app = commandData.Application;

Rvt.Creation.Application creApp = app.Create;

Rvt.Creation.Document creDoc = app.ActiveDocument.Create;

Rvt.Element buildingElement;

RvtGeom.Element geometryElement;

WinForms.MessageBox.Show( "Hello", "Revit Tips" );

return Rvt.IExternalCommand.Result.Succeeded;

#### Global 'using' and Type Alias

Here is a similar example using the second approach. The Revit API namespaces are used globally. Whenever a type name is ambiguous, an alias for that type is introduced to disambiguate.

using Autodesk.Revit;

using Autodesk.Revit.Geometry;

using RvtElement = Autodesk.Revit.Element;

using GeoElement = Autodesk.Revit.Geometry.Element;

using CmdResult = IExternalCommand.Result;

In this idiom, no namespace prefixes are required, making the code yet more readable and totally avoiding clutter.

Application app = commandData.Application;

RvtElement buildingElement;

GeoElement geometryElement;

return CmdResult.Succeeded;

#### Region

We can use the C# #region and #endregion directives to enclose boilerplate code or individual code sections. These can then be collapsed or expanded as needed to provide an overview of an entire module to further avoid screen clutter. Managed C++ and VB provide similar directives, #pragma region, endregion and #Region, #End Region respectively. An example in C# might look like this:

**#region Test**

**// This is a test.**

**#endregion // Test**

Here is equivalent code in VB:

**#Region "Test"**

**' This is a test.**

**#End Region**

Here is equivalent code in C++:

**#pragma region Test**

**// This is a test.**

**#pragma endregion**

### 2 - Version and Flavour Checking

Access to the version information of the running instance of Revit was added in the Revit 2008 API. One can check the application name and version and act accordingly. The version information is available in three properties on the Revit Application object:

* VersionName

"Revit Architecture"

"Revit Structure"

"Revit MEP"

* VersionNumber

9

2008

etc.

* VersionBuild

#### Version Number

Check version number and enable or disable specific code or functionality:

Application app = commandData.Application;

if( 2009 > int.Parse( app.VersionNumber ) )

{

message = string.Format( "You need a higher version of Revit,"

+ " at least 2009, not {0}.", app.VersionNumber );

return CmdResult.Cancelled;

}

#### Revit Flavour

Check the Revit flavour:

enum Flavour { Architecture, Structure, MEP, Other };

Flavour flavour = Flavour.Other;

foreach( Flavour a in Enum.GetValues( typeof( Flavour ) ) )

{

if( versionName.Contains( a.ToString() ) )

{

flavour = a;

break;

}

}

Wall newWall = null;

switch( flavour )

{

case Flavour.Architecture:

newWall = creDoc.NewWall( line, level, false );

break;

case Flavour.Structure:

newWall = creDoc.NewWall( line, level, true );

break;

case Flavour.MEP:

default:

// no wall created in MEP or other flavour

break;

}

### 3 - Command Results, Error Handling and Transactions

#### Revit Command Results

The Revit external command Execute() method has two output arguments, an optional error message and an optional element set, as well as returning a status code. The status code is an IExternalCommand.Result enumeration value, which can be Succeeded, Cancelled or Failed. You can define a handy alias for it:

using CmdResult = Autodesk.Revit.IExternalCommand.Result;

The error message and element set output arguments are used to display a message to the user and highlight entities if any other command result status code than Success is returned.

The sample command works only with wall elements and takes the following scenarios into account:

1. If nothing is selected, return an error message and Cancelled result code;

2. If only walls are selected, pop up a good message and return a Succeeded result code;

3. If some elements other than walls are selected, put those elements into the returned ElementSet collection, put an error message into the returned string, and return an error code. Revit will highlight those elements and pop up a dialog to alert us of the error.

#### Exception versus Return Code

* Exceptions replace and vastly simplify traditional error handling
* Almost every line of code can cause out of memory or disk space or other problem
* Traditional error handling handles this at every line of code, and every level of the call stack
* Every function call and every return value requires checking and handling
* This produces code bloat, clutter, and complexity overhead
* Exceptions can replace all this by one single top level exception handler

#### Error Handling

A Revit application should use exception handling to deal with error conditions.

try

{

// normal execution stays here

return CmdResult.Succeeded;

}

catch( Exception ex )

{

// handle exceptions here

message = ex.Message;

return CmdResult.Failed;

}

finally

{

// do clean up here

}

An exception should be exceptional! Do not misuse exceptions to handle expected situations, return values, invalid input etc. Exception handling can be combined with the Revit command return values. You can sometimes eliminate the ‘finally’ code block by using the .NET ‘using’ statement, as objects opened or created in the ‘using’ statement will be disposed automatically when they go out of scope.

#### Transactions

You can combine Revit transactions with .NET exception handling. The Revit Document class provides methods to begin, end and abort a transaction:

* BeginTransaction()
* EndTransaction()
* AbortTransaction()

Typically, one would wrap any section of application code in which unexpected error conditions might occur in a try block, start a transaction at its beginning, and end the transaction at its end. Code in the catch block might abort the transaction, since an unexpected situation occurred. Both the try blocks and the transactions can be nested.

Note that Revit automatically begins a transaction for each external command execution, and then decides whether to submit or abort the changes depending on the command return value.

Working with a non-UI document, e.g. opened using OpenDocumentFile(), absolutely requires a transaction to be opened and managed for any changes to take effect.

You can look at the Revit SDK sample TransactionControl to see transactions in action. It implements a user interface form named TransactionForm with three buttons to begin, end or abort a transaction and three other buttons to create, move or delete a wall. The requested operations and transactions and their relationships and nesting are displayed live in a tree view as it is being defined.

### 4 - Language Independence

Use language independent values whenever possible. Such values are provided by the BuiltInCategory and BuiltInParameter enumerations for categories and parameters respectively:

RvtElement e;

Categories categories = e.Document.Settings.Categories;

Category doorCategory = categories.get\_Item( BuiltInCategory.OST\_Doors );

bool languageDependentTest = ( "Doors" == e.Category.Name ); // avoid this!

bool invalidCategoryComparison = ( e.Category == doorCategory ); // wrong!

bool languageIndependentTest = ( e.Category.Id.Equals( doorCategory.Id ) ); // yes

Categories can no longer be compared by direct equality like they could in Revit 2008. Instead, you now need to compare categories by comparing their element id.

#### Determining the Revit Language

For a localised application, you can use the .NET resource manager to handle localised resources.

ResourceManager rm = Resources.ResourceManager;

To determine the currently running Revit language through the API, you can make use of known localised names of the BuiltInCategory and BuiltInParameter enumerations. Depending on the Revit language, an appropriate matching application resource can be loaded. Another possibility is to scan the Revit.ini file for the language section:

[Language]

Select=ENU

### 5 - Data Caching

Retrieving certain Revit document elements or other data costs a certain amount of time. In Revit 2008, no command should ever iterate over the elements collection more than once. There, all required data can and should be collected in one single iteration. From Revit 2009 onwards, by using the new element filtering, repeated calls to retrieve different sets of database elements is more tolerable. Still, you need to carefully benchmark your application and its element access using real-world sample models to ensure that no bottlenecks ensue.

#### Caching Data for Reuse

You can sometimes make use of a data cache for later use to improve application performance. For instance, one can use static members to store cached data. In the sample code, we implement a cache containing all the current document levels and wall types, paralleling the Document.WallTypes property.

public class CommandBuildDataCache : IExternalCommand

{

static private ElementSet \_levelSet = new ElementSet();

static private WallTypeSet \_wallTypeSet = new WallTypeSet();

// . . .

}

One command can populate the cache for other commands to reuse.

Note: currently, this approach has a limitation. You can use this with an assumption that there is no change in the collected data. In future new events will be made available through the Revit API to notify the application when elements are added, deleted, modified, etc. These will enable an application to keep a data cache up to date at all times. Currently, the entire cache needs to be rebuilt to ensure it is up to date.

#### Reuse Cached Data

The cached data can be reused in another command:

string str = string.Empty;

int n = CommandBuildDataCache.WallTypes.Size;

if( 0 < n )

{

str += string.Format( "{0} wall type{1} cached:\n", n, Util.PluralSuffix( n ) );

foreach( WallType wt in CommandBuildDataCache.WallTypes )

{

str += "\n " + wt.Name;

}

}

n = CommandBuildDataCache.Levels.Size;

if( 0 < n )

{

str += string.Format( "\n\n{0} level{1} cached:\n", n, Util.PluralSuffix( n ) );

foreach( Level l in CommandBuildDataCache.Levels )

{

str += "\n " + l.Name;

}

}

Util.InfoMsg( str );

### 6 - Journaling Mechanism

Testing is an important and potentially time-consuming step of commercial software development. Any possibility of automating this repetitive task should be fully exploited. The Revit platform and its API offer significant support for automated unit testing through the journaling mechanism.

All user actions in Revit are recorded in a journal file in the Journals subdirectory of the Revit root directory. To rerun journal file, you can either drag it onto Revit icon or pass it as command line parameter.

An external application can make full use of the journaling mechanism. It can check whether the application is being driven by a user or a journal file. When user driven, it can add appropriate information to the journal file. When driven by rerunning the journal file, it can retrieve the information from the journal file and act appropriately, repeating the stored actions. The relevant class is StringStringMap, provided by the ExternalCommandData.Data property, and its main methods are Insert() to record data and Item() to access it for rerunning the file.

The Revit SDK Journaling sample demonstrates this. The example application adds a wall and saves its data. The journal file content generated by running the Journaling sample command looks like this:

'E 19-Oct-2008 19:04:39.406; 0:<

Jrn.Command "Menu" , "Journaling , 6604"

' 3:< VM Delta: Available -14 MB -> 1577 MB, Used +0 MB -> 173 MB

' 14.236749!!! 2:!!!BIG\_GAP API External Command Time

' 5:< DBG\_INFO: RegenStep changes another element: GoodSymbol of FamilyInstanc

' 14.368883 1:<<API External Command Time

'H 19-Oct-2008 19:04:53.796; 0:<

Jrn.Data "APIStringStringMapJournalData" \_

, 4, "End Point", "(10,10,0)" \_

, "Level Id", "13071", "Start Point" \_

, "(0,0,0)", "Wall Type Name", "Curtain Wall"

'H 19-Oct-2008 19:04:53.812; 1:<

Jrn.Data "Transaction Successful" \_

, "Journaling"

' 0:< idle0\_doc

The journal file is a simple text file, so it can also be generated or manipulated by other means and used to automatically drive Revit for certain purposes. Please note that this is unsupported, and you are completely on your own here if and when you run into problems or limitations, which is pretty probable.

#### Unit Testing

The Revit journaling mechanism can be combined with automated unit testing to ensure that basic application functionality is intact and no regressions occur. To do so, unit tests can be set up which launch Revit.exe with a predefined journal file containing commands which test application functionality, possibly on a pre-existing model loaded from a Revit project, and save the results in an output format which can be compared to verify that it matches the expected result.

### 7 - Assembly Information

Every .NET assembly includes specific version information.

* Major Version
* Minor Version
* Build Number
* Revision

Visual Studio can auto increment the build and revision number, as noted in the default version of AssemblyInfo.cs:

// You can specify all the values or you can default the

// Revision and Build Numbers by using the '\*' as shown below:

[assembly: AssemblyVersion("1.0.\*")]

[assembly: AssemblyFileVersion( "1.0.0.0" )]

The compiler-generated build is the number of days since 2000-01-01, and the revision is the number of seconds since the previous midnight MOD 2.

An application can retrieve the assembly name and version information and make appropriate use of it, for instance in a message or the about box.

System.Reflection.Assembly asm

= System.Reflection.Assembly.GetExecutingAssembly();

string str = "Assembly Information:"

+ "\nName: " + asm.GetName().Name

+ "\nVersion: " + asm.GetName().Version;

Util.InfoMsg( str );

The most complete demonstration of an About box completely driven by the .NET assembly information is provided in the ADN sample provided for the session **DE301-2** The Ins and Outs of Revit MEP Programming.

### 8 - Suspend Updating

Creating an instance of a SuspendUpdating object will cause Autodesk Revit to suspend certain updating and consistency operations, thereby enabling increased performance of large scale changes via the API. When the object is disposed, updating will take place if required. Currently, suspend updating functionality works primarily with object transformation operations such as move and rotate. See the Revit API help file for a list of the methods and properties supporting SuspendUpdating.

No performance increase is guaranteed, and indeed the small scale tests we ran did not show any important gains, but it is important to test whether this improves performance in any large scale operations you require in your application.

Stopwatch stopWatch = Stopwatch.StartNew();

if( useSuspendUpdating )

{

using( SuspendUpdating suspender = new SuspendUpdating( \_doc ) )

{

rc = \_doc.Mirror( walls, line );

}

}

else

{

rc = \_doc.Mirror( walls, line );

}

stopWatch.Stop();

TimeSpan ts = stopWatch.Elapsed;

For another example of using this functionality, please refer to the Revit SDK sample application FrameBuilder.

### 9 - Comparison of C#, VB and managed C++

Most of the sample code accompanying this presentation is written in C# using constant naming conventions, some utility functions, the same type aliases, and similar coding style. Some developers prefer other languages such as VB.NET and managed C++ to customise Revit. Reasons may be personal preference and to reuse existing code in a straightforward way. In this section, we discuss similarities and differences among the three most popular .NET compliant languages, C#, VB.NET and C++. The comparison is useful for migration and comparison purposes, and above all, it gives you an impression of what the same code looks like in the languages you may not commonly use.

In addition, since the Revit SDK does not provide any C++ samples, the skeleton provided here provides a useful starting point. Just like in C# and VB, creating a C++ add-in project for Revit is just a matter of a few minutes. C++ and its option to mix managed and unmanaged code makes it an attractive alternative for some developers, even though VB.NET and C# are easier to use.

We compare code in all three environments and compare the syntax of various statements.

#### C# or VB.NET

Coding in C# or VB.NET is quite similar. It is relatively easy to convert code from one to the other, and there are some existing tools on the internet, which can help us do so. It is still useful to know the coding differences in the two languages.

* C# and VB.NET are equivalent
* The Intermediate Language or IL code generated is identical
* Automatic translators are available
* Google for "c# vb.net translator"
* Reflector decompiles IL into C#, VB, managed C++ and other languages
* Many SDK samples are in C#, some in VB.NET, none in managed C++
* Except for language comparison, presentation labs are in C#

Decompilation also brings up the topic of obfuscation. Decompilation, translation, and obfuscation are discussed on the blog. Here is a brief overview of some prominent coding differences between C# and VB.NET:

|  |  |  |
| --- | --- | --- |
| **Feature** | **C# syntax and Sample Code** | **VB.NET syntax and Sample Code** |
| End of line | ; | Nothing |
| Importing namespace | Keyword: Using  using Rvt = Autodesk.Revit; | Keyword: Imports  Imports Rvt = Autodesk.Revit |
| Retrieving some properties | Class.get\_PropertyName  elem.Document.get\_Element(  param.AsElementId ); | Class.PropertyName  elem.Document.Element(  param.AsElementId ) |
| Conversion to string | Explicitly:  param.AsDouble().ToString(); | Implicitly:  param.AsDouble |
| Passing reference arguments | Explicitly use ‘ref’ keyword  listWallFamilyAndTypeInfo(  elem, ref catName ); | Implicitly:  listWallFamilyAndTypeInfo(  elem, catName ) |

#### Managed C++

Managed C++ is more powerful and flexible than C# or VB. It uses some special symbols and features not available in the other two environments which allow us to mix managed and unmanaged code and introduces some new syntax to specify memory management:

|  |  |  |
| --- | --- | --- |
| **Feature** | **C# syntax and Sample Code** | **C++ syntax and Sample Code** |
| Assembly reference | Use the Add Reference UI to add references to the References folder | #using <RevitAPI.dll>  Alternatively, through a different UI than in C# IDE: Project > Properties > Common Properties > References. This enables setting the ‘Copy Local’ property to false. |
| Namespace and type delimiter | Dot:  System.Windows.Forms | Double colon:  System::Windows::Forms |
| Import namespace | using System; | using namespace System; |
| Alias | Keyword: using  using Rvt = Autodesk.Revit; | Keyword: namespace  namespace Rvt = Autodesk::Revit; |
| Handle to object | none | ^  This symbol declares a handle to an object on the managed heap. It was introduced in VS 2005 to replace the old \_\_gc \* syntax. Garbage collector uses it to release heaps property at right time.  All pointers should be declared as handles to object so that they can be tracked precisely in the C++ managed code, and member selection through a handle (^) uses the pointer-to-member operator (->). |
| Tracking reference | none | %  A tracking reference is similar to a C++ reference, to specify that a variable be passed to a function by reference and to create an alternative name for an object. However, the object referenced by a tracking reference can be moved during execution by the common language runtime garbage collector. |
| Execute() function signature | public IExternalCommand.Result  Execute(  ExternalCommandData,  ref string message,  ElementSet elements); | virtual IExternalCommand::Result  Execute(  ExternalCommandData ^,  String ^% message,  ElementSet ^ elements); |

One point of interest are the C++ special symbols for parameters and variables, handle to object (^) and tracking reference (%). They are used by the .NET garbage collector to manage heaps precisely and efficiently. Both of these are used in C++ version of the external command Execute() method argument list. .NET assemblies can be referenced explicitly in the code, double colon rather than dot is used as a separator, the statement to import a namespace is using, and namespace defines an alias.

There are actually two possibilities to reference a namespace: either with a #using pragma, similar to including a header file, as in

#using <System.Windows.Forms.dll>

#using <RevitAPI.dll>

This has the disadvantage that certain properties such as the ‘Copy Local’ flag cannot be set. Therefore, I instead recommend referencing namespaces in managed C++ the same way as in C# or VB, through Project > Properties > Common Properties > References.

Here is a snippet of sample code in managed C++ to define an external Revit command:

IExternalCommand::Result CommandListWallInfo::Execute(

ExternalCommandData ^ cmdData,

String ^% message,

ElementSet ^ elements )

{

try

{

Document ^ doc = cmdData->Application->ActiveDocument;

ElementSet ^ selSet = doc->Selection->Elements;

if( 1 != selSet->Size ) {

message = "Please select a wall element";

return IExternalCommand::Result::Cancelled;

}

for each( Element ^ elem in selSet ) {

String ^ catName("");

MessageBox::Show( ListWallFamilyAndTypeInfo( elem, catName ), catName );

MessageBox::Show( ListParameters( elem ), "Wall Parameters" );

Wall ^ wall = dynamic\_cast<Wall ^>( elem );

if( wall ) {

Symbols::WallType ^ wallType = wall->WallType;

MessageBox::Show( ListParameters( wallType ), "Wall Type Parameters" );

}

else {

message = "Please select a wall element";

return IExternalCommand::Result::Cancelled;

}

}

return IExternalCommand::Result::Succeeded;

}

catch( Exception ^ ex ) {

message = ex->Message;

return IExternalCommand::Result::Failed;

}

}

### 10 – Creating a new Type

A frequent question from developers is how to create a new type for a given family, which is not obvious from the help file or existing samples. You can achieve this using the Duplicate() method to create a new type, and then modify its parameters or properties.

## Geometry

In the second part of this session, we take a look at the Revit geometry library. It is encapsulated in the Geometry namespace and includes a set of classes representing commonly used geometrical objects, such as points, lines, curves, and surfaces. It also includes classes representing solids, whose boundary information can be queried, such as edges and faces. It includes point, vector, and matrix calculations, special functions to find intersection of two geometric objects, and much more. Our discussion also includes some comparison with the ObjectARX AcGe geometry library.

The geometry library works with temporary objects in memory, i.e. non database resident objects. Some of the objects can be created through the API using methods on the Application class in the Autodesk.Revit.Creation namespace. Others cannot yet be created through the API and are only provided when existing database elements are queried for their geometry generated internally by the Revit kernel.

### Geometry Class Hierarchy

The Revit API geometry library class hierarchy uses three main root classes:

* Non APIObject
* APIObject
* GeometryObject

All geometrical classes are now reference types derived from APIObject, object, or enumeration interfaces. The XYZ, UV, and BoundingBoxUV classes were value types in 2008 and earlier versions, but are so no longer.

The figure provided in "Revit API Diagram.dwf" in the Revit SDK shows all the Revit API classes, including those defined in the Geometry namespace. The geometry library provides both simple and complex geometry classes. UV and XYZ represent a point and a vector in 2D and 3D, respectively. Transform offers basic matrix operations, such as translation, rotation and scaling. The Curve class provides a base class and common functionality for Line, Arc, Ellipse and the more complex subclass NurbSpline. The Revit API Geometry namespace also includes solid representations including Solid, Face, and Edge. These are used to retrieve geometry information for architectural elements in Revit. These different classes can be grouped together based on the class hierarchy like this:

* Non APIObject – this includes UV and XYZ that represent points and vectors. They are derived directly from standard system classes, and you can create instance of them using the new keyword.
* Autodesk.Revit.APIObject – this includes Transform, Plane, and other helper classes that are used in conjunction with GeometryObject derived classes.
* Autodesk.Revit.Geometry.GeometryObject – this includes Line, Arc, Ellipse, and Solid, Faces, and Edges. Curve instances can be created by NewXXX() methods of the Autodesk.Revit.Creation.Application class, but Solid instances cannot.

### Non-APIObject Classes

In Revit, UV and XYZ are used to represent points and vectors, 2D and 3D respectively. Unlike the more complex geometry classes, they are not derived from APIObject. Unlike ObjectARX's AcGe library, there is no strict distinction between point and vector. You can use them interchangeably depending on the context of the actual usage. UV could be a parameter on a surface. You will find many usages of these classes in the Revit SDK samples. Some classes provide static properties that define convenient predefined data. XYZ, for example, defines XYZ.BasisX as a typical vector representing the X axis, i.e., (1,0,0). Similarly, BasisY and BasisZ define the Y and Z axes, and XYZ.Zero defines the origin (0,0,0). You can find the corresponding 2D properties on UV as well.

Another non-APIObject class is BoundingBoxUV, which represents a 2-dimensional rectangle parallel to coordinate axes.

### Classes derived from APIObject

Some of the classes derived from APIObject are BoundingBoxXYZ, IntersectionResult, MeshTriangle, Options, Plane, Reference and Transform. Transformations are represented by the Transform class, which can be compared to a transformation matrix. You can set up a transformation for translation, rotation, and reflection, and perform such operations on points and vectors. Transform defines an Identity property.

### Classes derived from GeometryObject

Some of the classes derived from GeometryObject are Element, which is a sort of geometry container class, Instance, which represents repeatable instances of symbol geometry, and the "real" geometry classes Curve, Edge, Face, Mesh, Profile, and Solid. The Autodesk.Revit.Geometry.Element class is not to be confused with the Autodesk.Revit.Element one. The latter represents a Revit database element, the former a geometrical one, and they are completely different.

#### Element

The geometry element class is a container for geometric primitives. It is generated from a database element when its geometry is queried. The database element uses a parametric description to define its geometry. The main property of the Element class is Objects, which provides access to the geometric primitives contained in the element. In other words, the geometry element is a container for the database element geometry, and provides the interface and temporary repository to query a building element for its geometry.

#### Instance

The Instance class represents an instance of symbol geometry, which is positioned by this database element. Some of its important properties and methods are:

* Symbol, returning the symbol element that this object is referring to
* SymbolGeometry, the geometric representation of the symbol
* Transform, an affine transformation from the local coordinate space
* Transformed, which transforms this instance and returns the result

#### Curve

The Curve class represents a parametric curve and is the base class for arc, ellipse, line, and nurb spline. It has the typical properties and methods of a geometric curve class such as ApproximateLength, Length, ComputeDerivatives, Distance, Evaluate, Intersect, IsInside, etc.:

* ApproximateLength Get the approximate length of the curve
* IsBound Judge if the curve is restricted to a particular interval
* IsCyclic Bounded geometric line
* Length Get the exact length of the curve
* Period Get the period of this curve

Its methods include Clone, ComputeDerivatives, ComputeNormalizedParameter, ComputeRawParameter, Distance, Evaluate, Intersect, IsInside, MakeBound, MakeUnbound, Project, Tessellate, Transformed, etc. Transformed applies a specified transformation to the curve and returns the result. Tessellate returns a set of line segments approximating the curve. It is used by the simple geometry viewers included in the Revit SDK, which reduce all geometry to line segments and display the result in a .NET form.

#### Edge

The edge class represents a one-dimensional boundary segment of a face or a solid. Its properties include ApproximateLength, EndPointReference, Face, Reference, and it provides methods such as ComputeDerivatives, Evaluate, EvaluateOnFace, Tessellate,TessellateOnFace, etc.

#### Face

The Face class represents a parametric face. It is a base class for more specialised derived classes such as ConicalFace, CylindricalFace, HermiteFace, PlanarFace, RevolvedFace and RuledFace. It provides a Triangulate method to simplify complex curved geometry into a collection of simple planar triangles approximating it. Its properties include Area, EdgeLoops, IsCyclic, IsTwoSided, MaterialElement, Period, Reference, etc., and its methods include ComputeDerivatives, Evaluate, Intersect, IsInside, Project, Triangulate, etc.

There is currently no way to create face from scratch in the Revit API. Generally, making a face is a lengthy process: you need to define the surface and all the edges and how it relates to all the other faces in the solid and the model at large. In fact, in Revit, faces are rarely created hand, but by geometry kernel routines. At the moment, there is no external API to these routines, and designing one is hard. So, for now you can only obtain a face from an existing model element. For instance, this satisfies the need for calculating the closest point to an existing object. The Revit API is not suited for generic geometry with no relation to the model; our expectation is that the client requiring this kind of computation has an own in-house math library anyway.

#### Mesh

The Mesh class represents a mesh consisting of a number of triangles. It can be used to exactly represent a solid, if all its surfaces are planar, and it can be used to approximate a non-planar solid. An approximating mesh instance is returned by the Solid.Tesselate method. The mesh properties include NumTriangles, Transformed, Triangle, and Vertices.

#### Solid

The Solid is a top level representation of a database element solid geometry. From the solid, we can traverse the solid's boundary representation by querying it for its faces and edges. We can also simplify the solid by calling the Tesselate method to obtain simple triangles approximating the geometry, which may initially include complex non-planar surfaces.

### Coverage, Creation, and Retrieval

The Revit API does not provide full coverage of geometry creation. For instance, there is currently no way to create face from scratch in the Revit API. Generally, making a face is a lengthy process: you need to define the surface and all the edges and how it relates to all the other faces in the solid and the model at large. In fact, inside Revit, faces are very rarely created by hand, instead internal geometry kernel routines are used. At the moment, there is no external API to these routines, and designing one is hard too. So, for now you can only obtain a face from an existing model element. For instance, this satisfies the need for calculating the closest point to an existing object. The Revit geometry API is not suited for generic geometry calculations with no relation to the building model. One would expect that developers requiring this kind of computation have their own in-house math library anyway.

To retrieve Revit element geometry, you use the Element.Geometry property, which is accessed through the get\_Geometry() method in C#. The various viewers such as the VB ElementViewer and C# ObjectViewer in the Revit SDK samples demonstrate this. They traverse all the geometry and simplify it down to simple line segments, which are then displayed in a graphics window in a .NET form. The geometry traversal may become recursive, because some geometrical elements represent instances of other geometry, with a transformation applied to it.

Generally, the geometry traversal will start at the top level solid and then iterate down through its faces and edges. The best way to understand this properly is to run the ElementViewer or ObjectViewer in the debugger and track the traversal through the geometry of some simple element.

### Geometry sample code

The Revit geometry sample code is located in a separate module RevitGeometry.cs. To demonstrate and compare the Revit geometry functionality with ObjectARX AcGe, we migrated some ObjectARX AcGe sample code presented in the ARX reference to use the Revit geometry classes instead. That is included in the first section. The following ones discuss the basic curve functionality, access to Revit building elements geometry, its solid, faces, and edges, determining face normals, and how to make use of references to existing geometry and define dimensioning:

1. XYZ and Transform
2. Curve and Line
3. Circle
4. Face
5. References and Dimensioning

For more details, please refer to the comments in the code. Several other geometrical Revit API samples are discussed in the blog.

## Appendix

### Learning More

Here are some suggestions on where to go for further information. First of all, of course, we suggest you learn to know and use the online help and SDK samples which are already at your disposal. After that, here are some further suggestions:

DevTV Introduction to Revit 2008 Programming  
<http://adn.autodesk.com/adn/servlet/item?siteID=4814862&id=10194238&linkID=4901650>

* Recording of Revit 2009 Programming Introduction Webcast  
  <http://adn.autodesk.com/adn/servlet/index?siteID=4814862&id=5475217&linkID=4901650>  
  <http://www.adskconsulting.com/adn/cs/api_course_sched.php>
* Discussion Groups  
  <http://discussion.autodesk.com> > Revit API
* API Training Classes  
  <http://www.autodesk.com/apitraining>
* ADN, The Autodesk Developer Network  
  <http://www.autodesk.com/joinadn>
* DevHelp Online for ADN members  
  <http://adn.autodesk.com>
* The Building Coder, a Revit API blog  
  <http://blogs.autodesk.com/thebuildingcoder>  
  <http://thebuildingcoder.typepad.com>

### Acronyms

ADN Autodesk Developer Network

* AEC Architecture, Engineering, Construction
* API Application Programming Interface
* BIM Building Information Model
* HVAC Heating, Ventilation, and Air Conditioning
* IL .NET Intermediate Language
* MEP Mechanical, Electrical, Plumbing
* RAC Revit Architecture
* RME Revit MEP
* RST Revit Structure
* SDK Software Development Kit