**The ABCs of Programming in Revit®   
Architecture, Revit® Structure, and Revit® MEP**

Jeremy Tammik – Autodesk Inc.

**DE105-1** This class will introduce you to the API for Revit Architecture, Revit Structure, and Revit MEP, and will guide you into the world of Revit programming. You’ll learn about the contents of the Revit Software Development Kit and accompanying documentation, how to set up the Microsoft Visual Studio development environment, and get an understanding of the architecture of a Revit add-in and how it interacts with the software. We’ll explore the Revit database structure and object model; demonstrate how to traverse the Revit Building Information Model (BIM); query, modify, and add new elements and element data; analyze built-in and shared element parameters; list and load families and symbols; and access and display the BIM 3D geometry. This session assumes basic knowledge of C# or VB.NET.

**About the Speaker:**Jeremy 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 South Africa. He was a cofounder of ADGE (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 to work in the AEC group of the Autodesk Developer Network Technical Services Team.

# The ABCs of Programming in Revit

**Revit® Architecture, Revit® Structure, and Revit® MEP**

**Jeremy Tammik, Autodesk Inc., DevTech EMEA**

[1. Introduction 3](#_Toc182039991)

[2. Getting Started 6](#_Toc182039992)

[3. Elements Collection 12](#_Toc182039993)

[4. Families and Types 17](#_Toc182039994)

[5. Parameters 22](#_Toc182039995)

[6. Geometry and Groups 26](#_Toc182039996)

## Introduction

**Content:** Introduce the Revit Application Programming Interface (API) for Revit Architecture, Structure and MEP, and guide the participants into the world of Revit programming. Explain the Revit Software Development Kit (SDK) contents and accompanying documentation, set up the Microsoft Visual Studio development environment, describe the architecture of a Revit add-in and its interaction with Revit. Explore the Revit database structure and object model. Demonstrate how to traverse the Revit Building Information Model (BIM), query, modify and add new elements and element data, analyse built-in and shared element parameters, list and load families and symbols and access and display the BIM 3D geometry. Assumes basic knowledge of C# or VB.NET.

**Target Audience, Who Should Attend:** Application developers and Revit users interested in Revit Architecture, Structure and MEP programming.

**Session Objectives:** After attending this session, you will be able to:

* Create a complete Revit add-in from scratch, both as an external command and an external application
* Export data from Revit to external applications such as Excel, import modified data back in again and update the BIM
* List existing and load new families and symbols into Revit programmatically
* Analyse, query, modify and create elements in the Revit BIM
* Access, analyse and display the BIM 3D geometry

**Companion Presentations:** Please also refer to the following companion presentations, which expand on the basic topics discussed here:

* DE305-1 A Closer Look at the Database with the Revit API
* DE111-1 Autodesk's Revit SDK Sample Smörgåsbord
* DE115-1 Ten Steps for Enhancing Your Revit Add-In
* DE201-1 Create Your Own Bidirectional Revit Structure Stress Analysis Integration Link
* DE205-1 Reinforce Your Design: Revit Structure API for Rebar and Detailing
* DE315-1 Answers to Your API Questions: Experts Talk About Revit and AutoCAD Architecture & MEP APIs

**Materials:** This presentation is accompanied by a set of labs. The labs can be used for self-learning and as reference material. The Labs consist of five separate sections. Each covers basic common topics for beginners. Please refer to the labs readme.txt for installation details.

After discussing the basics of the environment and the creation of a Revit add-in, we will go through a live demonstration of creating such an add-in from scratch, either in C# or VB or both. We will not be able to cover all the rest of the material in depth, and will skim over part of it.

There will be time at the end for questions. Some of the topic and concepts will be discussed in depth in the other Revit API session.

This material is based on the materials used for the Autodesk DevTV presentation and the ADN webcasts that we presented in May and June of 2007, which in turn was based on labs created by Miroslav Schonauer of Autodesk Consulting in VB.NET.

### Agenda

* Introduction: Product, SDK, documentation and samples
* Getting Started and Hello World: Development environment, external command and application interfaces, modifying Revit.ini
* Elements Collection: Identifying an element, getting all model elements, filtering, manipulation
* Families and Types: Standard versus system families, loading, changing type
* Parameters: Access, built-in versus shared, exchange with external applications, hidden, per-doc data
* Geometry and Groups: 2D and 3D geometry, rooms, groups

What are the initial steps in creating a Revit application? The first thing is to understand what material is provided by the SDK, the API architecture, and where to obtain more information. Then we explore how to set up the development environment and create a first "Hello world" type application. After that, we will look into the Revit database structure and its data and elements. The samples provide a valuable knowledgebase on how to solve Revit programming tasks.

### Acronyms

ADN Autodesk Developer Network

AEC Architecture, Engineering, Construction

API Application Programming Interface

BIM Building Information Model

MEP Mechanical, Electrical, Plumbing

RAC Revit Architecture

RME Revit MEP

RST Revit Structure

SDK Software Development Kit

### Revit Products

Revit comes in three flavours:

* Revit Architecture, previously known as Autodesk Revit Building
* Revit MEP, previously Autodesk Revit Systems
* Revit Structure

Product build and distribution:

The DVD version is posted to the ADN site (members only):

* Software & Support > Revit > Downloads

The web version is available from the Autodesk home page:

* Products > Revit Architecture > Product Download

Here are links to the latest download from the public product site, where fixes are constantly applied. There are no separate “patch” service packs.

Revit Architecture

<http://usa.autodesk.com/adsk/servlet/item?siteID=123112&id=9408083>

Revit Structure

<http://usa.autodesk.com/adsk/servlet/item?siteID=123112&id=9281007>

Revit MEP

<http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=9262907>

### Revit SDK

The Revit SDK is available in the DVD version in

\Utilities\Common\Software Development Kit\Revit 2008 SDK.zip

It is also available in the web version at

<Revit install>\Download\Utilities\Common\Software Development Kit\Revit 2008 SDK.zip

Check the Autodesk Revit Developer Center

<http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=248497>

and the ADN site

Software & Support > Revit > Knowledgebase > Code Samples > Revit 2008 SDK

<http://adn.autodesk.com/adn/servlet/item?siteID=4814862&id=9550723&linkID=4901650>

for latest updates.

The Revit API is provided by a .NET assembly, RevitAPI.dll, which is present in every Revit installation.

The Revit SDK is purely for support and documentation purposes. All you actually need to develop a Revit add-in is the development environment and the RevitAPI.dll, nothing else.

### SDK Documentation

* Read Me First.doc
* Getting Started Revit 2008.doc
* What's New in Revit Architecture 2008.pdf
* Revit API Diagram.rvt
* RevitAPI 2008.chm
* Revit Structure: Separate directory
* API Changes

Added.xml and Removed.xml

xslt Transformation

* Samples

Converting 9.1 applications to Revit 2008 API.doc

Revit 2008 New Samples.doc

Here are the top-level contents of the SDK. Revit Structure information is placed in a separate subdirectory. Revit API Diagram.rvt provides an object model, actually the class hierarchy, including the relationships between over 400 classes.

The Revit SDK samples provide the largest knowledgebase of how to address specific programming tasks using the Revit API. The API documentation in the help file lists all the classes and their methods and properties, but does not explain how they work together to solve specific tasks. For this, the current main source of information is the samples collection.

The Revit SDK samples are discussed in detail in the companion session:

* **DE111-1** The Revit SDK Sample Smörgåsbord

### Revit API History

* 5, 6, 7 no API and no verticals
* 8.0 first public API for Building and Structure
* 8.1 "first and a half release"
* 9.0 many new objects and creation methods
* 9.1 journal, units, new creation methods
* 2008 major new features
* 2008.2 includes an SDK update and new samples
* API size has doubled in every single release
* Target is API and kernel-based application
* Still evolving ...

First, a little bit on the history of the Revit API. The API has been and still is evolving very strongly, since Revit 8.0, and has reached a certain maturity now. We are in the long-term process of restructuring Revit to make it more API driven, i.e. create a kernel providing the API on top of which the various Revit flavours can be implemented, instead of implementing the API as the outermost layer on top of the completed product.

## Getting Started

That concludes the introduction. Now let us start looking at real development issues and take our first steps up to a simple “Hello World” example.

The accompanying labs demonstrate the concepts discussed. They can also be used for self-learning, and as a reference to answer the most common Revit API beginner’s questions.

### Getting Started

* Development environment
* .NET class modules
* IExternalApplication interface
* IExternalCommand interface
* Modifying Revit.ini file
* External application user interface
* External tools commands
* Input and output arguments

The recommended development environment is Microsoft Visual Studio 2005 and C# or VB.NET. Actually, in the getting started document, it says "The Autodesk Revit API requires the Microsoft .NET Framework v2.0 and Microsoft Developer Studio 2005." More detailed setup information is provided there. Other languages can be used, of course, since the .NET framework is language independent. A Revit application consists of one or more .NET assemblies implementing a certain interface. To make it known to Revit, some information needs to be added to Revit.ini. In this section, we will look at the topics listed above in more detail. The commands defined by Lab 1 are simple “Hello World” style commands to ensure that the development environment and Revit.ini is correctly set up, and to examine the external command input and output.

### Revit API DLL

* .NET API

Microsoft Visual Studio 2005

Microsoft .NET Framework 2.0

Reference "[Revit XXX n.m]\Program\RevitAPI.dll"

C# or VB.NET

* Revit Architecture, Structure and MEP

Same API DLL

Certain functionality only in Architecture or Structure

No MEP specific API in 2008

For instance, room-related functionality is available in RAC only, the analytical model only in RST.

### C# or VB.NET

* C# and VB.NET are equivalent
* The IL code generated is identical
* Automatic translators are available
* Google for "c# vb.net translator"
* Many SDK samples are in C#
* Some SDK samples are in VB.NET
* Presentation labs are in both C# and VB.NET

### Extending Revit

There are two ways to extend Revit:

1. Add an external command

* Implement IExternalCommand
* Commands are added to the menu
* Tools > External Tools

2. Define an external application

* Implement IExternalApplication
* Applications can add menus and toolbars
* External application makes use of external commands

Both are listed in Revit.ini. We have two flavours of Revit add-in, the external command and the external application.

An external application can define a user interface by adding new elements to the Revit framework: new top-level menus, menu entries, toolbars and toolbar buttons. The widgets are then hooked up with external commands that implement the application functionality.

An external command listed in Revit.ini is always added to the specific Revit submenu Tools > External Tools.

### External Command

* Implement Autodesk.Revit.IExternalCommand interface
* Implement IExternalCommand.Execute method

In VB.NET:

Public Class Lab1\_1\_HelloWorld

Implements IExternalCommand

Public Function Execute( \_

ByVal commandData As ExternalCommandData, \_

ByRef message As String, \_

ByVal elements As ElementSet) \_

As IExternalCommand.Result \_

Implements Autodesk.Revit.IExternalCommand.Execute

MsgBox("Hello World")

Return IExternalCommand.Result.Succeeded

End Function

End Class

Each command is implemented in an own class. The class derives from IExternalCommand. It implements the method Execute(). This method is called when the command is invoked. Execute() always takes three arguments. One is an input and two are output arguments. It also returns a result signalling success, cancel or failure.

The input argument is an ExternalCommandData instance, which gives the application access to the Revit application and documents and so on, as we will see further on.

The two return arguments are only used in case the method returns a failure code, in which case the string message is displayed to the user in a standard Revit error message dialogue, and the ElementSet elements are highlighted, to enable the application to show the user which objects may be causing a problem.

### Command Return Values

public class Command : IExternalCommand

{

public IExternalCommand.Result Execute(

ExternalCommandData commandData,

ref string message,

ElementSet elements )

{

try

{

// normal execution

return IExternalCommand.Result.Succeeded;

}

catch( Exception ex )

{

message = ex.ToString();

return IExternalCommand.Result.Failed;

}

}

}

The second and third parameters to an external command are for passing back an error message and a set of elements to highlight to the user. They are only displayed in the Revit UI if the command returns 'Failed'.

### External Application

* Implement Autodesk.Revit.IExternalApplication interface
* Implement OnStartup() and OnShutdown() methods

In C#:

using CmdResult = Autodesk.Revit.IExternalApplication.Result;

namespace mep

{

class App : IExternalApplication

{

CmdResult IExternalApplication.OnStartup( ControlledApplication app )

{

AddMenu( app );

return CmdResult.Succeeded;

}

CmdResult IExternalApplication.OnShutdown( ControlledApplication app )

{

return CmdResult.Succeeded;

}

}

}

An external application does not automatically appear anywhere in the Revit UI. It can define menu entries and custom toolbars as it likes. The objects are hooked up with external command implementations that are invoked and receive the same input and output parameters as normal external commands added to the external tools menu.

### Application Menu and Command

OnStartup() adds UI elements and hooks them up to IExternalCommand implementations

Here is some code creating a new top level menu and some entries in it, and using the external application assembly path to determine the path of the external commands to hook up to the menu entries:

/// <summary>

/// Add this external application's menu to the Revit menu.

/// </summary>

private static void AddMenu( ControlledApplication app )

{

const string m = "mep.Cmd"; // namespace and command prefix

string path = System.Reflection.Assembly.GetExecutingAssembly().Location;

Autodesk.Revit.MenuItem rootMenu = app.CreateTopMenu( "ME&P API Samples" );

//bool success = rootMenu.AddToExternalTools();

MenuItem.MenuType mt = MenuItem.MenuType.BasicMenu;

rootMenu.Append( mt, "&Do Something", path, m + "Do" );

rootMenu.Append( mt, "A&bout...", path, m + "About" );

}

The input to the Append() method is similar to the Revit.ini EC items, including name, assembly path, and compete class name. The EC Description is accessible through the MenuItem StatusbarTip property. The menu type can be basic, popup menu, or separator.

### Application Toolbar and Command

OnStartup() adds UI elements and hooks them up to IExternalCommand implementations

Here is an example in C# of how a toolbar entry is hooked up with an external command. The command is implemented in a different assembly, so we are juggling three different paths here: the application and command assemblies and the toolbar button images paths.

OnStartup( ControlledApplication application )

{

string appAssemblyName = "RSLinkRevitApp";

string appDllName = appAssemblyName + ".dll";

string cmdNamespace = "RSLinkRevitClient";

string appPath = this.GetType().Assembly.Location;

string cmdPath = appPath.Replace( appDllName, cmdDllName );

string imgPath = appPath.Replace( appDllName, "toolbar.bmp" );

//

// create a custom tool bar and set its image path:

//

Autodesk.Revit.Toolbar toolBar = application.CreateToolbar();

toolBar.Image = imgPath;

toolBar.Name = Title;

//

// add export button:

//

ToolbarItem item = toolBar.AddItem( cmdPath, cmdNamespace + ".RSLinkExport" );

item.ItemType = ToolbarItem.ToolbarItemType.BtnRText;

item.ItemText = "Export";

item.StatusbarTip = item.ToolTip = "Export analytical data from Revit";

}

### Modify Revit.ini

[ExternalCommands]

ECCount=<n>

ECName<i> = Export Fire Rating

ECDescription<i> = Export fire rating values to an Excel file

ECClassName<i> = Revit.SDK.Samples.FireRating.VB.NET.ExportFireRating

ECAssembly<i> = C:\Revit\SDK\Samples\FireRating\VB.NET\bin\Debug\FireRating.dll

...

[ExternalApplications]

EACount=2

EAClassName1=RSLinkRevitApp.RsApp

EAAssembly1=C:\a\j\adn\rst\rst\_api\RSLinkRevitApp\bin\Debug\RSLinkRevitApp.dll

EAClassName2=RvtMgdDbg.App

EAAssembly2=C:\Program Files\Revit Architecture 2008\Program\RvtMgdDbg.dll

Once we have created the application and/or command assemblies, we need to make them known to Revit. This is achieved by adding some information to Revit.ini. An external command appears in the external tools menu if listed as an external command in Revit.ini. It can also be accessed through an external application, with no such entry, or both.

### Getting Started

Now we are ready for our “Hello World” demonstration. Here are some SDK samples and labs for getting started:

* External commands: HelloRevit and HelloWorld
* External applications: APIAppStartup, Toolbar
* "Hello World" basic sample

Lab 1-1 demonstrates a minimal “Hello World” command.

* What arguments are passed to the command?
* What can the command return?

Lab 1-2 explores the command argument issues.

### Minimal Command

Here is the minimal “Hello World” command in VB.NET:

Public Class Lab1\_1\_HelloWorld

Implements IExternalCommand

Public Function Execute( \_

ByVal commandData As ExternalCommandData, \_

ByRef message As String, \_

ByVal elements As ElementSet) \_

As IExternalCommand.Result \_

Implements Autodesk.Revit.IExternalCommand.Execute

MsgBox("Hello World")

Return IExternalCommand.Result.Succeeded

End Function

End Class

And here is the same thing in C#:

public class Lab1\_1\_HelloWorld : IExternalCommand

{

public IExternalCommand.Result Execute(

ExternalCommandData commandData,

ref string message,

ElementSet elements )

{

LabUtils.InfoMsg( "Hello World" );

return IExternalCommand.Result.Succeeded;

}

}

LabUtils.InfoMsg() is a wrapper for System.Windows.Forms.MessageBox.Show().

You cannot get much more minimal than this.

### Command Input Argument

Display some application and document properties, retrieved from the command data input argument.

' List the app, doc and view data

Dim revitApp As Revit.Application = commandData.Application

Dim doc As Revit.Document = revitApp.ActiveDocument

Dim view As Revit.Elements.View = commandData.View

Dim sMsg As String = "Application = " & revitApp.VersionName & \_

" " & revitApp.VersionNumber & vbCrLf

sMsg += "Document path = " & doc.PathName & vbCrLf ' Empty if not saved

sMsg += "Document title = " & doc.Title & vbCrLf

sMsg += "View name = " & view.Name

MsgBox(sMsg)

' List the current selection set

Dim sel As Selection = doc.Selection

sMsg = "There are " & sel.Elements.Size & " elements in the selection:"

Dim elem As Revit.Element

For Each elem In sel.Elements

sMsg += vbCrLf & " " & elem.Category.Name

sMsg += " Id=" & elem.Id.Value.ToString

Next

MsgBox(sMsg)

### Command Output Arguments

Check the document selection set contents. Return an error code to see the message displayed and the element highlighted.

' Let's pretend that something is wrong with the first element

' We pass a message back to the Revit user and indicate the error result

If Not sel.Elements.IsEmpty Then

Dim iter As ElementSetIterator = sel.Elements.ForwardIterator

iter.MoveNext()

Dim errElem As Revit.Element = iter.Current

elements.Clear()

elements.Insert(errElem)

message = "We pretend something is wrong with this element"

+ " and pass back this message to user"

Return IExternalCommand.Result.Failed

Else

Return IExternalCommand.Result.Succeeded

End If

So, that is all there is to say about the Revit add-in architecture, the external application and command interfaces, and their arguments. In the next section, we will start exploring the contents of the Revit database.

## Elements Collection

Now that we understand how to create the bare bones of our add-in and how it sets up its initial communication with the Revit application, let us explore and analyse the contents of the Revit database.

This topic is covered in more depth in the separate presentation:

* **DE305-1** A Closer Look at the Database with the Revit API.

### Elements Collection

* Object model
* Database structure
* Types of elements
* Accessing and identifying elements
* Filtering for types and categories
* Getting all model elements
* Model elements' manipulation

We saw that the only input to the external command is the external command data argument. We will explore how to access the application, current document and their properties through this command data argument passed in to an external command. Then we will explore more detailed access to the Revit BIM and its data. The entire content of the Revit BIM is stored in a database and accessed through the Revit Document. Most of the objects are accessible through the document Elements collection property. Often, one is interested in elements of a particular type, so a common task is iterating over the elements collection and filtering for a specific type and/or property.

### Revit Object Model

The Revit SDK includes a graphical overview of the class hierarchy provided by the API in “Revit API Diagram.rvt”. It includes a large number of classes grouped in several different clusters. Some of the important groups are the elements, which contain almost all the visible building elements in the Revit BIM.

### Important Tree Nodes

* APIObject – root object
* Element – BIM elements
* Symbol – type definition
* GeometryObject – abstract geometry

Here are some of the important tree nodes and parent classes in the Revit object model. Some objects are available only in the RAC or only in the RST environments, such as RoomTagType in RAC, RebarTagType, the analytical model and the loads and boundary conditions in RST.

Everything is derived from APIObject. All physical BIM objects are derived from Element. A separate version of Application and Document is provided in the Creation namespace, for creating new elements.

### Subset of Object Model

Here is a subset of the most important classes that one may have to deal with in a typical Revit programming task. The red classes are the most commonly used, the green Roof class does actually not exist. The family base and family are used to manage collections of related types. Symbol is a base class for all types, also known as symbols. Family symbol is the generic class for these, whereas wall and floor type are more specialised classes. Family instance represents an occurrence or usage instance of a generic family symbol, whereas wall and floor represent the same for a wall or floor type.

### Accessing Revit Elements

How to access the Revit data from an external command?

Two ways

Complete collection:

ExternalCommandData.Application.ActiveDocument.Elements

Current selection:

ExternalCommandData.Application.ActiveDocument.Selection.Elements

The external command input argument provides the ExternalCommandData instance.

So, how do we access all these objects? We start with the external command data argument passed in to the external command.

### Accessing Revit Elements

* Document.Elements

Returns ElementIterator

Go through iterator.MoveNext() and iterator.Current()

Can be saved in an ElementSet for easier manipulation

* Document.Selection.Elements

Returns ElementSet

Use foreach, Size, Contains

### Identifying Revit Elements

* Normal OO approach uses object type or class
* Identifying Revit elements can be brain twisting
* An object can be identified by

Category

Object type or class, i.e. .NET System.Type

Derived from Element or Element/Symbol

* We will need to use a combination of these
* Sometimes additional checks are needed

The identification of elements and determining what type they have, or iterating the entire building model and extracting the specific instances that we are interested in, depends on the objects type. In some cases, we use standard object oriented techniques, simply querying their type or class. In others, we use the object category, which is a Revit API element property. Sometimes, a combination of the two is used, or additional parameters are queried.

More details on this are discussed in the separate presentation:

* **DE305-1** A Closer Look at the Database with the Revit API.

### Elements Collection

* In the Revit API, everything is an APIObject
* In the Revit BIM, everything is an Element
* All are bundled together within and almost exclusively accessed via Elements collection
* Search for ElementIterator and ElementFilterIterator in the SDK samples

91 occurrences in 56 files, many examples

ElementIterator and ElementFilterIterator occur in many of the SDK samples. The command defined in Lab 2-1 iterates over commandData.Application.ActiveDocument.Elements and prints out a line for each element encountered to C:\tmp\RevitElements.txt.

Note that the result of this iteration is large. There are a lot of predefined elements and properties in the Revit database, even before we start adding our own. Once we have added a few elements, note that these are listed at the end, sequentially.

Here is another short example of iterating over the document elements and extracting some basic common element properties:

Document doc = commandData.Application.ActiveDocument;

ElementIterator iter = doc.Elements;

while( iter.MoveNext() )

{

elem = iter.Current as Element;

msg = "Id=" + elem.Id.Value.ToString();

msg += "; Class=" + elem.GetType().Name;

if( null != elem.Category )

{

msg += "; Category=" + elem.Category.Name;

}

}

For comparison, here is a similar iteration over the currently selected elements:

foreach( Element elem in doc.Selection.Elements )

{

msg = "Id=" + elem.Id.Value.ToString();

msg += "; Class=" + elem.GetType().Name;

if( null != elem.Category )

{

msg += "; Category=" + elem.Category.Name;

}

}

### 3D or Model Elements

Often one is interested in 3D elements only

LabUtils.GetAllModelElements()

* Input Revit.Application
* Output ElementSet
* Predicates

Type is not Symbol or FamilyBase

Category is not null

Geometry is not null

Lab 2-2 demonstrates how to extract all elements with a 3D geometry, i.e. visible objects like walls, doors, windows, furniture, etc. This lab iterates over the active document elements collection, like before, and picks out relevant ones. This is a very common operation in Revit programming. The utility function LabUtils.GetAllModelElements() implements this. Its input and output are as defined above. It identifies model elements by applying the checks listed above.

### Specific Element Classes

* How do we get specific element classes, e.g. all Doors or Walls?
* Wall is a separate class

APIObject ← Element ← HostObject ← Wall

(An in-place family wall is a FamilyInstance)

* Door can be identified by its category

System.Type is a generic FamilyInstance

Use BuiltInCategory enum for language independence

Lab 2-3 extracts walls by checking for each element for its class, and doors by checking its category. This is similar to Lab 2-2, but now we check for an even more specific type or a built-in category.

### Element manipulation

* Adding
* Editing
* Deleting

Autodesk.Revit.Creation.Document defines 74 methods for adding elements, supporting 44 different object types supported, because some of the methods are overloaded. For instance, for walls and floors, we have

NewWall( CurveArray profile, bool structural ); // + 4 overloads

NewSlab( CurveArray profile, Level, Line slopedArrow,

double angle, bool isImperial, bool isStructural );

Some objects are still missing.

Editing is rather comprehensive.

Deleting is demonstrated by various SDK samples, for instance DeleteDimensions and DeleteObject.

Lab 2-4 demonstrates some element manipulation:

* Select a wall
* Extract wall from current selection
* Obtain wall top and bottom level constraints
* Calculate geometry from levels and wall curve
* Obtain family symbol for column type
* Insert columns at start, mid and end
* Move wall away from the columns
* Prerequisites

Wall constrained at top

"M\_Wood Timber Column" type loaded

To run Lab 2-4, select a wall. It must be constrained to a level at the top: Element Properties... > Constraints > Top Constraint, otherwise, an error message is displayed. Load the column family named "M\_Wood Timber Column" and the type named "191 x 292mm" prior to running the command, or else highlight the error message displayed, or modify the names to refer to some family type that is loaded. The selected wall is used to define three columns at its end and mid points, and then the wall is moved out of the way to clearly display the columns.

### Categories

Lab 2-5 lists all built-in categories and the entire document category tree.

* Not all built-in categories have a corresponding document category
* Not all document categories have a corresponding built-in category
* There are 598 built-in categories
* 397 of them have associated document categories
* 190 of these are top level parents
* These lead to 423 top-level and children categories

## Families and Types

Many objects in the Revit building model are organised into and defined by families. Each family defines a number of symbols or types. When an element is inserted into the model, the user specifies which family it belongs to, and which specific type it has, so the inserted object is an instance of this family and type. If it is a component family, its class is FamilyInstance. In this section, we discuss the management of families and types.

### Families and Types

* Types are also know as symbols
* Standard versus system families
* Using loaded families and symbols
* Loading new families and symbols
* Determining and changing element's type
* Standard and system families
* Families and Types

In order to insert the column in the last example, we already had the need to determine a suitable family type for it. If the family or that specific type was not available, we had to prompt the user to load it manually. This procedure can be automated. In this section, we explore the handling of families and types in more detail.

There are two types of families: standard ones, which are stored in external Revit family files with the extension \*.rfa, and built-in system families.

### Listing Families and Types

One will frequently need to determine

* Which standard families and symbols are loaded in a model
* Which category do they belong to

This is demonstrated by Lab 3-1.

### Family Iteration

// Iterate all elements and look for Family classes

Document doc = commandData.Application.ActiveDocument;

string sMsg = "Standard Families already loaded in this model:";

ElementIterator iter = doc.Elements;

while( iter.MoveNext() )

{

Family fam = iter.Current as Family;

if( null != fam )

{

// Try to get its Category name; notice that the

// Category property is NOT implemented for the Family class!

string famCatName = (null == fam.Category) ? "?" : fam.Category.Name;

sMsg += "\r\n Name=" + fam.Name + "; Category=" + famCatName;

}

}

LabUtils.InfoMsg( sMsg );

We iterate over the document elements in two loops. In the first, we simple determine all the families. The Category property is NOT implemented for the Family class. Therefore, in the second loop, we retrieve and list all the symbols of each family, and retrieve the family category from the first of its symbols.

### Family Symbol Iteration

// Similar Loop, but now get all the child Symbols (Types) as well.

// These Symbols can also be used to determine the category!

iter = doc.Elements;

while( iter.MoveNext() )

{

Family fam = iter.Current as Family;

if( null != fam )

{

string catName;

bool first = true;

// Loop all contained symbols (types)

foreach( FamilySymbol symb in fam.Symbols )

{

// Determine the category via first symbol

if( first )

{

first = false;

try

{

catName = symb.Category.Name;

}

catch( Exception )

{

catName = "?";

}

sMsg = "Family: Name=" + fam.Name

+ "; Id=" + fam.Id.Value.ToString()

+ "; Category=" + catName

+ "\r\nContains Types:";

}

sMsg += "\r\n " + symb.Name + "; Id=" + symb.Id.Value.ToString();

}

// Show the symbols for this family and allow user to proceed

// to the next family (OK) or cancel (Cancel)

sMsg += "\r\nContinue?";

if( !LabUtils.QuestionMsg( sMsg ) )

{

break;

}

}

}

Here, in the second loop, we retrieve and list all the symbols of each family, and retrieve the family category from the first of its symbols. We can cancel out of this loop once we have seen enough.

### Loading Families and Types

Loading additional standard families and symbols from RFA files is simple, because the document class provides specialised methods for doing so:

doc.LoadFamily()

doc.LoadFamilySymbol()

This is demonstrated in Lab 3-2. This lab loads an entire family and a single symbol from another family, determined by the global variables gsWholeFamilyFileToLoad1, gsWholeFamilyFileToLoad2, gsFamilyFileToLoadSingleSymbol, gsSymbolName.

### Determining Family and Type

Lab 3-3 shows how to determine the Family and Type of an element, and how to list all the symbols applicable to a specific category, such as Windows.

Before running this command, create a wall with a window in it and select the window. First, all the loaded window types are determined, i.e. all the symbols defined by the window family. This is done by iterating over all elements and determining all FamilySymbol instances whose category equals the windows one. Secondly, the selected window's family symbol and from that the family itself is queried and displayed.

### List Window Symbols

//

// Loop through the model to report all FamilySymbol

// objects of "Windows" category.

//

Categories cats = doc.Settings.Categories;

Category catWindows = cats.get\_Item( BuiltInCategory.OST\_Windows );

string sMsg = " Windows Family Symbols in the model are:";

ElementIterator iter = doc.Elements;

while( iter.MoveNext() )

{

FamilySymbol symb = iter.Current as FamilySymbol;

// Check for FamilySymbol having Windows category

if( null != symb )

{

try

{

Category catFS = symb.Category; // for "Profiles" it fails

if( null != catFS )

{

// Either of these comparisons will do :

// if catFS.Id.Value.Equals(catWindows.Id.Value)

// if catFS.Id.Equals(catWindows.Id)

// if catFS.Name.Equals(catWindows.Name)

if( catFS.Equals( catWindows ) )

{

sMsg += "\r\n " + symb.Name + ", Id="

+ symb.Id.Value.ToString();

try

{

Family fam = symb.Family;

sMsg += "; Family name=" + fam.Name

+ ", Family Id=" + fam.Id.Value.ToString();

}

catch( Exception )

{

}

}

}

}

catch( Exception )

{

}

}

}

LabUtils.InfoMsg( sMsg );

Here is an example of determining all the loaded window types, i.e. all the symbols defined by the window family. This is done by iterating over all elements and determining all FamilySymbol instances whose category equals the windows one.

Note that the category can be checked and compared in several different ways. In general, the best way is to make use of the language independent BuiltInCategory enumeration and avoid using the language dependent Name property for comparison, since the category objects themselves can be compared directly.

### Get Family and Symbol

We can query and display the selected window's family symbol and from that the family itself. The family and symbol assigned to a specific family instance are available as properties on the given element and its symbol. For instance, to determine a selected window’s family and type, we read the Symbol property of the given element and its symbol’s Family property.

### Change Family Instance Type

The family type of a family instance can also be changed by modifying its Symbol property.

Lab 3-4 displays a simple dialog to change the type of a family instance. It does the following:

* Determines the family instance category, e.g. Windows
* Assemble a list of all applicable symbols
* Iterate over all elements, check each family category
* If it matches, include all its symbols
* Display dialogue with list box
* Assign selected symbol

Before running this command, create a wall with a window in it, load some additional window families so there is some material to play with, and select the window. Any other symbol instance can also be selected. First, the selected symbol instance category is determined. From that, a list of all other applicable types is determined by iterating over all elements and assembling a dictionary of them. This is a two-step process, identifying the matching families first, and then for each matching family adding all its symbols. With this information in place, a dialogue is displayed allowing the user to select any one of the applicable symbols. If one is selected, it is applied to the selected instance, changing its symbol.

#### Determine Category

We ensure a single element is selected, that it is a family instance, and determine its category.

#### List all Symbols for Category

For the selected symbol instance category, a list of all applicable types is determined by iterating over all elements and assembling a dictionary of them. There are many ways how to store the matching objects, but we choose whatever is most suitable for the relevant UI: We could use Revit's generic Map class, but it is probably more efficient to use the new 2005 .NET strongly-typed Dictionary with KEY = Family name (String) and VALUE = ArrayList (implements iList so we can elegantly bind it to combo box) of corresponding FamilySymbol objects. This is a two-step process, identifying the matching families first, and then for each matching family adding all its symbols. With this information in place, a dialogue is displayed allowing the user to select any one of the applicable symbols. If one is selected, it is applied to the selected instance, changing its symbol.

#### Assign new Type

With this information in place, a dialogue is displayed allowing the user to select any one of the applicable symbols.

If one is selected, it is applied to the selected instance, changing its symbol.

### System Families and Types

Above, we discussed listing and manipulating standard families and types and family instances. Similar but slightly different principles apply when working with system families.

* Simpler to manipulate than generic standard ones
* Dedicated classes derived typically from HostObject
* Wall and Floor classes well-exposed
* No Roof yet

Some properties are accessible via Element base class

Lab 3-5 demonstrates this:

* List all Wall types and remember last one: use Document property WallTypes
* List all Floor types and remember last one
* Change all selected Walls' and Floors' type to the stored ones

Before running this command, draw four walls and a floor and select them all. First, all wall types defined in the current document are listed and the last one is stored for later use. Then all floor types including foundations (slab) defined in the current document are listed and the last one is stored for later use. Finally, all the selected wall and floor types are changed to the stored ones.

#### List all Wall Types

All wall types defined in the current document are listed and the last one is stored for later use.

#### List all Floor Types

All floor types including foundations (slab) defined in the current document are listed and the last one is stored for later use.

#### Change Wall and Floor Type

Finally, all the selected wall and floor types are changed to the stored ones.

## Parameters

The main data container on every Revit building element is its collection of parameters. This section explores these parameters.

### Parameters

* Accessing element parameters
* Built-in versus shared parameters
* Exchanging data with external applications
* Strategy for storing custom per-element data
* Manipulating shared parameters' file and groups
* Hidden parameters
* Strategy for storing per-document (per-model) data
* SDK sample FireRating

We explore different methods of accessing and modifying parameters, exporting them to external applications, importing modified data back in again, and handling shared, hidden, and per-document parameters.

The SDK sample FireRating demonstrates a full parameter manipulation use case by implementing a suite of three commands for adding a new shared ‘fire rating’ parameter to all door objects in the Revit model, exporting the current door fire ratings to an external application, and importing the externally modified values back into the Revit model.

### Accessing Parameters

How do we access element parameters?

* Loop through Element.Parameters
* If built-in parameter, definition or GUID is known, get it from one of the three Element.Parameter overloads

Parameter( parameterId As Parameters.BuiltInParameter )

Parameter( definition As Parameters.Definition )

Parameter( guid As System.Guid )

* Get parameter by localised name by looping until found

LabUtils.GetElemParam()

* A shared parameter GUID can be retrieved from its external definition

LabUtils.SharedParamGUID()

Individual parameters can be accessed in different ways: by localised name, by built-in parameter id, by definition or by GUID. You can also loop over the entire element Parameters collection. Lab 4-1 lists all selected elements' parameters. In addition, it demonstrates retrieving a built-in parameter as well as a parameter by localised name, BuiltInParameter.FAMILY\_BASE\_LEVEL\_OFFSET\_PARAM and "Base Offset" respectively. Select a column to display both of these.

#### Loop all Element Parameters

Loop over the entire Parameters collection of an element:

foreach( Parameter param in elem.Parameters )

{

string paramName = param.Definition.Name;

string paramType = param.StorageType.ToString();

string paramValue = param.AsValueString();

msg += " Name=" + paramName + "; Type=" + paramType

+ "; Value=" + paramValue;

}

#### Access Built-in Parameter

Access a specific built-in element parameter:

parBuiltIn = elem.get\_Parameter( BuiltInParameter.ROOF\_WIDTH\_PARAM );

#### Access Named Parameter

Access a specific named element parameter. This kind of access is language dependent!

public static Parameter GetElemParam( Element elem, string name )

{

ParameterSet parameters = elem.Parameters;

foreach( Parameter parameter in parameters )

{

if( parameter.Definition.Name == name )

{

return parameter;

}

}

return null;

}

### BuiltInParamsChecker

* Most elements have more parameters than those listed in the Parameters collection
* One way to find more is to attempt to retrieve a parameter for each built-in parameters enum value
* This is implemented by the BuiltInParamsChecker and also by RvtMgdDbg 'Built-in Enums Snoop...'
* For instance, every family instance has a Room property represented by the ELEM\_ROOM\_ID built-in parameter, which is not listed in the Parameters collection on a door element

The parameters returned in an element’s Parameters collection sometimes differs from the list of parameters that can be queried individually from the element by using the individual built-in parameter access. Therefore, it is useful to be able to determine all valid built-in parameters for a selected element. This is implemented by the BuiltInParamsChecker utility included in the labs, also described in a devnote or technical solution TS87913 on the ADN site.

For a selected element, it loops through all the built-in parameter enums and tries to obtain a valid parameter for each, printing out the list of valid results.

Currently, the built-in parameters and their meanings are not documented, so you have to find them out by trial and error.

### Exporting Parameters

Frequently, we may want to export Revit BIM data to external applications.

Lab 4-2 demonstrates how to use COM interop in .NET to access e.g. MS Excel from Revit.

* Out-of-process client
* Create a map containing all non-symbol elements which have a valid category, sorted into separate sets for each category
* Start up Excel and create a new workbook
* For each category, add a sheet
* Iterate over all elements in the set for that category and determine all the parameter names used
* Add a title row listing the parameter names, and then one row for each element listing all its parameters

This is achieved in the following steps:

#### Create Map of all Elements

Iterate over all document elements and sort them into separate collections for each category. At the end, we will have separate collections for doors, walls, windows, etc.

#### Launch Excel, Add Workbook

Launch Excel through .NET COM interop, add a new workbook, and add a new sheet for each of the categories we found.

#### Determine all Table Headers

For each category, i.e. worksheet, use the Revit parameter definition names to create the worksheet headers. Loop through all the elements in the current category and determine all parameters for each. 'For each parameter in Parameters' is one way to iterate, the ParameterSetIterator is another.

#### Create Table Headers

Create the table headers in the current sheet.

#### Export Parameters to Excel

Iterate over all the elements in the current category. For each element, iterate over all the parameters we have determined for this category, get their values from the Revit element and write them to the worksheet cells.

### Shared Parameters

In Revit, shared parameters can be used to add per-element data to Revit elements.

This data can be exported and imported to and from third party applications.

Lab 4-3 is very similar to the SDK sample FireRating. It defines three commands:

* Create shared parameter for all doors
* Export shared parameter data to Excel
* Import shared parameter data from Excel

Create a wall with a door in it, at least, before running this command. In addition, you may need to set the full path defined in GetSharedParamsFile(). It is currently "C:\tmp\Labs-4-3-1.txt". After running the second command, show the excel spreadsheet, update the value of "API FireRating", and save the file, noting its name and location. In the third step, reopen the recently saved file. Note that the parameter of the door is updated from the Excel file.

#### Create Shared Parameter

* Get shared parameters definition file

Class Parameters.DefinitionFile

Application.Options.SharedParametersFilename

Application.OpenSharedParameterFile

* Get shared parameters group

Class Autodesk.Revit.Parameters.DefinitionGroup

DefinitionFile.Groups

DefinitionFile.Groups.Create

* Get shared parameters definition

Class Parameters.Definition

DefinitionGroup.Definitions

DefinitionGroup.Definitions.Create

* Create category set for binding to 'Doors'

CategorySet = revitApp.Create.NewCategorySet()

catSet.Insert( doc.Settings.Categories.Item( BuiltInCategory.OST\_Doors ) )

* Bind the parameter

Dim binding As Parameters.Binding = revitApp.Create.NewInstanceBinding( catSet )

doc.ParameterBindings.Insert( fireRatingParamDef, binding )

#### Export Shared Parameter

* Start up Excel
* Get all standard family instances for the given category 'Doors'
* Determine the shared parameter GUID for accessing values

Dim guid As Guid = guid.Empty

Dim file As Parameters.DefinitionFile = revitApp.OpenSharedParameterFile

Dim group As Parameters.DefinitionGroup = file.Groups.Item(defGroup)

Dim definition As Parameters.Definition = group.Definitions.Item(defName)

Dim externalDefinition As Parameters.ExternalDefinition = definition

guid = externalDefinition.GUID

* Loop all doors and export each to an Excel row

id, level, tag, fire rating

worksheet.Cells(row, 1).Value = door.Id.Value

worksheet.Cells(row, 2).Value = door.Level.Name

Dim tagParameter As Parameter = door.Parameter( BuiltInParameter.ALL\_MODEL\_MARK )

worksheet.Cells(row, 3).Value = tagParameter.AsString

Dim parameter As Parameter = door.Parameter( paramGuid )

worksheet.Cells(row, 4).Value = parameter.AsDouble

row = row + 1

#### Import Shared Parameter

* Select file, start up Excel, open file
* Import data

We retrieve the door id and the updated fire rating value from the Excel file, get the door element from Revit, get its shared parameter and update its value.

### Model Shared Parameters

* How do we store per-document (per-model) data using shared parameters?
* Bind hidden shared parameters to the singleton element of 'Project Information' category
* The shared parameter definition uses ExternalDefinition, which has an attribute 'Visible'

## Geometry and Groups

Revit provides full access to the element geometry. The geometrical objects used are transient. They are generated on the fly by the API or an application and used to query or define element properties. They live in a separate namespace Autodesk.Revit.Geometry. We explore this namespace and its functionality in greater depth in the companion session

* **DE115-1** Ten Steps for Enhancing Your Revit® Add-In

### Access 3D Geometry

Revit SDK sample viewers

* RevitViewer

A simple geometry viewer helper class

Used by the other viewer samples

* ElementViewer

Display wireframe model of one or more selected elements

* RoomViewer

Wireframe viewer of the selected room

* AnalyticalViewer

Display analytical model of one or more selected elements

* ObjectViewer

Display analytical or physical model and set parameter of selected elements

* FamilyExplorer

Profiles

#### Obtain 3D Element Geometry

Each element may have geometry associated with it. The element geometry is returned by its Geometry method, which takes an argument specifying the required detail level: Coarse, Medium, or Fine. Here is code from the ElementViewer sample querying each element in the selection set for its geometry and passing it into a drawing routine:

Private mOptions As Autodesk.Revit.Geometry.Options

mOptions = app.Create.NewGeometryOptions

mOptions.DetailLevel = Geometry.Options.DetailLevels.Fine

For Each elem In selSet

Dim geom As Autodesk.Revit.Geometry.Element = elem.Geometry(mOptions)

DrawElement(geom)

Next

If selSet.Size > 0 Then

mViewer.ShowModal()

End If

#### Display Element Geometry

Iterate over the element geometry, which in turn is a Geometry.Element instance containing curve, instance, mesh and solid objects, which in turn are broken down into line segments for the viewer. For more details, please refer to the SDK sample source code.

Private Sub DrawElement(ByVal elementGeom As Geometry.Element)

Dim geomObject As Autodesk.Revit.Geometry.GeometryObject

For Each geomObject In elementGeom.Objects

If (TypeOf geomObject Is Autodesk.Revit.Geometry.Curve) Then

DrawCurve(geomObject)

ElseIf (TypeOf geomObject Is Autodesk.Revit.Geometry.Instance) Then

DrawInstance(geomObject)

ElseIf (TypeOf geomObject Is Autodesk.Revit.Geometry.Mesh) Then

DrawMesh(geomObject)

ElseIf (TypeOf geomObject Is Autodesk.Revit.Geometry.Solid) Then

DrawSolid(geomObject)

End If

Next

End Sub

Private Sub DrawCurve(ByVal geomCurve As Geometry.Curve)

DrawPoints(geomCurve.Tessellate)

End Sub

### Access to 2D Geometry

Rooms also have geometry associated with them. In this case, the geometry is the 2D room boundary lines. This is demonstrated by Lab 5-3:

* Display room id, name and number
* Iterate over room boundary lines
* See also SDK sample RoomViewer

In order to explore lab 5-3, create four walls and a room. The room id, name and number as well as the geometrical coordinates of its boundary lines are displayed. RoomViewer displays the room boundary graphically in its own embedded viewer.

#### Iterate Room Boundary

The room boundary is accessible through the property Boundary, which returns a BoundarySegmentArrayArray, i.e. a collection of arrays of boundary segments. Each array of boundary segments is one boundary, and each boundary segment has a curve, which can be a line or an arc, for instance.

BoundarySegmentArrayArray boundaries = room.Boundary;

int iB = 0;

foreach( BoundarySegmentArray boundary in boundaries )

{

++iB;

sMsg += "\r\n Boundary " + iB + ":";

int iSeg = 0;

foreach( BoundarySegment segment in boundary )

{

++iSeg;

Curve crv = segment.Curve;

if( crv is Line ) // LINE

{

Line line = crv as Line;

XYZ ptS = line.get\_EndPoint( 0 );

XYZ ptE = line.get\_EndPoint( 1 );

sMsg += "\r\n Segment " + iSeg + " is a LINE:"

+ ptS.X + ", " + ptS.Y + ", " + ptS.Z + " ; "

+ ptE.X + ", " + ptE.Y + ", " + ptE.Z;

}

else if( crv is Arc ) // ARC

{

Arc arc = crv as Arc;

XYZ ptS = arc.get\_EndPoint( 0 );

XYZ ptE = arc.get\_EndPoint( 1 );

double r = arc.Radius;

sMsg += "\r\n Segment " + iSeg + " is an ARC:"

+ ptS.X + ", " + ptS.Y + ", " + ptS.Z + " ; "

+ ptE.X + ", " + ptE.Y + ", " + ptE.Z + " ; R=" + r;

}

}

}

### Groups

* Determining types available in the model
* Determining elements' containment
* Manipulating group instance types

#### List Groups and GroupTypes

Determining all the groups and group types in the model is a standard element iteration, checking for the System.Type Group or GroupType. This is demonstrated by Lab 5-1. Before running Lab 5-1, create at least one model group. Pick Edit > Group > Create Group or use the GP shortcut, then add some model elements such as walls and floors.

#### List Model Groups

To list model group types only, perform the same iteration as above. In addition, check each element for the built-in parameter SYMBOL\_FAMILY\_NAME\_PARAM and verify that its value is “Model Group”.

#### Swap Group Type

Swapping the group type of a group is achieved by setting its GroupType property. This is demonstrated in Lab 5-2. Before running Lab 5-2, create at least one model group. Pick Edit > Group > Create Group or use the GP shortcut, then add some model elements such as walls and floors. Then, select an element before running the command.

### Learning More

To learn more, please refer to the additional presentations listed in the introduction. The Revit online help offers a wealth of product related information, and the reference material included in the SDK documentation lists all the classes provided by the API and their methods and properties. The largest Revit programming knowledgebase is provided by the SDK samples. Finally, there are several useful online resources available:

DevTV Introduction to Revit Programming

* <http://adn.autodesk.com/adn/servlet/item?siteID=4814862&id=10194238&linkID=4901650>

Discussion Groups

* [http://discussion.autodesk.com](http://discussion.autodesk.com/)
* Revit API

API Training Classes

* [www.autodesk.com/apitraining](http://www.autodesk.com/apitraining)

Autodesk Developer Network

* [www.autodesk.com/joinadn](http://www.autodesk.com/joinadn)

DevHelp Online for ADN members

* [adn.autodesk.com/](http://www.autodesk.com/joinadn)

We are offering free API training to attendees of our DevTech AU seminars. If you leave your business card during the seminar, you can attend one of our scheduled 2008 API training classes free of charge. Normal costs for these classes are up to 1500 USD per person. The schedule is posted on www.autodesk.com/apitraining. The business cards collected can be given to the presentor, Stephen Preston, or any of our San Rafael based DevTech team.

### The End

Thank you very much for your attention. I wish you much success and enjoyment with your Revit development work.

### Acknowledgement

Miroslav Schonauer of Autodesk Consulting created the original labs in VB.NET. They were further developed and enhanced to include a C# version and html instructions by Jeremy Tammik of Developer Technical Services. Many thanks to Miroslav for making the labs available to the Revit API community!