**Major renovations to the Revit API**

**.NET 4.0 now used by the Revit API**

The Revit API has been enhanced to run with the .NET 4.0 runtime. As a result, Visual Studio 2010 with a framework target of .NET 3.5 or 4.0 must be used to debug your addins. Addins compiled with Visual Studio 2008 will run normally outside the debugging environment.

All samples in the Revit API SDK have been upgraded to Visual Studio 2010.

Microsoft has not yet announced when Visual Studio Tools for Applications (VSTA) for .NET 4.0 will be available. VSTA is the technology used for Revit macros.  The Microsoft VSTA debugger is not compatible with the default Revit 2012 .NET 4.0 environment.  VSTA macros can run successfully in the .NET 4.0 environment, but a special Revit configuration is required for debugging.  In order to use VSTA debugging, you must:

1. Ensure you have user permissions sufficient to modify files in the Revit installation folder.
2. Exit Revit.
3. Run '<Revit.exe directory>\RevitVSTAConfig.exe'
4. Click the 'Prepare Revit for VSTA' button.
5. Restart Revit.

Note: Revit’s native components continue to be compiled with the VS 2008 C++ compiler.  Therefore Revit 2012 does not include the VS2010 redistributable. Third-party applications which include natively compiled C++ components should use the VS 2008 C++ compiler or must include the VS 2010 C++ redistributables with the installation.

**RegenerationMode**

Automatic regeneration mode has been removed. You no longer have to include the RegenerationMode attribute on your add-ins. All APIs expect that add-in code is written using manual regeneration when necessary.

**Add-In registration - new required property VendorId**

Two new registration properties have been added:

* VendorId - a mandatory string conforming to the Autodesk vendor ID standard. Register your vendor id string with Autodesk at <http://www.autodesk.com/symbreg>. An error will be shown if the add-in manifest file does not contain this required node for add-in entry.
* VendorDescription - optional string containing vendor's legal name and/or other pertaining information.

Here is an example of the new entries:

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

<RevitAddIns>

<AddIn Type="Command">

<Assembly>Command.dll</Assembly>

<ClientId>d7e30025-97d4-4012-a581-5f8ed8d18808</ClientId>

<FullClassName>Revit.Command</FullClassName>

<Text>Command</Text>

<VendorId>ADSK</VendorId>

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

</AddIn>

</RevitAddIns>

**CompoundStructure and WallSweeps**

The CompoundStructure class has been replaced. The old CompoundStructure class in the API was read-only and supported only the nominal layers list without support for the settings related to vertical regions, sweeps, and reveals. The new CompoundStructure class supports read and modification of all of the structure information include vertical regions, sweeps and reveals.

The CompoundStructure.Layers property has been replaced by CompoundStructure.GetLayers() and CompoundStructure.SetLayers().

The CompoundStructureLayer class has also been replaced for the same reasons as CompoundStructure.

The following table maps properties from the old CompoundStructureLayer to the new version of the class

|  |  |  |
| --- | --- | --- |
| **Old property** | **New property** | **Notes** |
| DeckProfile | DeckProfileId | Is now an ElementId |
| DeckUsage | DeckEmbeddingType | Uses a different enum. |
| Function | Function | Uses a different enum. |
| Material | MaterialId | Is now an ElementId |
| Thickness | Width |  |
| Variable | N/A | Not a part of the layer class, use CompoundStructure.VariableLayerIndex instead. |

The property HostObjAttributes.CompoundStructure has been replaced by two methods:

* HostObjAttributes.GetCompoundStructure()
* HostObjAttributes.SetCompoundStructure()

Remember that you must set the CompoundStructure back to the HostObjAttributes instance in order for any change to be stored in the element.

In addition to the information on wall sweeps found in the CompoundStructure class, there is a new API representing a wall sweep or reveal. The WallSweep class is an element that represents either a standalone wall sweep/reveal, or one added by the settings in a given wall's compound structure. Standalone wall sweeps and reveals may be constructed using the static method Create().

**LinePattern**

The LinePattern class has been replaced. The old LinePattern class in the API represented both the line pattern itself and the element that contains it, and offered no details on the contents of the pattern beyond its name. The new classes available are:

* LinePatternElement - an element that contains a line pattern
* LinePattern - the line pattern. This class provides access to the pattern name and the list of segments that make up the pattern. The line segments define a repeating pattern of dashes and dots for the line pattern.

The method

* LinePatternElement.Create()

offers the ability to add new LinePattern elements to the Revit database.

The property of the Settings class:

* Settings.LinePatterns

has been removed. LinePatterns may be found by the following approaches:

1. Use of a FilteredElementCollector filtering on class LinePatternElement
2. Use of the static methods of LinePatternElement:
   * LinePatternElement.GetLinePattern(Document, ElementId)
   * LinePatternElement.GetLinePatternByName (Document, string)

**FillPattern**

The FillPattern class has been replaced. The old FillPattern class in the API represented both the fill pattern itself and the element that contains it, and offered no details on the contents of the pattern beyond its name. The new classes available are:

* FillPatternElement - an element that contains a fill pattern
* FillPattern - the fill pattern. This class provides access to the pattern name and the set of grids that make up the pattern.
* FillGrid - a single fill pattern grid, described in the two dimensions of a face.

The method

* FillPatternElement.Create()

offers the ability to add new FillPattern elements to the Revit database.

The property of the Settings class:

* Settings.FillPatterns

has been removed. FillPatterns may be found by the following approaches:

1. Use of a FilteredElementCollector filtering on class FillPatternElement
2. Use of the static methods of FillPatternElement:
   * FillPatternElement.GetFillPattern(Document, ElementId)
   * FillPatternElement.GetFillPatternByName (Document, string)

**IndependentTag**

A good portion of the IndependentTag class and related classes have been renovated.

* Leader has been renamed HasLeader
* LeaderMode has been renamed LeaderEndCondition, and the members of the LeaderEndCondition enum have been renamed
* A new method CanLeaderEndConditionBeAssigned() determines if the LeaderEndCondition can be set.
* TagMode has been replaced by multiple properties: IsMaterialTag, IsMulticategoryTag.
* Members of the TagOrientation enum have been renamed.

Some new members were added to determine the elements referenced by the tag:

* New property TaggedElementId - provides the id of the element referenced by the tag.
* New property TaggedLocalElementId - provides the id of a linked element reference by the tag.
* New method GetTaggedLocalElement() - returns the handle of the element reference by the tag.
* New property IsOrphaned - Orphans are those tags that are associated with an instance of a linked Revit file but have no host element. Tags become orphaned when the element they were tagging was deleted from the link.

**Import and Export APIs changes**

The import and export APIs no longer have special argument combinations which permit use of the "active view". If you wish to export from or import to the active view, you must obtain it directly and pass it as input to the method.

* For the following APIs, a non-empty ViewSet must be provided:
  + bool Export (string folder, string name, ViewSet views, DGNExportOptions options);
  + bool Export (string folder, string name, ViewSet views, DWGExportOptions options);
  + bool Export (string folder, string name, ViewSet views, DXFExportOptions options);
  + bool Export (string folder, string name, ViewSet views, SATExportOptions options);
  + bool Export (string folder, string name, ViewSet views, DWFExportOptions options);
  + bool Export (string folder, string name, ViewSet views, DWFXExportOptions options);
  + bool Export (string folder, string name, ViewSet views, FBXExportOptions options);
* For the following API, a 3D view must be provided:
  + bool Export(string folder, string name, View3D view, ViewPlan grossAreaPlan, BuildingSiteExportOptions options);
* For import APIs, the 'View' property has been removed from the DWGImportOptions and ImageImportOptions. Instead, a new required argument has been added for the following APIs:
  + bool Import (string file, DWGImportOptions options, View view, out Element element);
  + bool Import (string file, ImageImportOptions options, View view, out Element element);
  + bool Link (string file, DWGImportOptions options, View view, out Element element);

The methods

* ACADExportOptions.GetPredefinedSetupNames()
* DWGExportOptions.GetPredefinedOptions()
* DXFExportOptions.GetPredefinedOptions()

provide access predefined setups and settings from a given document for DWG and DFX export.

The method

* Document.Import(String, InventorImportOptions)

has been replaced by

* Application.OpenBuildingComponentDocument(String)

**Save and Close API changes**

* Document.Close()
* Document.Close(bool saveModified)  
  The behavior of these methods has been changed. Previously, they would prompt the interactive user to pick a target path name if the document's path name was not already set, or if the target file was read-only. Now they will throw an exception if the document's path name is not yet set, or if the saving target file is read-only.
* Document.Save()  
  The behavior of this method has been changed. Previously, it would prompt the interactive user to pick a target path name if the document's path name was not already set, or if the target file was read-only. Now it will throw an exception if the document's path name is not set yet. In this case, it needs to be first saved using the SaveAs method instead, or when the target file is read-only.
* Document.Save(SaveOptions options)  
  This new method behaves identically to Document.Save(), but allows you to specify a temporary id to use to generate the file preview.
* Document.SaveAs(string fileName)
* Document.SaveAs(string fileName, bool changeDocumentFilename)  
  The behavior of these methods has been changed. Previously, they would prompt the user before overwriting another file. Now an exception will be thrown in that situation.
* Document.SaveAs(string fileName, SaveAsOptions options)  
  This new method allows you to save a document, and encapsulates the options to rename the document in session, overwrite an existing file (if it exists), and temporarily assign a view id to use to generate the file preview.
* UIDocument.SaveAndClose()  
  This new method closes the document after saving it. If the document's path name has not been set the "Save As" dialog will be shown to the Revit user to set its name and location.
* UIDocument.SaveAs()  
  This new method saves the document to a file name and path obtained from the Revit user (via the "Save As" dialog).

**Reference properties**

The Reference class is being renovated to be more closely aligned with the native Revit class it wraps. Because of this change, References will no longer carry information on the document, element, or geometry object it was obtained from. The following table lists the replacement APIs for the Reference properties:

|  |  |
| --- | --- |
| **Reference property** | **Replacement** |
| .Element | .ElementId or Document.GetElement(Reference) |
| .GeometryObject | Element.GetGeometryObjectFromReference(Reference) |
| .Transform | ReferenceWithContext.GetInstanceTransform() (see below) |
| .ProximityParameter | ReferenceWithContext.ProximityParameter (see below) |
| .ElementReferenceType | Unchanged |
| .GlobalPoint | Unchanged |
| .UVPoint | Unchanged |

The method

* Document.FindReferencesByDirection()

is affected most by this change. The method was the only method to populate the Transform and ProximityParameter properties of Reference. You should switch to the replacement method:

* Document.FindReferencesWithContextByDirection()

This replacement method returns a list of ReferenceWithContext objects, containing the Reference plus additional members:

* ReferenceWithContext.ProximityParameter
* ReferenceWithContext.GetInstanceTransform()

Note that methods which existed in Revit 2011 and earlier will continue in this release to return a fully populated Reference object (it is not necessary to change code dealing with these methods). However, methods added in Revit 2012 may not return a fully populated Reference handle and will need to be parsed using the replacement methods above. Also, affected members have been marked obsolete and planned to be removed in a future release.

**Event argument changes**

A few changes have been made to event argument classes as we align frameworks to generate code for events.

Note: some of these classes and methods may change again before the final release.

The following table lists the changes

|  |  |  |  |
| --- | --- | --- | --- |
| **Member** | **Event argument class** | **New member** | **Notes** |
| Property Cancel | Many event argument classes | Methods Cancel(), IsCancelled() | Now available only in argument classes for pre-events that derive from RevitAPIPreEventArgs |
| Property ImportedInstance | FileImportedEventArgs | Property ImportedInstanceId | Now an ElementId |
| Property PrintedViews | DocumentPrintedEventArgs | Method GetPrintedViewElementIds() |  |
| Property FailedViews | DocumentPrintedEventArgs | Method GetFailedViewElementIds() |  |
| Property Views | DocumentPrintingEventArgs | Method GetViewElementIds() |  |

**Event sender changed for UI events**

For some UI events on UIApplication and UIControlledApplication, the type of the "sender" object has been changed.

* For the ApplicationClosing event, the sender object is now a UIControlledApplication object (it was previously a ControlledApplication);
* For all other UI events, the sender object is now a UIApplication (it was previously an Application).

**Move/Mirror/Rotate/Array changes**

**Move method replacements**

The new static methods

* ElementTransformUtils.MoveElements(Document, ICollection<ElementId>, XYZ)
* ElementTransformUtils.MoveElement(Document, ElementId, XYZ)

replace the Document.Move() overloads.

**Mirror method replacements**

The new static methods

* ElementTransformUtils.MirrorElements(Document, ICollection<ElementId>, Plane)
* ElementTransformUtils.MirrorElement(Document, ElementId, Plane)

replace the Document.Mirror() methods. These new methods allow you to mirror one or more elements directly about a geometric plane which you may construct directly. Previously the Document.Mirror() methods required you to obtain a reference to an actual plane, or implicitly used the active view to calculate the mirror plane.

**Rotate method replacements**

The new static methods

* ElementTransformUtils.RotateElements(Document, ICollection<ElementId>, Line, double)
* ElementTransformUtils.RotateElement(Document, ElementId, Line, double)

replace the Document.Rotate() methods.

**New Copy methods**

The new static methods

* ElementTransformUtils.CopyElements(Document, ICollection<ElementId>, XYZ)
* ElementTransformUtils.CopyElement(Document, ElementId, XYZ)

allow you to copy and translate one or more physical elements and place the newly copied elements by applying the given translation vector.

**LinearArray creation replacement methods**

The new methods

* LinearArray.Create(Document, View, ICollection<ElementId>, int, XYZ, ArrayAnchorMember)
* LinearArray.Create(Document, View, ElementId, int, XYZ, ArrayAnchorMember)

replace the Document.Array() overloads that return LinearArray.

The new methods

* LinearArray.ArrayElementsWithoutAssociation(Document, View, ICollection<ElementId>, int, XYZ, ArrayAnchorMember)
* LinearArray.ArrayElementWithoutAssociation(Document, View, ElementId, int, XYZ, ArrayAnchorMember)

replace the Document.ArrayWithoutAssociate() methods that create sets of elements based on linear translation.

**RadialArray creation replacement methods**

The new methods

* RadialArray.Create(Document, View, ICollection<ElementId>, int Line, double, ArrayAnchorMember)
* RadialArray.Create(Document, View, ElementId, int Line, double, ArrayAnchorMember)

replace the Document.Array() overloads that return RadialArray. The new methods require a view input (where previously the old overloads would use the active view implicitly, which could lead to unexpected failure in some conditions).

The new methods

* RadialArray.ArrayElementsWithoutAssociation(Document, View, ICollection<ElementId>, int, Line, double, ArrayAnchorMember)
* RadialArray.ArrayElementWithoutAssociation(Document, View, ElementId, int, Line, double, ArrayAnchorMember)

replace the Document.ArrayWithoutAssociate() methods that create sets of elements based on rotation.

**Structural AnalyticalModel changes**

**AnalyticalModel now an Element**

The AnalyticalModel class is now a subclass of Element.

One consequence of this change: the AnalyticalModel of a newly created structural element can no longer be obtained without regeneration (as the AnalyticalModel element must be generated).

**AnalyticalModelSelector**

The following two constructors were removed from AnalyticalModelSelector:

* AnalyticalModelSelector(IList<Curve>, int index)
* AnalyticalModelSelector(IList<Curve>, int index, AnalyticalCurveSelector)

The following method was also removed:

* bool IsValidIndex(IList<Curve>, int index)

Instead of passing the array of curves and the index to create the selector, you should use the constructor that accepts the Curve directly (obtain the curve from the collection prior to input).

**SlabFoundationType**

The enumerated value SlabFoundationType.Slab is now SlabFoundationType.SlabOneWay. A new option, SlabTwoWay, is also available. Existing floors assigned the value Slab will report SlabOneWay.

**Major enhancements to the Revit API**

**Extensible Storage**

The Revit API now allows you to create your own class-like Schema data structures and attach instances of them to any Element in a Revit model. This functionality can be used to replace the technique of storing data in hidden shared parameters. Schema-based data is saved with the Revit model and allows for higher-level, metadata-enhanced, object-oriented data structures. Schema data can be configured to be readable and/or writable to all users, just a specific application vendor, or just a specific application from a vendor.

The extensible storage classes are all found in Autodesk.Revit.DB.ExtensibleStorage

* Autodesk.Revit.DB.ExtensibleStorage.SchemaBuilder -- Used to create Schemas definitions
* Autodesk.Revit.DB.ExtensibleStorage.Schema --Contains a unique schema identifier, read/write permissions, and a collection of data Field objects.
* Autodesk.Revit.DB.ExtensibleStorage.Field  -- Contains data name, type, and unit information and is used as the key to access corresponding data in an Element
* Autodesk.Revit.DB.ExtensibleStorage.FieldBuilder -- A helper class used with SchemaBuilder used when creating a new field.
* Autodesk.Revit.DB.ExtensibleStorage.Entity -- An object containing data corresponding to a Schema that can then be inserted into an Element.

The following data types are currently supported:

* int
* short
* double
* float
* bool
* string
* Guid
* ElementId
* Autodesk.Revit.DB.XYZ
* Autodesk.Revit.DB.UV
* Autodesk.Revit.DB.ExtensibleStorage.Entity (An instance of another Schema, also known as a SubSchema)
* Array (as a System.Collections.Generic.IList<T>)
* Map (as a System.Collections.Generic.IDictionary<TKey, TValue>)
* All types supported for simple types, including Entity, are supported for the <TValue> parameter.
* All types supported for simple types \*except\* double, float, XYZ, UV, and Entity are supported for the <TKey> parameter.

Simple usage of ExtensibleStorage

// Create a data structure, attach it to a wall, populate it with data, and retrieve the data back from the wall

public void StoreDataInWall(Wall wall, XYZ dataToStore)

{

Transaction createSchemaAndStoreData = new Transaction(wall.Document, "tCreateAndStore");

createSchemaAndStoreData.Start();

SchemaBuilder schemaBuilder =

new SchemaBuilder(new Guid("720080CB-DA99-40DC-9415-E53F280AA1F0"));

// allow anyone to read the object

schemaBuilder.SetReadAccessLevel(AccessLevel.Public);

// restrict writing to this vendor only

schemaBuilder.SetWriteAccessLevel(AccessLevel.Vendor);

// required because of restricted write-access

schemaBuilder.SetVendorId("ADSK");

// create a field to store an XYZ

FieldBuilder fieldBuilder =

schemaBuilder.AddSimpleField("WireSpliceLocation", typeof(XYZ));

fieldBuilder.SetUnitType(UnitType.UT\_Length);

fieldBuilder.SetDocumentation("A stored location value representing a wiring splice in a wall.");

schemaBuilder.SetSchemaName("WireSpliceLocation");

Schema schema = schemaBuilder.Finish(); // register the Schema object

// create an entity (object) for this schema (class)

Entity entity = new Entity(schema);

// get the field from the schema

Field fieldSpliceLocation = schema.GetField("WireSpliceLocation");

entity.Set<XYZ>(fieldSpliceLocation, dataToStore, DisplayUnitType.DUT\_METERS); // set the value for this entity

wall.SetEntity(entity); // store the entity in the element

// get the data back from the wall

Entity retrievedEntity = wall.GetEntity(schema);

XYZ retrievedData =

retrievedEntity.Get<XYZ>(schema.GetField("WireSpliceLocation"), DisplayUnitType.DUT\_METERS);

createSchemaAndStoreData.Commit();

}

**Worksharing API**

Several new classes were added to provide access to worksharing information in the document:

* Workset - Represents a workset in the document. Worksets are a way to divide a set of elements in the Revit document into subsets for worksharing.
* WorksetId - Identifies a workset within a single document.
* WorksetKind - An enumerated type that indicates one of the standard kinds of workset (as available in the UI).
* WorksetTable - A table containing references to all the worksets contained in a document.
* WorksetVisibility - An enumerated type that indicates the visibility settings of a workset in a particular view.
* WorksetDefaultVisibilitySettings - An object that manages default visibility of worksets in a document.
* FilteredWorksetCollector - This class is used to search, filter and iterate through a set of worksets. Developers can assign a condition to filter the worksets that are returned. If no condition is applied, it attempts to access all the worksets in the document.
* FilteredWorksetIdIterator - An iterator to a set of workset ids filtered by the settings of a FilteredWorksetCollector.
* FilteredWorksetIterator - An iterator to a set of worksets filtered by the settings of a FilteredWorksetCollector.
* WorksetFilter - A base class for a type of filter that accepts or rejects worksets based upon criteria.
* WorksetKindFilter - A filter used to match worksets of the given WorksetKind.
* ElementWorksetFilter - A filter used to match elements which reside in a given workset (use this filter with FilteredElementCollector).
* WorksharingUtils – access to information about a work-shared document.
* WorksharingTooltipInfo – basic read-only information about a work-shared document, such as owner, creator, etc.

Some related additions were made to existing classes:

* Document.GetWorksetTable() - Gets the WorksetTable of this document. There is one WorksetTable for each document.
* Document.GetWorksetId(ElementId id) - Gets the id of the Workset which owns the element.
* Element.WorksetId - Gets the id of the workset which owns the element.
* View.GetWorksetVisibility(WorksetId worksetId) - Returns the visibility settings of a workset for this particular view.
* View.SetWorksetVisibility(WorksetId worksetId, WorksetVisibility visible) - Sets visibility for a workset in this view. This setting overrules implicit visibility of the workset for this particular view.
* View.IsWorksetVisible(WorksetId worksetId) - Indicates whether the workset is visible in this view.

In addition, there is API support for the new 2012 worksharing visualization functionality:

* View.SetWorksharingDisplayMode and View.GetWorksharingDisplayMode allow the API to control which worksharing display mode is enabled in the view.
* WorksharingDisplaySettings allows getting and setting the specific graphic overrides that will be applied in the various worksharing display modes.

**Setting the Active View**

The new property

* UIDocument.ActiveView

has both a getter and setter, so it allows you to query the currently active view of the currently active document, and also allows you to set it similarly to what an end user can do by changing a view in the Project Browser in Revit.  
The setter has a number of limitations:

* It can only be used in an active document, which must not be in read-only state and must not be inside a transaction.
* The setter also cannot be used during ViewActivating and ViewActivated events, or during any pre-action event, such as DocumentSaving, DocumentExporting, or other similar events.

**Opening and activating a document**

A new method:

* UIApplication.OpenAndActivateDocument(String)

was added to the UI API. It opens a Revit document and makes it the active one. The document is opened with its default view displayed.

There are limitations preventing this method to be called at certain situations:

* when there is a transaction open in the currently active document (if there is an active document)
* during execution of any event handler

**Adding a custom Ribbon tab**

The new methods

* UIApplication.CreateRibbonTab()
* UIApplication.CreateRibbonPanel(string, string)
* UIApplication.GetRibbonPanels(string)

(and the corresponding methods in UIControlledApplication) provide the ability to add a new ribbon tab to Revit, at the end of the list of static tabs (to the right of the Add-Ins tab, if shown). If multiple tabs are added, they will be shown in the order added.

There is a limit to the number of custom tabs supported in a given session of Revit (20). This limit is provided to ensure that the standard tabs remain visible and usable. Because of this, your application should only add a custom tab if it's really needed.

**Construction modeling API**

New functionality in Revit 2012 allows elements to be divided into sub-parts, collected into assemblies, and displayed in special assembly views. The API for dividing parts is still under development and likely to change.

Read, write and create access to assemblies in the Revit environment is provided through the classes:

* Autodesk.Revit.DB.Assembly.AssemblyInstance
* Autodesk.Revit.DB.Assembly.AssemblyType

A new assembly containing the selected elements can be created as follows:

ElementId categoryId = doc.get\_Element(uidoc.Selection.GetElementIds().

FirstOrDefault()).Category.Id;

ElementId titleblockId = doc.TitleBlocks.Cast<FamilySymbol>().First<FamilySymbol>().Id;

AssemblyInstance instance = null;

Transaction t = new Transaction(doc);

if (AssemblyInstance.IsValidNamingCategory(doc, categoryId, uidoc.Selection.GetElementIds()))

{

t.SetName("Create Assembly Instance");

t.Start();

instance = AssemblyInstance.Create(doc, uidoc.Selection.GetElementIds(), categoryId);

t.Commit();

t.SetName("Set Assembly Name");

t.Start();

string assemblyName = "Assembly #1";

if (AssemblyInstance.IsValidAssemblyName(doc, assemblyName, categoryId))

{

instance.AssemblyTypeName = assemblyName;

}

t.Commit();

}

Other important methods include AssemblyInstance.GetMemberIds(), AssemblyInstance.SetMemberIds(), and AssemblyInstance.Disassemble().

Assembly Views, that display only the elements in the assembly, are created with the AssemblyViewUtils class such as:

if (instance.AllowsAssemblyViewCreation())

{

ViewSheet viewSheet = AssemblyViewUtils.CreateSheet(doc, instance.Id, titleblockId);

View3D view3d = AssemblyViewUtils.Create3DOrthographic(doc, instance.Id);

ViewSection detailSectionA = AssemblyViewUtils.CreateDetailSection(doc, instance.Id, AssemblyDetailViewOrientation.DetailSectionA);

View materialTakeoff = AssemblyViewUtils.CreateMaterialTakeoff(doc, instance.Id);

View partList = AssemblyViewUtils.CreatePartList(doc, instance.Id);

}

The PartUtils class provides methods to identify Part elements that are created by sub-dividing model elements. These methods describe the relationship between Parts and the elements (such as walls, floors, etc) that are divided to create the Parts.

**DB-level applications**

The add-in framework has been extended to support database-level add-ins. These add-ins should be used when the purpose of your application is to assign events and/or updaters to the Revit session, but not to add anything to the Revit user interface or use APIs from RevitAPIUI.dll.

To implement a DB-level application, implement the methods in the Autodesk.Revit.DB.IExternalDBApplication interface:

* public Result OnStartup(Autodesk.Revit.ApplicationServices.ControlledApplication app)
* public Result OnShutdown(Autodesk.Revit.ApplicationServices.ControlledApplication app)

Within the OnStartup() method you should register events and updaters which your application will respond to during the session.

To register the DB-level application with Revit, add the appropriate registry entry to a manifest file in the Addins folder. A DB-level application has a similar structure as for a UI external application:

<?xml version="1.0" standalone="no"?>

<RevitAddIns>

<AddIn Type="DBApplication">

<Assembly>c:\MyDBLevelApplication\MyDBLevelApplication.dll</Assembly>

<AddInId>DA3D570A-1AB3-4a4b-B09F-8C15DFEC6BF0</AddInId>

<FullClassName>MyCompany.MyDBLevelAddIn</FullClassName>

<Name>My DB-Level AddIn</Name>

</AddIn>

</RevitAddIns>

**Geometry API enhancements**

**Get original geometry of a FamilyInstance**

The new method

* FamilyInstance.GetOriginalGeometry()

returns the original geometry of the instance, before the instance is modified by  
joins, cuts, coping, extensions, or other post-processing.

**Extrusion analysis of a solid**

The utility class

* ExtrusionAnalyzer

allows you to attempt to “fit” a given piece of geometry into the shape of an extrusion. An instance of this class is a single-time use class which should be supplied a solid geometry, a plane, and a direction. The utility will calculate a base boundary parallel to the input plane which is the outer boundary of the shadow cast by the solid onto the input plane and along the extrusion direction.

After the extrusion has been calculated, the class permits a second step analysis to identify all faces from the original geometry which do not align with the faces of the calculated extrusion.

This utility works best for geometry which are at least somewhat “extrusion-like”, for example, the geometry of a wall which may or may not be affected by end joins, floor joins, roof joins, openings cut by windows and doors, or other modifications.

**GeometryCreationUtilities**

The new utility class GeometryCreationUtilities offers the ability to create solid geometry from input curves:

* GeometryCreationUtilities.CreateBlendGeometry()
* GeometryCreationUtilities.CreateExtrusionGeometry()
* GeometryCreationUtilities.CreateRevolvedGeometry()
* GeometryCreationUtilities.CreateSweptGeometry()
* GeometryCreationUtilities.CreateSweptBlendGeometry()

The resulting geometry is not added to the document as a part of any element. However, you may use the created Solid, and its constituent faces and edges, in several ways:

* As the input face(s) to the methods in the Analysis Visualization framework (SpatialFieldManager.AddSpatialFieldPrimitive())
* As the input solid to [finding 3D elements by intersection](#RevitAPIChanges2012-Find3Delementsbyinte)
* As one or more of the inputs to a [Boolean operation](#RevitAPIChanges2012-Booleanoperations)
* As a part of a geometric calculation (using, for example, Face.Project(), Face.Intersect(), or other Face, Solid, and Edge geometry methods)

**Find 3D elements by intersection**

The new element filters:

* ElementIntersectsElementFilter
* ElementIntersectsSolidFilter

pass elements whose actual 3D geometry intersects the 3D geometry of the target object. With ElementIntersectsElementFilter, the target object is another element. The intersection is determined with the same routine used to determine if an interference exists during generation of an InterferenceReport. With ElementIntersectsSolidFilter, the target object is any solid, including one created from scratch using GeometryCreationUtilities.

This example uses a created cylinder centered on the end of a wall to find other walls in close proximity (whether or not they are actually joined):

public ICollection<ElementId> FindWallJoinsAtEndUsingProximity(Wall wall, int end)

{

// Get properties of wall at the end point

LocationCurve wallCurve = wall.Location as LocationCurve;

XYZ endPoint = wallCurve.Curve.get\_EndPoint(end);

double height = wall.get\_Parameter(BuiltInParameter.WALL\_USER\_HEIGHT\_PARAM).AsDouble();

double elevation = wall.Level.Elevation;

// Build cylinder centered at wall end point, extending 3' in diameter

CurveLoop cylinderLoop = new CurveLoop();

XYZ arcCenter = new XYZ(endPoint.X, endPoint.Y, elevation);

Application application = wall.Document.Application;

Arc firstArc = application.Create.NewArc(arcCenter, 1.5, 0, Math.PI, XYZ.BasisX, XYZ.BasisY);

Arc secondArc = application.Create.NewArc(arcCenter, 1.5, Math.PI, 2 \* Math.PI, XYZ.BasisX, XYZ.BasisY);

cylinderLoop.Append(firstArc);

cylinderLoop.Append(secondArc);

List<CurveLoop> singleLoop = new List<CurveLoop>();

singleLoop.Add(cylinderLoop);

Solid proximityCylinder = GeometryCreationUtilities.CreateExtrusionGeometry(singleLoop, XYZ.BasisZ, height);

// Filter walls

FilteredElementCollector proximityCollector = new FilteredElementCollector(wall.Document);

proximityCollector.OfClass(typeof(Wall));

// Exclude the wall itself

List<ElementId> toExclude = new List<ElementId>();

toExclude.Add(wall.Id);

proximityCollector.Excluding(toExclude);

// Filter only elements intersecting our target cylinder

proximityCollector.WherePasses(new ElementIntersectsSolidFilter(proximityCylinder));

// Return matches

return proximityCollector.ToElementIds();

}

**Boolean operations**

The new methods

* BooleanOperationsUtils.ExecuteBooleanOperation()
* BooleanOperationsUtils.ExecuteBooleanOperationModifyingOriginalSolid()

execute a boolean operation combining a pair of solid geometry objects. Options to the method include the operations type: Union, Difference, or Intersect.

The first method takes a copy of the input solids and produces a new solid as a result. Its first argument can be any solid, either obtained directly from a  
Revit element or created via another operation like GeometryCreationUtils.

The second method performs the boolean operation directly on the first input solid. The first input must be a solid which is not obtained directly from a Revit  
element. The property

* GeometryObject.IsElementGeometry

can identify whether the solid is appropriate as input for this method.

In this example, the geometry of intersecting columns and walls is obtained by a Boolean intersection operation. The intersection volume and number of boundary faces is shown in the resulting dialog.

/// <summary>

/// A data structure containing the details of each intersecting wall/column pair.

/// </summary>

struct Intersection

{

public Element Wall;

public Element Column;

public Solid Solid;

}

/// <summary>

/// A collection of all intersections.

/// </summary>

private List<Intersection> m\_allIntersections;

/// <summary>

/// Finds and posts information on wall/column intersections.

/// </summary>

/// <param name="doc">The document.</param>

public void FindIntersectionVolumes(Document doc)

{

// Find all Wall elements.

FilteredElementCollector collector = new FilteredElementCollector(doc);

collector.OfClass(typeof(Wall));

m\_allIntersections = new List<Intersection>();

foreach (Wall wall in collector.OfType<Wall>())

{

// Find all intersecting columns

FilteredElementCollector columnIntersectionCollector = new FilteredElementCollector(doc);

// Columns may be one of two different categories

List<BuiltInCategory> categories = new List<BuiltInCategory>();

categories.Add(BuiltInCategory.OST\_Columns);

categories.Add(BuiltInCategory.OST\_StructuralColumns);

ElementMulticategoryFilter categoryFilter = new ElementMulticategoryFilter(categories);

// Apply intersection filter to find matches

ElementIntersectsElementFilter intersectsFilter = new ElementIntersectsElementFilter(wall);

columnIntersectionCollector.WhereElementIsNotElementType().WherePasses(categoryFilter).WherePasses(intersectsFilter);

foreach (Element element in columnIntersectionCollector)

{

// Store information on intersection

Intersection intersection;

intersection.Wall = wall;

intersection.Column = element;

Solid wallSolid = GetGeometry(wall);

Solid columnSolid = GetGeometry(element);

// Intersect the solid geometry of the two elements

intersection.Solid = BooleanOperationsUtils.ExecuteBooleanOperation(wallSolid, columnSolid, BooleanOperationsType.Intersect);

m\_allIntersections.Add(intersection);

}

}

TaskDialog td = new TaskDialog("Intersection info");

td.MainInstruction = "Intersections found: " + m\_allIntersections.Count;

StringBuilder builder = new StringBuilder();

foreach (Intersection intersection in m\_allIntersections)

{

builder.AppendLine(String.Format("{0} x {1}: volume {2} faces {3}", intersection.Wall.Name, intersection.Column.Name, intersection.Solid.Volume, intersection.Solid.Faces.Size));

}

td.MainContent = builder.ToString();

td.Show();

}

/// <summary>

/// Gets the solid geometry of an element.

/// </summary>

/// <remarks>Makes an assumption that each element consists of only one postive-volume solid, and returns the first one it finds.</remarks>

/// <param name="e"></param>

/// <returns></returns>

private Solid GetGeometry(Element e)

{

GeometryElement geomElem = e.get\_Geometry(new Options());

foreach (GeometryObject geomObj in geomElem.Objects)

{

// Walls and some columns will have a solid directly in its geometry

if (geomObj is Solid)

{

Solid solid = (Solid)geomObj;

if (solid.Volume > 0)

return solid;

}

// Some columns will have a instance pointing to symbol geometry

if (geomObj is GeometryInstance)

{

GeometryInstance geomInst = (GeometryInstance)geomObj;

// Instance geometry is obtained so that the intersection works as

// expected without requiring transformation

GeometryElement instElem = geomInst.GetInstanceGeometry();

foreach (GeometryObject instObj in instElem.Objects)

{

if (instObj is Solid)

{

Solid solid = (Solid)instObj;

if (solid.Volume > 0)

return solid;

}

}

}

}

return null;

}

**HostObject - top, bottom, side faces**

The new methods

* HostObjectUtils.GetTopFaces()
* HostObjectUtils.GetBottomFaces()
* HostObjectUtils.GetSideFaces()

provide a shortcut to locate the faces of a given roof, floor, or wall which act as the exterior or interior boundary of the object's CompoundStructure. Top and bottom faces are applicable to roofs and floors. Side faces are applicable to walls.

**Get host face of a FamilyInstance**

The property

* FamilyInstance.HostFace

gets the reference to the host face of family instance, or if the instance is placed on a work plane, the reference to the geometry face underlying the work plane.

**Element.Geometry**

This property has been modified to throw an exception when attempting to read the geometry of a newly added or modified element, if the element has not yet been regenerated. Call Document.Regenerate() to regenerate the affected element and to be able to obtain the geometry.

This property now returns results for ElementType subclasses which own geometry.

**GeometryObject.GraphicsStyleId**

This new property allows access to the GraphicsStyle and Category of individual family primitives to be determined while working in the project environment.

**Curve representation of an Edge**

The methods

* Edge.AsCurve()
* Edge.AsCurveFollowingFace(Face faceForDir)

provide a curve that corresponds to the edge (either oriented along the edge's parametric direction, or oriented in the edge's topological direction on the specified face).

**Centroid of a Solid**

The method

* Solid.ComputeCentroid

calculates the centroid of the solid using an approximation, with an accuracy suitable for architectural purposes. This will correspond only with the center of gravity if the solid represents a homogeneous structure of a single material.

**Transforming geometry**

The new method

* GeometryElement.GetTransformed()

returns a copy of the geometry in the original element, transformed by the input coordinate transformation.

**Instance.GetTransform() and Instance.GetTotalTransform()**

These new methods provide the transform for a given Instance (which is the parent class for elements like family instances, link instances, and imported CAD content). GetTransform() obtains the basic transform for the instance based on how the instance is placed. GetTotalTransform() provides the transform modified with the true north transform, for instances like import instances.

**Serialization/deserialization of References**

The new methods

* Reference.ConvertToStableRepresentation()
* Reference.ParseFromStableRepresentation()

allow you to save a reference to a geometry object, for example a face, edge, or curve, as a string, and to obtain an identical Reference later using the String as input.

**Face.HasRegions & Face.GetRegions()**

This property and method provide information about the faces created by the Split Face command. HasRegions returns a boolean indicating if the face has any Split Face regions. GetRegions returns a list of faces. As the material of these faces can be independently modified through the UI with the Paint tool, the material of each face can be found from its MaterialElementId property.

**Face.MaterialElementId replaces Face.MaterialElement**

Face.MaterialElement has been obsoleted. Face.MaterialElementId should be used instead.  
In some cases the return face will not have access to its Document, so the MaterialElement cannot be returned, but the id can.

**NewHermiteSpline tangency control**

A new overload takes optional XYZ inputs for the tangency at the start and end of the spline.

**New NewNurbSpline overload**

The new overload

* Autodesk.Revit.Creation.Application.NewNurbSpline(IList<XYZ> controlPoints, IList<double> weights)

creates a NurbSpline in an identical manner to how the spline sketching tool creates such a curve in the Revit UI. Remaining fields in the curve definition (Knots, Degree) are automatically assigned from the calculation.

**PolyLine returned from Element.Geometry**

A new geometry object called a PolyLine is exposed through the API. The PolyLine represents a set of coordinate points forming contiguous line segments. Typically this type of geometry would be seen in geometry imported from other formats (such as DWG). Previous Element.Geometry[] would skip extraction of these geometry object completely.

**Analysis of Room and Space 3D geometry**

The new method:

* SpatialElementGeometryCalculator.CalculateSpatialElementGeometry()

computes the 3D geometry of the input spatial element (room or space) and returns it, along with information about the elements which form the boundary of the element.

The new classes:

* SpatialElementGeometryResults
* SpatialElementBoundarySubface

encapsulate the results of the geometric calculation.

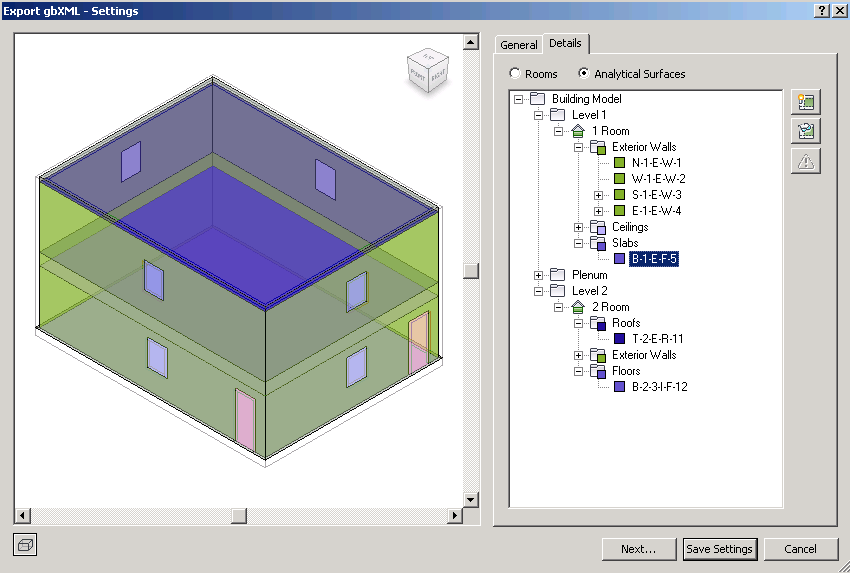
The class:

* SpatialElementBoundaryOptions

provides the available options for the calculation (currently limited to an option to calculate the boundaries at the finish face or at the boundary object's centerlines and whether to include the free boundary faces in the calculation result).

**Detailed energy analysis model API**

A new API is provided to obtain and analyze the contents of a project's detailed energy analysis model, as seen in the Export to gbXML and the Heating and Cooling Loads features:



This analysis produces an analytical thermal model from the physical model of a building. The analytical thermal model is composed of spaces, zones and planar surfaces that represent the actual volumetric elements of the building.

The new classes in Autodesk.Revit.DB.Analysis namespace:

* EnergyAnalysisDetailModel
* EnergyAnalysisDetailModelOptions
* EnergyAnalysisOpening
* EnergyAnalysisSpace
* EnergyAnalysisSurface
* Polyloop

can be used to generate and analyze the contents of the detailed energy analysis model. Use

* EnergyAnalysisDetailModel.Create()

to create and populate the model (while setting up appropriate options); use

* EnergyAnalysisDetailModel.GetAnalyticalSpace()
* EnergyAnalysisDetailModel.GetAnalyticalSurfaces()
* EnergyAnalysisDetailModel.GetAnalyticalOpenings()
* EnergyAnalysisDetailModel.GetAnalyticalShadingSurfaces()

to extract the entities from the analysis; and use

* EnergyAnalysisDetailModel.Destroy()

to clean up the Revit database after finishing with the analysis results.

**Conceptual energy analysis API**

The new classes in the Autodesk.Revit.DB.Analysis namespace:

* ConceptualConstructionType
* ConceptualSurfaceType
* MassEnergyAnalyticalModel
* MassLevelData
* MassSurfaceData
* MassZone

provide access to the elements and objects created by Revit to perform energy analyses on conceptual design models.

The method

* Document.Export(string,string,MassGBXMLExportOptions)

supports export of a gBXML file containing conceptual energy analysis elements (mass elements) only.

**Analysis visualization framework**

The analysis visualization framework was improved to support multiple analysis results shown in the same view at the same time.

The new class

* AnalysisResultSchema

was added to store meta-data for each analysis result. The Results Visibilty view frame control in the user interface and the API's SpatialFieldManager.ResultsEnabledInView and AnalysisResultSchema.IsVisible properties control which results (if any) are displayed.

The class SpatialFieldManager now has new methods:

* RegisterResult()
* GetResultSchema()
* SetResultSchema()
* GetRegisteredResults()

to register and access results meta-data. Corresponding methods and properties of SpatialFieldManager:

* SetUnits()
* CurrentUnits
* GetDescription()
* SetDescription()

are deprecated. They can be used if SpatialFieldManager contains only one analysis result, but they throw an exception if multiple results are registered.

The method

* UpdateSpatialFieldPrimitive()

is changed to take a result index as an additional argument; the old version is deprecated and can be used if SpatialFieldManager contains only one analysis result.

New classes allow for different types of analysis data and different appearance of results:

* AnalysisDisplayDiagramSettings
* AnalysisDisplayVectorSettings

// Create a SpatialFieldManager for the active view

SpatialFieldManager sfm = SpatialFieldManager.CreateSpatialFieldManager(doc.ActiveView, 1);

int primitiveIndex = sfm.AddSpatialFieldPrimitive();

// This example creates two result schema. Each schema contains a single value at the origin.

IList<XYZ> pts = new List<XYZ>();

pts.Add(XYZ.Zero);

FieldDomainPointsByXYZ pnts = new FieldDomainPointsByXYZ(pts);

// Create the schema

AnalysisResultSchema resultSchemaA = new AnalysisResultSchema("Schema A", "Time");

AnalysisResultSchema resultSchemaB = new AnalysisResultSchema("Schema B", "Distance");

// Data in Schema A measures time and can be displayed using Hours or Minutes for units

List<string> unitsList = new List<string>();

unitsList.Add("Hours");

unitsList.Add("Minutes");

List<double> unitsMultipliers = new List<double>();

unitsMultipliers.Add(1);

unitsMultipliers.Add(60);

resultSchemaA.SetUnits(unitsList, unitsMultipliers);

List<double> doubleList = new List<double>();

// The data value in Schema A is 3.5 hours

doubleList.Add(3.5);

IList<ValueAtPoint> valueList = new List<ValueAtPoint>();

valueList.Add(new ValueAtPoint(doubleList));

FieldValues fieldValuesA = new FieldValues(valueList);

// Data in Schema B measures distance and can be displayed using Feet or Inches for units

unitsList.Clear();

unitsMultipliers.Clear();

unitsList.Add("Feet");

unitsList.Add("Inches");

unitsMultipliers.Add(1);

unitsMultipliers.Add(12);

resultSchemaB.SetUnits(unitsList, unitsMultipliers);

doubleList.Clear();

valueList.Clear();

// The data value in Schema B is 5 feet

doubleList.Add(5);

valueList.Add(new ValueAtPoint(doubleList));

FieldValues fieldValuesB = new FieldValues(valueList);

// Update the view's spatial field primitive with the schema

sfm.UpdateSpatialFieldPrimitive(primitiveIndex, pnts, fieldValuesA, sfm.RegisterResult(resultSchemaA));

sfm.UpdateSpatialFieldPrimitive(primitiveIndex, pnts, fieldValuesB, sfm.RegisterResult(resultSchemaB));

**Point Cloud API**

Revit offers two sets of APIs related to Point Clouds.

The client API is capable of working with point cloud instances within Revit (creating them, manipulating their properties, and reading the points found matching a given volumetric filter).

The major classes of the client API are:

* PointCloudType - a type of a point cloud, representing the points obtained from a single file or engine's identifier.
* PointCloudInstance - an instance of a point cloud in a location in the Revit project.
* PointCloudFilter - a filter determining the volume of interest when extracting points.
* PointCollection - a collection of points obtained from an instance and a filter.
* PointIterator - an iterator for the points in a PointCollection.
* CloudPoint - an individual point cloud point, representing an X, Y, Z location in the coordinates of the cloud, and a color.

There are two methods to access the points as a client:

1. In the traditional IEnumerable interface, you can iterate the resulting points directly from the PointCollection.
2. In an unsafe interface usable only from C# and C++/CLI, you can get a pointer to the point storage of the collection and access the points directly in memory. Although you must deal with pointers directly, there may be performance improvements when traversing large buffers of points.

The following snippets show how to iterate part of a point cloud using both methods. The same point cloud filter is applied for both routines:

**Reading point cloud points by iteration**

private int ReadPointCloud\_Iteration(PointCloudInstance pcInstance)

{

PointCloudFilter filter = CreatePointCloudFilter(pcInstance.Document.Application, pcInstance);

// Get points. Number of points is determined by the needs of the client

PointCollection points = pcInstance.GetPoints(filter, 10000);

int numberOfPoints = 0;

foreach (CloudPoint point in points)

{

// Process each point

System.Drawing.Color color = System.Drawing.ColorTranslator.FromWin32(point.Color);

String pointDescription = String.Format("({0}, {1}, {2}, {3}", point.X, point.Y, point.Z, color.ToString());

numberOfPoints++;

}

return numberOfPoints;

}

**Reading point cloud points by pointer**

public unsafe int ReadPointCloud\_Pointer(PointCloudInstance pcInstance)

{

PointCloudFilter filter = CreatePointCloudFilter(pcInstance.Document.Application, pcInstance);

// Get points. Number of points is determined by the needs of the client

PointCollection points = pcInstance.GetPoints(filter, 10000);

int totalCount = points.Count;

CloudPoint\* pointBuffer = (CloudPoint\*)points.GetPointBufferPointer().ToPointer();

for (int numberOfPoints = 0; numberOfPoints < totalCount; numberOfPoints++)

{

CloudPoint point = \*(pointBuffer + numberOfPoints);

// Process each point

System.Drawing.Color color = System.Drawing.ColorTranslator.FromWin32(point.Color);

String pointDescription = String.Format("({0}, {1}, {2}, {3}", point.X, point.Y, point.Z, color.ToString());

}

return totalCount;

}

**Point cloud filter creation**

private PointCloudFilter CreatePointCloudFilter(Application app, PointCloudInstance pcInstance)

{

// Filter will match 1/8 of the overall point cloud

// Use the bounding box (filter coordinates are in the coordinates of the model)

BoundingBoxXYZ boundingBox = pcInstance.get\_BoundingBox(null);

List<Plane> planes = new List<Plane>();

XYZ midpoint = (boundingBox.Min + boundingBox.Max) / 2.0;

// X boundaries

planes.Add(app.Create.NewPlane(XYZ.BasisX, boundingBox.Min));

planes.Add(app.Create.NewPlane(-XYZ.BasisX, midpoint));

// Y boundaries

planes.Add(app.Create.NewPlane(XYZ.BasisY, boundingBox.Min));

planes.Add(app.Create.NewPlane(-XYZ.BasisY, midpoint));

// Z boundaries

planes.Add(app.Create.NewPlane(XYZ.BasisZ, boundingBox.Min));

planes.Add(app.Create.NewPlane(-XYZ.BasisZ, midpoint));

// Create filter

PointCloudFilter filter = PointCloudFilterFactory.CreateMultiPlaneFilter(planes);

return filter;

}

There are two special API-only tools intended to help your client application interact with the user:

1. The SetSelectionFilter() method and FilterAction property of PointCloudInstance allow you to specify a volumetric filter to be applied to the cloud. The parts of the cloud that pass this filter will be rendered differently in the user interface than the rest of the cloud. If the FilterAction is Highlight, the selected part of the cloud will show in highlight color (blue). If the action is Isolate, only the selected part of the cloud will be visible.
2. The overloaded method Selection.PickBox() invokes a general purpose two-click editor that lets the user to specify a rectagular area on the screen. While this editor makes no changes in Revit as a result of the selections, you can use the returned box to generate a filter and apply a highlight or isolate action to the point cloud.

This example prompts the user to select a portion of the cloud, and creates a highlight filter for it.

**Prompt for cloud selection and highlight**

public void PromptForPointCloudSelection(UIDocument uiDoc, PointCloudInstance pcInstance)

{

Application app = uiDoc.Application.Application;

Selection currentSel = uiDoc.Selection;

PickedBox pickedBox = currentSel.PickBox(PickBoxStyle.Enclosing, "Select region of cloud for highlighting");

XYZ min = pickedBox.Min;

XYZ max = pickedBox.Max;

//Transform points into filter

View view = uiDoc.ActiveView;

XYZ right = view.RightDirection;

XYZ up = view.UpDirection;

List<Plane> planes = new List<Plane>();

// X boundaries

bool directionCorrect = IsPointAbovePlane(right, min, max);

planes.Add(app.Create.NewPlane(right, directionCorrect ? min : max));

planes.Add(app.Create.NewPlane(-right, directionCorrect ? max : min));

// Y boundaries

directionCorrect = IsPointAbovePlane(up, min, max);

planes.Add(app.Create.NewPlane(up, directionCorrect ? min : max));

planes.Add(app.Create.NewPlane(-up, directionCorrect ? max : min));

// Create filter

PointCloudFilter filter = PointCloudFilterFactory.CreateMultiPlaneFilter(planes);

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

t.Start();

pcInstance.SetSelectionFilter(filter);

pcInstance.FilterAction = SelectionFilterAction.Highlight;

t.Commit();

}

private static bool IsPointAbovePlane(XYZ normal, XYZ planePoint, XYZ point)

{

XYZ difference = point - planePoint;

difference = difference.Normalize();

double dotProduct = difference.DotProduct(normal);

return dotProduct > 0;

}

The engine API is capable of supplying points in a point cloud to Revit. A custom engine implementation consists of the following:

* An implementation of IPointCloudEngine registered to Revit via the PointCloudEngineRegistry.
* An implementation of IPointCloudAccess coded to respond to inquiries from Revit regarding the properties of a single point cloud.
* An implementation of IPointSetIterator code to return sets of points to Revit when requested.

Engine implementations may be file-based or non-file-based:

* File-based implementations require that each point cloud be mapped to a single file on disk. Revit will allow users to create new point cloud instances in a document directly by selecting point cloud files whose extension matches the engine identifier. These files are treated as external links in Revit and may be reloaded and remapped when necessary from the Manage Links dialog.
* Non-file-based engine implementations may obtain point clouds from anywhere (e.g. from a database, from a server, or from one part of a larger aggregate file). Because there is no file that the user may select, Revit's user interface will not allow a user to create a point cloud of this type. The engine provider should supply a custom command using PointCloudType.Create() and PointCloudInstance.Create() to create and place point clouds of this type. The Manage Links dialog will show the point clouds of this type, but since there is no file associated to the point cloud, the user cannot manage, reload or remap point clouds of this type.

Regardless of the type of engine used, the implementation must supply enough information to Revit to display the contents of the point cloud. There are two ReadPoints methods which must be implemented:

* IPointCloudAccess.ReadPoints() - this provides a single set of points in a one-time call from Revit. Revit uses this during some display activities including selection prehighlighting. It is also possible for API clients to call this method directly (via PointCloudInstance.GetPoints()).
* IPointSetIterator.ReadPoints() - this provides a subset of points as a part of a larger iteration of points in the cloud. Revit uses this method during normal display of the point cloud; quantities of points will be requested repeatedly until it obtains enough points or until something in the display changes. The engine implementation must keep track of which points have been returned to Revit during any given point set iteration.

**Material API changes and PropertySets**

The Revit Materials API is largely renovated to allow for a representation of materials that is both more compact and more extensible.

The following classes are now obsolete:

* MaterialWood
* MaterialConcrete
* MaterialSteel
* MaterialGeneric
* MaterialOther

The properties on these classes are now accessible from PropertySets or the Material class itself.  
All named properties on Material specific to appearance or structure (e.g. Shininess) are also obsolete.

In their place, a material will have one or more *aspects* pertaining to rendering appearance, structure, or other major material category. Each aspect is represented by a PropertySet or PropertySetElement. Each material can own its properties of an aspect via a PropertySet or share them with other materials as a reference to a PropertySetElement.

New enumerated types:

* MaterialAspect - An enumeration of material aspects. Currently, we support rendering and structural aspects.

New classes:

* PropertySetElement - an Element containing a PropertySet – used for sharing PropertySets among materials.

New methods:

* Material.Create()
* PropertySetElement.Create()
* Material.SetMaterialAspectByPropertySet()
* Material.SetMaterialAspectIndependent()
* Material.GetMaterialAspectPropertySet()

**Performance Adviser**

The new Revit feature Performance adviser is designed to analyze the document and flag for the user any elements and/or settings that may cause performance degradation. The Performance Adviser command executes set of rules and displays their result in a standard review warnings dialog.

The API for performance adviser consists of 2 classes (PerformanceAdviser and IPerformanceAdviserRule). PerformanceAdviser is an application-wide singleton that has a dual role: it is a registry of performance checking rules and an engine to execute them. The methods of PerformanceAdviser:

* AddRule()
* DeleteRule()
* GetNumberOfRules()
* GetRuleIDs()
* GetRuleName()
* GetRuleDescription()
* ExecuteRules()
* ExecuteAllRules()

allow you to manipulate what rules are checked. Applications that create new rules are expected to use AddRule() to register the new rule during application startup and DeleteRule() to deregister it during application shutdown.

Methods of PerformanceAdviser

* SetRuleEnabled
* IsRuleEnabled
* ExecuteRules

allow UI or API application to mark rules for execution and run them on a given document, getting report as a list of FailureMessage objects.

The new interface IPerformanceAdviserRule allows you to define new rules for the Performance Adviser. Your application should create a class implementing this interface, instantiate an object of the derived class and register it using PerformanceAdviser.AddRule(). Methods of IPerformanceAdviserRule available for override include:

* GetName()
* GetDescription()

which provide rule identification information;

* InitCheck()
* FinalizeCheck()

which are executed once per check and can be used to perform checks of the document "as a whole";

* WillCheckElements()
* GetElementFilter()
* ExecuteElementCheck()

which allow the rule to identify a subset of elements in the document to be checked and run the check on the individual elements.

Potentially problematic results found during rule execution are reported by returning FailureMessage(s).

**External File References (Linked Files)**

The API can now tell what elements in Revit are references to external files, and can make some modifications to where Revit loads external files from.

An Element which contains an ExternalFileReference is an element which refers to some external file (ie. a file other than the main .rvt file of the project.) Two new Element methods, IsExternalFileReference() and GetExternalFileReference(), let you get the ExternalFileReference for a given Element.

ExternalFileReference contains methods for getting the path of the external file, the type of the external file, and whether the file was loaded, unloaded, not found, etc. the last time the main .rvt file was opened.

The classes RevitLinkType and CADLinkType can have IsExternalFileReference() return true. RevitLinkTypes refer to Revit files linked into another Revit project. CADLinkTypes refer to DWG files. Note that CADLinkTypes can also refer to DWG imports, which are not external file references, as imports are brought completely into Revit. A property IsImport exists to let users distinguish between these two types.

Additionally, the element which contains the location of the keynote file is an external file reference, although it has not been exposed as a separate class.

There is also a class ExternalFileUtils, which provides a method for getting all Elements in a document which are references to external files.

Additionally, the classes ModelPath and ModelPathUtils have been exposed. ModelPaths can store paths to a location on disk, a network drive, or a Revit Server location. ModelPathUtils provides methods for converting between modelPath and String.

Finally, the class TransmissionData allows users to examine the external file data in a closed Revit document, and to modify path and load-state information for that data. Two methods, ReadTransmissionData, and WriteTransmissionData, are provided. With WriteTransmissionData, users can change the path from which to load a given link, and can change links from loaded to unloaded and vice versa. (WriteTransmissionData *cannot* be used to add links to or remove links from a document, however.)

Newly exposed classes:

* ExternalFileReference - A non-Element class which contains path and type information for a single external file which a Revit project references. ExternalFileReference also contains information about whether the external file was loaded or unloaded the last time the associated Revit project was opened.
* ExternalFileUtils - A utility class which allows the user to find all external file references, get the external file reference from an element, or tell whether an element is an external file reference.
* RevitLinkType - An element representing a Revit file linked into a Revit project.
* CADLinkType - An element representing a DWG drawing. CADLinkTypes may be links, which maintain a relationship with the file they originally came from, or imports, which do not maintain a relationship. The property IsImport will distinguish between the two kinds.
* LinkType - The base class of RevitLinkType and CADLinkType.
* ModelPath - A non-Element class which contains path information for a file (not necessarily a .rvt file.) Paths can be to a location on a local or network drive, or to a Revit Server location.
* ModelPathUtils - A utility class which provides methods for converting between strings and ModelPaths.
* TransmissionData - A class which stores information about all of the external file references in a document. The TransmissionData for a Revit project can be read without opening the document.

**Customizing IFC export**

The classes

* ExporterIFCRegistry
* IExporterIFC

allow your custom application to override the default implementation for the IFC export process.

The interface is passed an ExporterIFC object. The ExporterIFC object is the starting point for all IFC export activities and offers access to the options selected for the given export operation. It also contains access to information cached by Revit during the export process; this information is cached to provide easy access to it later, and sometimes to write it to the file at the end of the export process.

There are several auxiliary objects provided to support implementation of an IFC export client. These are the most important:

* IFCFile - a representation of the IFC file being written. This class contains methods which directly generate handles to IFC entries and write them to the file.
* IFCAnyHandle - a wrapper around any sort of IFC element, product, or other data.
* IFCLabel - a string in an IFC file.
* IFCMeasureValue – a parameterized value in an IFC file.
* ExporterIFCUtils - a collection of utilities to support the Revit IFC export client.

The IFC export client for Revit 2012 represents a transitional state between the version implemented internally in Revit 2011 and the final state which should be written 100% using the Revit API. Temporary APIs are exposed to bridge the gaps between the API client code and portions of the previous implementation. As Autodesk continues to evolve this export client, temporary APIs will be changed, deprecated and/or removed in future versions of Revit.

**MEP API major enhancements**

**Pipe settings and sizes**

The following new classes provide read/write access to MEP pipe settings:

* PipeSettings - The pipe settings. There is one object of this type in a document accessible through the static method GetPipeSettings().
* PipeSizeSettings - The main class to access pipe sizes. There is one object of this type in a document accessible through the static method GetPipeSizeSettings(). Stores PipeConnections per each pipe material, using the id of the pipe material element as key to access.
* PipeSizeSettingIterator - An iterator to a set of items from PipeSizeSettings. Each item is a KeyValuePair<ElementId, PipeConnections>.
* PipeConnections - Represents a set of pipe connection types. Stores PipeSchedules per each pipe connection type, using the id of the pipe connection type as key to access.
* PipeConnectionIterator - An iterator to a set of items from PipeConnections. Each item is a KeyValuePair<ElementId, PipeSchedules>.
* PipeSchedules - Represents a set of pipe schedules. Stores PipeSizes per each pipe schedule type, using the id of the pipe schedule type as key to access.
* PipeScheduleIterator- An iterator to a set of items from PipeSchedules. Each item is a KeyValuePair<ElementId, PipeSizes>.
* PipeSizes- Stores a set of pipe sizes, with the ability to add and remove from the set as needed.
* PipeSizeIterator - An iterator to a set of MEP pipe sizes from PipeSizes.
* MEPSize - Stores the basic size information for an MEP duct, pipe, cable tray, or conduit.

**Placeholder ducts and pipes**

The following new properties identify placeholder ducts and pipes:

* Duct.IsPlaceholder
* Pipe.IsPlaceholder

These new static methods allows creation of placeholder ducts and pipes:

* Duct.CreatePlaceholder
* Pipe.CreatePlaceholder

New utility methods are exposed to convert a set of placeholder ducts and pipes to real 3D entities:

* MechanicalUtils.ConvertDuctPlaceholders()
* PlumbingUtils.ConvertPipePlaceholders()

**Duct & pipe insulation & lining**

The new classes

* DuctInsulation
* PipeInsulation
* DuctLining

and related types support read/write and create access to duct & pipe insulation and lining. In Revit 2012, these objects are now accessible as standalone elements related to their parent duct, pipe, or fitting.

**Small enhancements & API interface changes**

**SiteLocation and City TimeZone**

The properties

* SiteLocation.Latitude
* SiteLocation.Longitude

now use Revit's TimeZone calculation engine to assign an appropriate time zone for the coordinates. (Previously the time zone was not modified when these values were changed).  
SiteLocation retains the ability to set the TimeZone manually as the calculation is may not be accurate for locations near the boundaries.

The City class has been adjusted to return the more accurate TimeZone values.

The new static method

* SunAndShadowSettings.CalculateTimeZone()

provides direct access to the results of a time zone calculation.

**FamilyParameter.GUID property**

Returns the GUID of a particular family parameter. Allows you to determine if family parameters are shared or not.

**InternalDefintion.Visible property**

Identifies if a parameter definition represents a visible or hidden parameter (hidden applies to shared parameters only).

**Selection.GetElementIds() method**

Returns a collection containing the ids of the selected elements. This collection can be used directly with FilteredElementCollector.

**API to prompt for rubber band box**

* PickBoxStyle- A new enum that controls the style of the pick box.
* PickedBox-A new class that contains two XYZ points representing the pick box on the screen.
* The following two new methods are added to Selection class that invoke a general purpose two-click editor that lets the user to specify a rectangular area on the screen:  
  PickedBox Selection.PickBox(PickBoxStyle style);  
  PickedBox Selection.PickBox(PickBoxStyle style, String statusPrompt)

**Save and SaveAs APIs permitted in most event handlers**

The limitation against calling Save/SaveAs has been removed for most events.

The restriction against calling Save/SaveAs remains only in the following special events:

* DocumentSaving
* DocumentSaved
* DocumentSavingAs
* DocumentSavedAs
* DocumentSynchronizingWithCentral
* DocumentSynchronizedWithCentral
* FileExporting
* FileImporting
* DocumentPrinting
* ViewPrinting
* DocumentClosing

Please note that other restrictions may still prevent a successful save (e.g. save cannot be performed if a transaction group or transaction is still opened by the API client or via Revit's UI when the event handler is invoked)

**Opening IFC Documents**

A new method allows opening an IFC Document. This method is similar in behavior to OpenDocumentFile rather than to standard Import methods. It opens the specified file as a newly created document rather than importing it into an existing one. The new document is retuned by the method if opening was successful.

* Application.OpenIFCDocument(String fileName)

**RevitUIFamilyLoadOptions**

The class RevitUIFamilyLoadOptions is no longer available for direct construction in the API. If you want to trigger the Revit UI to respond to situations when a loaded family is already found in the target project, obtain a special IFamilyLoadOptions instance from the new static method UIDocument.GetRevitUIFamilyLoadOptions() (in RevitAPIUI.dll) instead.

**Reference documentation for BuiltInParameter members**

The members of the BuiltInParameter enum (which are parameter "ids" for Revit use) now have automatically generated documentation. The documentation for each id includes the parameter name, as found in the Element Properties dialog in the English version of Autodesk Revit. Note that multiple distinct parameter ids may map to the same English name; in those case you must examine the parameters associated with a specific element to determine which parameter id to use.

**SolidSolidCutUtils.CanTwoElemsHaveSolidSolidCut() method**

This method has been removed and replaced by

* SolidSolidCutUtils.CanElementCutElement()

The new method provides a reason why the cutting element cannot cut the other element.

**Adaptive component API**

The methods in the new classes

* AdaptiveComponentFamilyUtils
* AdaptiveComponentInstanceUtils

provide access to data related to adaptive component families and instances.

**ViewSheet.ConvertToRealSheet()**

This new method converts a placeholder sheet to a real view sheet, optionally applying a titleblock at the same time.

**Initial View Settings**

A new class

* IntitialViewSettings

allows Access to the initial view settings for a document that controls which view should initially be shown when the model is opened.

It has the following public methods/properties:

* GetInitialViewSettings – Returns the initial view settings for the specified document.
* ViewId – Returns (if set) or sets the Id of an initial view
* IsAcceptableInitialView – Checks whether the given view is acceptable as an initial view

**Document preview**

The new method

* Document.GetDocumentPreviewSettings()

gets the settings related to the stored document preview image for a given document. It returns a DocumentPreviewSettings object, whose members include:

* DocumentPreviewSettings.PreviewViewId - the id of the view to use for the preview. This value is stored in the document and used in subsequent save operations.
* DocumentPreviewSettings.ForceViewUpdate() - sets the document to update the view before saving (useful when the document is never displayed)

Note that it is also possible to temporarily assign a preview view id for one save operation through the SaveOptions and SaveAsOptions classes. The id set to these classes is not stored in the saved documented.

**Document identification**

An override for

* Document.Equals()

was added to determine if two Documents represent the same document currently opened in the Revit session.

An override for

* Document.GetHashCode()

was added to return the same hashcode for document instances that represent the same document currently opened in the Revit session.

**Dynamic Update Framework API changes**

There are new settings to flag an updater as optional. Optional updaters will not cause prompting the end user when they are opening a document which was modified by that updater but the updater is not currently registered. red). Optional updaters should be used only when necessary. By default, updaters are non-optional. New methods introduced to support this change are:

* UpdaterRegistry.SetIsUpdaterOptional(UpdaterId id, bool isOptional)
* UpdaterRegistry.RegisterUpdater(UpdaterId id, bool isOptional)
* UpdaterRegistry.RegisterUpdater(UpdaterId id, Document doc, bool isOptional)

New methods were added to access information about currently registered updaters:

* UpdaterRegistry.GetRegisteredUpdaterInfos(Document doc)
* UpdaterRegistry.GetRegisteredUpdaterInfos()

Revit now disallows any calls to UpdaterRegistry from within the Execute() method of an updater. That means any calls to RegistryUpdater(), AddTrigger(), etc. will now throw an exception. The only method of UpdaterRegistry allowed to be called during execution of an updater is UnregisterUpdater(,) but the updater to be unregistered must be not the one currently being executed.

**Enclosure class renamed**

The Enclosure class, introduced in Revit 2011 as the parent class for Rooms, Spaces and Areas, was renamed to SpatialElement.

**Room, Area and Space boundary segments**

The individual BoundarySegment properties for the Room, Space and Area classes have been marked obsolete.

The SpatialElement class contains a new method:

* SpatialElement.GetBoundarySegments()

which works for all subclass types.

**Line origin and direction**

Two new properties added to Line class to get the origin and direction of a line:

* Line.Origin
* Line.Direction

**TextElement properties**

Five new read-only properties have been added to the TextElement class:

* TextElement.UpDirection - The direction towards the top of the text.
* TextElement.BaseDirection - The base direction for the text.
* TextElement.LineWidth - The TextElement line width.
* TextElement.Height - The TextElement height.
* TextElement.Align - The TextAlignFlags of the TextElement.

**FaceSplitter class**

The FaceSplitter class, representing the element produced by a Split Face operation, has been exposed to the API.

Use this class to identify the element whose face was split by the element (SplitElementId property).

Autodesk.Revit.DB.Options opt = app.Create.NewGeometryOptions();

opt.ComputeReferences = true;

opt.IncludeNonVisibleObjects = true;

FilteredElementCollector collector = new FilteredElementCollector(doc);

ICollection<FaceSplitter> splitElements = collector.OfClass(typeof(FaceSplitter)).Cast<FaceSplitter>().ToList();

foreach (FaceSplitter faceSplitter in splitElements)

{

Element splitElement = doc.get\_Element(faceSplitter.SplitElementId);

Autodesk.Revit.DB.GeometryElement geomElem = faceSplitter.get\_Geometry(opt);

foreach (GeometryObject geomObj in geomElem.Objects)

{

Line line = geomObj as Line;

if (line != null)

{

XYZ end1 = line.get\_EndPoint(0);

XYZ end2 = line.get\_EndPoint(1);

double length = line.ApproximateLength;

}

}

}

To find the faces created by the split, use the new Face.GetFaceRegions() method on the face of the host element for the split.

**ColumnAttachment**

The newly exposed ColumnAttachment class represents an attachment of the top or bottom of a column to a roof, floor, ceiling, or beam. Static methods:

* ColumnAttachment.GetColumnAttachment()
* ColumnAttachment.AddColumnAttachment()
* ColumnAttachment.RemoveColumnAttachment()

provide access to the settings in this class for a given element.

**Color class**

The API color class may represent an invalid or uninitialized color when an instance of it is obtained from Revit. Invalid colors cannot be read; a new exception will throw from the Red, Green, and Blue properties if you attempt to read them. Setting the color properties is permitted and makes the color no longer invalid.

The property

* Color.IsValid

identifies if the color is valid.

**FamilyInstance: flip work plane**

The new property

* FamilyInstance.IsWorkPlaneFlipped

is a settable property capable of changing if the work plane for a particular family instance is flipped.

The new property

* FamilyInstance.CanFlipWorkPlane

identifies if the family instance allows flipping of the work plane.

**New NewFamilyInstance() overloads**

Two new overloads are provided for creating family instances from References:

* Autodesk.Revit.Creation.ItemFactoryBase.NewFamilyInstance(Reference, Line, FamilySymbol)
* Autodesk.Revit.Creation.ItemFactoryBase.NewFamilyInstance(Reference, XYZ, XYZ, FamilySymbol)

These are identical to their counterparts that accept Faces as input. Because the Reference member will not always be populated in all Face handles, these overloads allow an alternate means of creating the target family instance attached to the right face.

The new overload

* Autodesk.Revit.Creation.ItemFactoryBase.NewFamilyInstance(Line, FamilySymbol, View)

provides the ability to create a line-based detail component.

**Allow and disallow wall end joins**

The new methods

* WallUtils.DisallowWallJoinAtEnd()
* WallUtils.AllowWallJoinAtEnd()
* WallUtils.IsWallJoinAllowedAtEnd()

provide access to the setting for whether or not joining is allowed at a particular end of the wall. If joins exist at the end of a wall and joins are disallowed, the walls will become disjoint. If joins are disallowed for the end of the wall, and then the setting is toggled to allow the joins, the wall will automatically join to its neighbors if there are any.

**Element.Pinned property**

The Element.Pinned property is no longer read-only. It now allows you to set an element to be pinned or unpinned.

**ElementFilter.PassesElement**

Two new overloads allow you to evaluate the result of an ElementFilter:

* bool ElementFilter.PassesElement(Element)
* bool ElementFilter.PassesElement(Document, ElementId)

**ElementMulticategoryFilter**

This new ElementFilter subtype allows you to easily find elements whose category matches any of a given set of categories.

**ElementMulticlassFilter**

This new ElementFilter subtype allows you to easily find elements whose class type matches any of a given set of classes.

**ElementPhaseStatus**

The new method

* Element.GetPhaseStatus()

returns the status of a given element in the input phase. Options include none (elements unaffected by phasing), new, demolished, past, future, existing and temporary.

The new element filter allows you to filter for elements which have a status matching one of the set of input statuses.

**Temporary view mode information**

The new method:

* View.IsInTemporaryViewMode()

identifies if a particular temporary view mode is active for a view.

The new method:

* View.IsElementVisibleInTemporaryViewMode()

identifies if an element should be visible in the indicated view mode. This applies only to the TemporaryHideIsolate and AnalyticalModel view modes.

**Generalized Array, Set, Map classes removed**

The Revit API classes in Autodesk.Revit.Collections:

* Array
* Set
* Map

have been removed. More flexible alternatives exist in the .NET framework System.Collections and System.Collections.Generic namespaces.

**Replacement for ExternalCommandData.Data**

The property

* ExternalCommandData.Data

has been replaced by

* ExternalCommandData.JournalData

The data type is now an IDictionary<String, String>.

The previous Data property has been marked Obsolete.

**Replacement for Application.LibraryPaths**

The property

* Application.LibraryPaths

has been replaced by two methods

* Application.GetLibraryPaths()
* Application.SetLibraryPaths()

The data type is now an IDictionary<String, String>.

The previous LibraryPaths property has been marked Obsolete.

**LinkElementId property changes**

The members of the LinkElementId class have changed in order to clarify what element is represented by an instance of this object. The new properties are:

* LinkElementId.LinkInstanceId - The id of the link, or invalidElementId if no link.
* LinkElementId.LinkedElementId - The id of the element in the link, or invalidElementId if no link.
* LinkElementId.HostElementId - The id of the element in the host, or invalidElementId if there is a link.

**Application properties**

The new properties:

* Application.Username
* Application.CentralServerName
* Application.LocalServerName

provide read access to information in the current Revit session.

**RevitAddInUtility.dll**

This DLL is now compiled as a compatible binary capable of execution on 32-bit or 64-bit systems.

**VSTA changes**

**VSTA enabled for multiple Revit sessions**

Macros can now be used in multiple Revit sessions launched from a single installation. Revit will no longer warn you about VSTA being disabled when the second and subsequent sessions are launched.

There are a few restrictions regarding what you can do from the second and subsequent sessions:

1. Application level macros in the second session may be run, but not edited, until the first session is closed.
2. Document level macros in the second session can be run and edited freely. However, edits to macro projects from a document opened in both sessions can't be saved until the document is closed by the first session.

**Structure API**

**Track Changes UI**

The track changes UI previously offered by Revit Structure has been removed. In previous releases, this offered the ability to highlight elements changed by API commands, and a limited ability to revert some or all of the changes.

Unfortunately, the functionality was limited in certain ways:

* It only watched and highlighted some types of changes that the API could make.
* It expected that only one transaction would be committed during a given API command.
* It did not respect changes made by applications during updaters and events.
* Some application developers did not want the changes they made to be undone partially.

If you wish to offer some ability for your users to see what changes were made by a given transaction, you can implement something similar or more sophisticated using the DocumentChanged event. End users are best served using the Undo mechanism to revert all changes made during a given transaction, if that is their desire. The Undo mechanism is consistent and compatible with multiple API transactions, events and updaters as well as individual commands.

**LineLoad.UniformLoad**

This property has been corrected to be Boolean instead of integer.

**NewBeamSystem() changes**

The family of NewBeamSystem() methods has changed. Previously curves could be input without a sketch plane or level as input, and the orientation of the active view would determine the orientation of the resultant beam system. Now, there are overloads accepting either the sketch plane or the level and those inputs are required.

The overloads of NewBeamSystem() now check that the input profile curves lie properly in the sketch plane or level. They also check to ensure that the profile forms a closed loop.

**NewTruss() change**

The SketchPlane input for this method is now required. Previously null could be input, and the orientation of the active view would determine the orientation of the resultant truss.

**Rebar changes**

**RebarShape creation and modification**

**New for alpha 3:** RebarShape elements are no longer modifiable (except for Allowed Bar Types). Instead of changing a RebarShape, create a new one based on the old one. This change results in a simplified API that works in the same way as the UI.

Also, the syntax for RebarShape creation has been changed. In 2011, the steps were:

1. Create a RebarShape inside the Document.
2. Create a RebarShapeDefinition inside the RebarShape.
3. Add data to the RebarShapeDefinition.
4. Commit the RebarShape.

In 2012, the steps are:

1. Create a RebarShapeDefinition. (It *refers* to the Document, but is not *inside* the document.)
2. Add data to the RebarShapeDefinition.
3. Create a RebarShape inside the Document based on the RebarShapeDefinition. (Only now is the document modified.)

Specifically, the following methods are removed:

* Autodesk.Revit.Creation.Document.NewRebarShape()
* RebarShape.NewDefinitionBySegments()
* RebarShape.NewDefinitionByArc()

and replaced by:

* RebarShape.Create(Document, RebarShapeDefinition, ... )
* The RebarShapeDefinitionBySegments constructor.
* The RebarShapeDefinitionByArc constructors.

The ability to modify hook angle, hook orientation, and style has been removed. These can only be set at creation.

* There is no SetHookAngle(int) method.
* There is no SetHookOrientation(int) method.
* The SetRebarStyle(RebarStyle) method has been removed.

**Rebar Shape Parameters**

**New for Alpha 3:** The interface for Rebar Shape Parameters has changed. Throughout the Rebar API, all methods now use ElementId to stand for a shared parameter, instead of ExternalDefinition. Methods are provided in the new RebarShapeParameters class to convert between ElementId and ExternalDefinition:

* RebarShapeParameters.GetAllRebarShapeParameters(Document), which returns an array of ElementIds.
* RebarShapeParameters.GetElementIdForExternalDefinition(Document, ExternalDefinition), which retrieves or adds the parameter to the document.
* RebarShapeParameters.GetExternalDefinitionForElementId(Document, ElementId, DefinitionFile), which retrieves a shared parameter corresponding to an id.
* The class RebarShapeMultiplanarDefinition and the RebarShape method getMultiplanarDefinition() have been added to support 3D rebar shape definitions.
* Families of category "Structural Connection" support the Structural Material Type parameter. Structural Connection families of type Concrete or Precast Concrete are considered corbels. Corbels support the following features:
  + Hosting rebar.
  + Autojoining to columns and walls.
  + Manual joining to other concrete elements.
* Families of category "Generic Model" support the "Can Host Rebar" parameter. Turning on this yes/no parameter allows instances to host rebar. Instances also now support the "Volume" parameter. (Join behavior does not change.)

The following RebarShape properties were replaced with get methods:

* property int HookAngle[int] -> GetHookAngle(int)
* property RebarHookOrientation HookOrientation[int] -> GetHookOrientation(int)
* property bool SameShapeIgnoringHooks[RebarShape] -> IsSameShapeIgnoringHooks(RebarShape)

The ability to modify hook angle, hook orientation, and style has been removed.

* There is no SetHookAngle(int) method.
* There is no SetHookOrientation(int) method.
* The SetRebarStyle(RebarStyle) method has been removed.

The return type of RebarShape.GetCurvesForBrowser() was changed from CurveArray to IList<Curve>.

**RebarHookType**

The method

* Autodesk.Revit.Creation.Document.NewRebarHookType(double angle, double multiplier)

was replaced by

* RebarHookType.Create(int angleDegrees, double multiplier)

New members were added to the RebarHookType class are new (its properties were only available through the Parameters interface before).

**RebarHostData class**

The RebarHostData class has the following changes:

* Method getRebarHostData(Element) renamed to GetRebarHostData(Element).
* Property CoverType[Reference] replaced with GetCoverType(Reference) and SetCoverType(Reference, CoverType).
* Property Valid deprecated; use IsValidHost() instead.
* Method HasCoverTypeForReference(Reference) deprecated; use IsFaceExposed(Reference) instead.
* New methods pertaining to cover: GetExposedFaces(), GetCommonCoverType(), SetCommonCoverType(CoverType).
* Other new methods: GetRebarsInHost(), GetAreaReinforcementsInHost(), GetPathReinforcementsInHost().

**Rebar class**

The two methods Creation.Document.NewRebar() were replaced by:

* Rebar.CreateFromCurves()
* Rebar.CreateFromRebarShape()

The new methods are similar to the old ones, except that they no longer return null for invalid arguments, and they no longer regenerate the document.

The new class RebarBendData is to support functionality that is not part of Alpha 3.

**MEP API small enhancements and changes**

**Spare and space circuits**

The new property

* ElectricalSystem.CircuitType

identifies the type of an electrical circuit (circuit, spare or space).

The method ElectricalSystem.AddToCircuit now throws an exception when the system is a spare or space circuit.

**Cable tray and conduit domain**

The enum class

* Autodesk.Revit.DB.Domain

adds a new enum value to represent cable tray & conduit.

**Connector**

New read-only properties have been added:

* Connector.JointType
* Connector.GenderType
* Connector.EngagementLength

**MEPSystem**

The new property

* MEPSystem.IsEmpty

identifies if the system currently contains no components.

**Graphical warnings for disconnects**

The new properties:

* Application.ShowGraphicalWarningCableTrayConduitDisconnects
* Application.ShowGraphicalWarningDuctDisconnects
* Application.ShowGraphicalWarningElectricalDisconnects
* Application.ShowGraphicalWarningPipeDisconnects

and the matching setters control whether Revit MEP will highlight disconnects in systems graphically.

**Space properties**

The new property

* Space.BaseHeatLoadOn

indicates if the value of Space.LatentHeatGainperPerson and Space.SensibleHeatGainperPerson properties is the default value or if it is user-defined.

To accomodate this property, the namespace of enum BaseLoadOn has changed from Autodesk.Revit.DB.Electrical to Autodesk.Revit.DB.

**Fitting methods**   
  
The following methods no longer remove unused or dangling curve connectors:

* Document.NewTeeFitting
* Document.NewCrossFitting
* Document.NewElbowFitting
* Document.NewUnionFitting
* Document.NewTransitionFitting
* Document.NewTakeoffFitting

Please use ConnectorManager.UnusedConnectors.Erase to remove unused connectors after creating a new fitting.