**Major changes and renovations to the Revit API**

**Document APIs**

**Document.Save()**

Several modifications have been made to the Document.Save() methods.

* The methods now return void instead of boolean.
* Failures are signaled with specified documented exceptions.
* The new property SaveOptions.Compact allows the caller to specify if the OS should eliminate all dead data from the file on disk.

**Document.SaveAs()**

Several modifications have been made to the Document.SaveAs() functions.

* The methods now return void instead of boolean.
* Failures are signaled with specified documented exceptions.
* The new property SaveAsOptions.MaximumBackups identifies the maximum number of backups to keep on disk.
* The new property SaveAsOptions.Compact allows the caller to specify if the OS should eliminate all dead data from the file on disk.
* The new property SaveAsOptions.WorksharingOptions offers options specific to workshared files:  
  + WorksharingSaveAsOptions.SaveAsCentral
  + WorksharingSaveAsOptions.OpenWorksetsDefault
  + WorksharingSaveAsOptions.ClearTransmitted
* The property SaveAsOptions.Rename has been obsoleted.  The property previously had no effect, use of SaveAs will always modify the document in memory.

In addition, a new version of SaveAs() taking a ModelPath and SaveAsOptions has been introduced.  This offers capabilities similar in behavior to SaveAs(string, SaveAsOptions).

**OpenOptions**

New options are available in this class:

* OpenOptions.Audit

This boolean option specifies whether to expand all elements, to check for corruption.  Defaults to false.

* OpenOptions.AllowOpeningLocalByWrongUser

This boolean option specifies whether a local file is allowed to be opened ReadOnly by a user other than its owner.

* OpenOptions.GetOpenWorksetsConfiguration()
* OpenOptions.SetOpenWorksetsConfiguration()

These methods access the WorksetConfiguration.   This class specifies which user-created worksets should be opened/closed when the document is opened.  Once an instance of this class is created, it can be further modified by calling available methods in any order.  It is a specification of a setting for model open; the methods of this class just adjust the specification, and do not themselves open or close worksets.

Only user-created worksets can be specified to be opened or closed.  All system worksets are automatically open.  An open workset allows its elements can be expanded and displayed. For a closed workset, Revit tries to not expand its elements, and to that end, does not display them. This is intended to help with performance by reducing Revit's memory footprint.

An application can determine how to populate this class by looking at workset information from a closed document.  This is done by calling

* WorksharingUtil.GetUserWorksetInfo()

This method returns a collection of WorksetPreview classes containing the workset id, unique id, owner, name and whether or not the workset is default.

**Application.OpenDocumentFile(ModelPath, OpenOptions)**

This method has been reimplemented.  While the signature has not changed, there are new options available to be set in OpenOptions.

Failures are now signaled with specific documented exceptions.

**Application.OpenDocumentFile(String)**

This method has been reimplemented.  While the signature has not changed, failures are now signaled with specific documented exceptions.

**UIApplication.OpenAndActivateDocument(ModelPath, OpenOptionsForUI)**

The signature and the implementation of this method has changed.

* The class OpenOptionsForUI has been removed.
* The new signature is (ModelPath, OpenOptions, Boolean detachAndPrompt).

If you wish to let the user answer interactively the prompt of whether to preserve or discard worksets when opening the documented detached from central, set the following values:

* OpenOptions.DetachFromCentralOption = DoNotDetach
* Boolean argment "detachAndPrompt" = true.

Failures are now signaled with specific documented exceptions.

**UIApplication.OpenAndActivateDocument(String)**

This method has been reimplemented.  Failures are now signaled with specific documented exceptions.

**FilteredElementCollector**

**Iteration and element deletion**

When an element is deleted active iterations over the document (via FilteredElementCollector) are now stopped. A new InvalidOperationException is thrown.  The iterator cannot proceed due to changes made to the Element table in Revit's database (typically, this can be the result of an Element deletion). This affects the use of FilteredElementIterator, FilteredElementIdIterator, and foreach loops over a FilteredElementCollector.

The exception can be triggered by direct calls to Document.Delete(), but also by other Revit APIs which change the document resulting in an element being deleted. In general it is best not to make any changes to the document while an iterator is running over the document.

The simplest workaround to fix existing code which encounters this error is to use one of:

* FilteredElementCollector.ToElements()
* FilteredElementCollector.ToElementIds()
* FilteredElementCollector.ToList<Type>() (LINQ method)

to get a standalone collection, then iterate that collection.  Even if elements are deleted from the document, the iteration of the already fetched collection can proceed.

**Geometry APIs**

**Curve creation**

New curve creation methods have been added as statics on the associated curve type:

* Curve.CreateTransformed()
* Line.CreateBound()
* Line.CreateUnbound()
* Arc.Create()
* Ellipse.Create()
* NurbSpline.Create()
* HermiteSpline.Create()

The older curve creation properties and methods are now obsolete:

* Curve.Transformed
* Line.Bound
* Line.Unbound
* Autodesk.Revit.Creation.Application.NewLine()
* Autodesk.Revit.Creation.Application.NewLineBound()
* Autodesk.Revit.Creation.Application.NewLineUnbound()
* Autodesk.Revit.Creation.Application.NewArc()
* Autodesk.Revit.Creation.Application.NewEllipse()
* Autodesk.Revit.Creation.Application.NewNurbSpline()
* Autodesk.Revit.Creation.Application.NewHermiteSpline()

Both the old and new curve creation routines are updated to consistently prevent creation of curves smaller than Revit's tolerance. This value is now exposed via:

* Application.ShortCurveTolerance

Some other adjustments have been made to the validation on specific curve creation routines, consult the documentation for details.

**Curve utilities**

The new methods:

* GetEndPoint()
* GetEndParameter()
* GetEndPointReference()

replace the indexed property utilities on Curve:

* EndPoint
* EndParameter
* EndPointReference

The setter for EndParameter is deprecated and code should be adjusted to call MakeBound(double, double) instead.

**Edge utilities**

The new methods:

* Edge.GetFace()
* Edge.GetEndPointReference()

replace:

* Edge.Face (property)
* Edge.EndPointReference (property)

**Transform initialization**

The new methods:

* Transform.CreateTranslation()
* Transform.CreateRotation()
* Transform.CreateRotationAtPoint()
* Transform.CreateReflection()

replace:

* Transform.Translation (property)
* Transform.Rotation (property)
* Transform.Reflection (property)

**SketchPlane creation**

The SketchPlane creation methods of Autodesk.Revit.Creation.ItemFactoryBase have been replaced.

* SketchPlane.Create(Document, Plane) replaces ItemFactoryBase.NewSketchPlane(Plane)
* SketchPlane.Create(Document, Reference) replaces ItemFactoryBase.NewSketchPlane(Reference)
* Either method may be appropriate to replace calls to ItemFactoryBase.NewSketchPlane(PlanarFace), depending on whether the goal is a sketch plane tied to a Revit geometric reference, or a sketch plane fixed in space to a given plane.

A new SketchPlane creation has been added:

* SketchPlane.Create(ElementId)

This creates a SketchPlane from a grid, reference plane, or level.

SketchPlane also now has two more methods to get related properties of the SketchPlane element.

* SketchPlane.GetPlane() Returns the corresponding Plane.
* SketchPlane.GetPlaneReference()  Returns a reference to this element as a plane.

**BeamSystem creation**

The BeamSystem creation methods of Autodesk.Revit.Creation.Document have been replaced:

* BeamSystem.Create(Document, IList<Curve>, SketchPlane, int) replaces Document.NewBeamSystem(CurveArray, SketchPlane)
* BeamSystem.Create(Document, IList<Curve>, SketchPlane, XYZ, bool) replaces Document.NewBeamSystem(CurveArray, SketchPlane, XYZ, bool)
* BeamSystem.Create(Document, IList<Curve>, Level, int, bool) replaces Document.NewBeamSystem(CurveArray, Level)
* BeamSystem.Create(Document, IList<Curve>, Level, XYZ, bool) replaces Document.NewBeamSystem(CurveArray, Level, XYZ, bool)

**Truss creation**

The Truss creation method of Autodesk.Revit.Creation.Document have been replaced.

* Truss.Create(Document, ElementId, ElementId, Curve) replaces Document.NewTruss(TrussType, SketchPlane, Curve)

**Family Symbol API**

Family symbols that are not used in the document are now inactive until they are used.  A symbol's geometry will be empty and should not be accessed until it is active.  To test if a symbol is active, use

* FamilySymbol.IsActive()

To activate an inactive family symbol, use

* FamilySymbol.Activate()

**Units API**

The API for Units in Revit has been expanded and changed. The methods

* Document.GetUnits()
* Document.SetUnits()

allow interaction with the units of a document.  The Units class provides access to data such as

* DecimalSymbol
* DigitGroupingAmount
* DigitGroupingSymbol
* FormatOptions

The FormatOptions class provides access to data including:

* Rounding
* Accuracy
* DisplayUnits
* SuppressLeadingZeros
* SuppressTrailingZeros
* SuppressSpaces

LabelUtils.GetLabelFor() has been enhanced so that it can now return the user-visible name for a UnitSymbolType.

**Unit Formatting and Parsing**

The methods:

* UnitFormatUtils.FormatValueToString()
* UnitFormatUtils.TryParse()

provide the ability to format a value into a string based on formatting options and to parse a formatted string (including units) into a value if possible.

**Unit Conversion**

The new class UnitUtils contains methods to convert between unit types:

* UnitUtils.Convert() - Converts a value from one unit type to another, such as square feet to square meters.
* UnitUtils.ConvertFromInternalUnits() - Converts a value from Revit's internal unit type.
* UnitUtils.ConvertToInternalUnits() - Converts a value to Revit's internal unit type.

**View API changes**

**Viewport.Create behavioral change**

The method

* Viewport.Create()

previously did not consistently align the center of the Viewport with the point supplied.  This has been changed, and now the center will be aligned with the input point.  This should allow easier alignment of multiple viewports on the same sheet.

**View.ViewName obsolete**

The property

* View.ViewName

is now obsolete.  View.Name can be used.

**View.SetVisibility()**

The name of this method has now been correctly capitalized.

**ViewSchedule changes**

ViewSchedule is now a child of TableView.   All previously existing ViewSchedule API is still valid, but TableView also brings in a set of APIs related to:

* Table sections (header and body)
* Table formatting
* The contents of individual table cells

There are methods on TableView (and its constituent TableData and TableSectionData class) that are useful for Electrical Panel Schedules and some varieties of specialized schedules but forbidden for use with standard Revit tabular schedules generated from Revit content (e.g. InsertRow(), RemoveRow()).   Use of these functions on standard Revit schedules will result in an exception.

Some new members were introduced on schedule related classes:

* ScheduleField.SetStyle()
* ScheduleField.GetStyle()
* ScheduleField.IsOverridden
* ScheduleField.ResetOverrides()
* ScheduleField.GetFormatOptions()
* ScheduleField.SetFormatOptions()

relate to the style and formatting applied to schedule columns, and:

* ScheduleField.IsCalculatedField
* ScheduleField.IsCombinedParameterField
* ScheduleField.HasSchedulableField

relate to information about the type of a field.

The new members:

* ViewSchedule.GetFontId()
* ViewSchedule.AddFontId()

provide access to fonts stored in the table and applied to cells.

The new members:

* ViewSchedule.GroupHeaders()
* ViewSchedule.UngroupHeaders()
* ViewSchedule.CanGroupHeaders()
* ViewSchedule.CanUngroupHeaders()

affect header grouping in the schedule.

The new method:

* ViewSchedule.GetTableData()

returns the object which provides access to the sections of the table.

**Materials API changes**

**Applying visual materials**

The method

* Material.SetRenderAppearance()

has been deprecated.  The Render appearance properties should be set via the related ApperanceAssetElement.

Use the new property:

* Material.AppearanceAssetId

to assign the element to the material.

AppearanceAssetElements can be found via element filtering - they expose the following members:

* AppearanceAssetElement.Create() - creates a new asset element for a given rendering Asset and name.
* AppearanceAssetElement.GetAppearanceAssetElementByName() - gets an asset element handle given the name.
* AppearanceAssetElement.SetRenderingAsset() - Sets the rendering Asset to the element

**AssetProperty changes**

AssetProperty no longer inherits from APIObject.

New subclasses of AssetProperty are exposed:

* AssetPropertyList - a property consisting of a list of AssetProperty members.
* AssetPropertyFloatArray - a property consisting of an array of float values.
* AssetPropertyUInt64 - a property of UInt64 value.
* AssetPropertyInt64 - a property of Int64 value.

Some of the return values of AssetProperty.GetTypeName() have been changed as shown in the following form:

|  |  |  |
| --- | --- | --- |
| **Input argument** | **old return** | **new return** |
| APT\_Double | "Double1" | "Double" |
| APT\_DoubleArray2d | "Double2" | "DoubleArray2d" |
| APT\_DoubleArray3d | "Double3" | "DoubleArray3d" |
| APT\_DoubleArray4d | "Double4" | "DoubleArray4d" |
| APT\_Asset | "RenderingAsset" | "Asset" |
| APT\_FloatArray | "Float3" | "FloatArray" |

**UI API changes**

**External commands now supported from Project Browser as active view**

API commands and macros are now enabled when the Revit active view is the Project Browser.

* If there are actively selected elements in the Project Browser, these will be returned from UIDocument.Selection
* However, add-ins cannot prompt for interactive selection when the Project Browser is the active view.
* The enumerated type ViewType now has separate entries for ProjectBrowser and SystemBrowser to allow applications to deal with situations where the Project Browser is active.  These view types used to be returned as ViewType.Internal, so code which keys of off ViewType.Internal may need to be updated to also deal with these new types.

Note that API commands are still disabled when the active view is the MEP system browser.

**Beam and Brace Parameters changes**

Revit 2014 includes several changes to control the position of structural framing members like beams and braces.  These changes do not affect the API members but can be accessed via parameters.

**Start/End Extension & Cutback**

There are new extension and cutback parameters assigned to Beam and Brace elements.

* BuiltInParameter.START\_EXTENSION
* BuiltInParameter.END\_EXTENSION
* BuiltInParameter.START\_JOIN\_CUTBACK
* BuiltInParameter.END\_JOIN\_CUTBACK

**Note**: In some families “Start Extension” and “End Extension” family parameters may also exist but it is recommended to use instead new the new built-in parameters.

These parameters work as follows:

* If the Beam or Brace element end doesn’t belong to a join Revit uses the new parameters “Start Extension" or "End Extension”.  Assigning positive values to these parameters lengthens the element).
* If the Beam or Brace element belongs to a join Revit uses the parameters “Start Join Cutback" or "End Join Cutback”.  Assigning positive values to these parameters shortens the element).

In certain cases it may be difficult to detect if an element node belongs to join or not. Therefore, it may be advisable to set both groups of parameters via the API. With both groups of parameters will be set for element, Revit automatically detects which should be applied to the structural element.

**Justifications**

There are new justification parameters assigned to Beam and Brace elements. The new set of parameters provides more options to manipulate the physical element in relation to its analytical curve.

|  |  |
| --- | --- |
| Parameter | Permitted values |
| BuiltInParameter.YZ\_JUSTIFICATION | YZJustificationOption.Uniform  YZJustificationOption.Independent |
| BuiltInParameter.Y\_JUSTIFICATION | YJustification.Left  YJustification.Center  YJustification.Origin  YJustification.Right |
| BuiltInParameter.Y\_OFFSET\_VALUE | double |
| BuiltInParameter.Z\_JUSTIFICATION | ZJustification.Top  ZJustification.Center  ZJustification.Origin  ZJustification.Bottom |
| BuiltInParameter.Z\_OFFSET\_VALUE | double |
| BuiltInParameter.START\_Y\_JUSTIFICATION | YJustification.Left  YJustification.Center  YJustification.Origin  YJustification.Right |
| BuiltInParameter.START\_Y\_OFFSET\_VALUE | double |
| BuiltInParameter.START\_Z\_JUSTIFICATION | ZJustification.Top  ZJustification.Center  ZJustification.Origin  ZJustification.Bottom |
| BuiltInParameter.START\_Z\_OFFSET\_VALUE | double |
| BuiltInParameter.END\_Y\_JUSTIFICATION | YJustification.Left  YJustification.Center  YJustification.Origin  YJustification.Right |
| BuiltInParameter.END\_Y\_OFFSET\_VALUE | double |
| BuiltInParameter.END\_Z\_JUSTIFICATION | ZJustification.Top  ZJustification.Center  ZJustification.Origin  ZJustification.Bottom |
| BuiltInParameter.END\_Z\_OFFSET\_VALUE | double |

Previously only Beam elements had any justification parameters.

In Revit 2014, Beams as well as Braces share this set of built-in justification parameters.

The following table shows a mapping from old justification parameters to new ones:

|  |  |
| --- | --- |
| Deprecated parameter | New parameter |
| BEAM\_H\_JUSTIFICATION | Y\_JUSTIFICATION |
| BEAM\_V\_JUSTIFICATION | Z\_JUSTIFICATION |
| BEAM\_V\_JUSTIFICATION\_OTHER\_VALUE ( valid only for Other value set to BEAM\_V\_JUSTIFICATION ) | Z\_OFFSET\_VALUE ( valid for all Z\_JUSTIFICATION values ) |

Old beam parameters can still be used.  When set, they will be used to recalculate the new parameters.  It is strongly recommended to use new justification parameters instead of old ones, because in some cases the old parameters may be not sufficient to correctly describe the element position (for example, when justification is set independently on either end of the member).

**Divided surface changes**

**Divided Surface API**

The class DividedSurfaceData and the associated method Element.GetDividedSurfaceData() have been replaced.  The new methods to access DividedSurfaces applied to elements are:

* DividedSurface.GetReferencesWithDividedSurface()
* DividedSurface.GetDividedSurfaceForReference()

The method:

* Autodesk.Revit.Creation.Document.NewDividedSurface()

has been replaced by:

* DividedSurface.Create()

**Point clouds**

Revit 2014 includes a new point cloud engine.  This new engine supports .rcp and .rcs file formats.  The introduction of this new engine has led to several changes in the client API around Point Cloud elements.

**PointCloudType.Create()**

The method

* PointCloudType.Create()

no longer supports input of .pcg files to directly create a new PointCloudType.  .pcg files can be indexed to create matching .rcs files, but this does not happen automatically when using this method.

This method does support creation of PointCloudTypes from .rcp or .rcs files, or from custom formats supplied by the Point Cloud Engine API.

**PointCloudInstance.GetPoints()**

The method

* PointCloudInstance.GetPoints(PointCloudFilter filter, int numPoints)

has been deprecated and replaced by

* PointCloudInstance.GetPoints(PointCloudFilter filter, double averageDistance, int numPoints)

The new averageDistance argument is the desired average distance between "adjacent" cloud points (in Revit units of length).  The smaller the averageDistance the larger number of points will be returned up to the numPoints limit.  Specifying this parameter makes the actual number of points returned for a given filter independent of the density of coverage produced by the scanner. This average distance parameter is only used when extracting points from .rcs or .rcp point clouds, and is not used when extracting from .pcg point clouds or custom point clouds from the Point Cloud Engine API.

**Point cloud overrides**

The classes

* PointCloudOverrides
* PointCloudOverrideSettings
* PointCloudColorSettings

allow read and write access to point cloud override settings assigned to a given view.  Overrides can be applied to an entire point cloud instance, or to specific scans within that instance. Options for the overrides include making the cloud non-visible, setting it to a fixed color, or to color gradients based on elevation, normals, or intensity.

The property

* PointCloudInstance.SupportsOverrides

identifies point clouds which support override settings (clouds which are based on .rcp or .rcs files).

**Point cloud scans**

The method

* PointCloudInstance.GetScans()

returns a list of scans contained within the .rcp point cloud.  Scan names can be used to set visibility and fixed color overrides independently for each scan in the PointCloudInstance.

**IFC API changes**

**IFC export now External Service**

The capability to override IFC export is now managed as an External Service.  As such, the explicit interfaces in ExporterIFCRegistry are no longer needed and have been marked obsolete.  It is now possible to register more than one external IFC export implementation in the same session, and manage the active implementation using the methods of the SingleServerService wrapper to IFC export.

**IFC APIs moved to new assembly**

Most IFC API classes have been moved from RevitAPI.dll to a new assembly: RevitAPIIFC.dll. The classes moved include all members of the Autodesk.Revit.DB.IFC namespace which enable development of a custom IFC exporter.

Any Add-In using any of the migrated APIs needs to reference the new DLL and rebuild to work in Revit 2014.

Note that the APIs to invoke an IFC export and import:

* Document.Export(String, String, IFCExportOptions)
* Application.OpenIFCDocument(String)

and the related options class have not moved. Applications which export or import IFC files but do not provide custom implementations do not need to make changes.

**PrintParameters**

The property HideUnreferencedViewTages was renamed to HideUnreferencedViewTags.

**Obsolete API Removal**

API classes and methods previously marked Obsolete in Revit 2013 or earlier have been removed from the API.

**Classes removed**

* CurtainCellSetIterator and CurtainCellSet - Use generic .NET collection classes instead.
* PipeMaterialType - replaced by Material
* Batch creation argument class types - batch creation is not needed with changes to Revit API regeneration
  + ProfiledWallCreationData
  + RectangularWallCreationData
  + RoomCreationData
  + TextNoteCreationData
* Subclasses of ConnectorElement have been removed, access these elements through the parent class
  + ElectricalConnector
    - ElectricalConnector.SystemType - query the parameter RBS\_ELEC\_CIRCUIT\_TYPE on ConnectorElement and cast to ElectricalSystemType
  + DuctConnector
    - DuctConnector.LinkedConnector - replaced by ConnectorElement.GetLinkedConnectorElement() and ConnectorElement.SetLinkedConnectorElement()
    - DuctConnector.SystemType - query the parameter RBS\_DUCT\_CONNECTOR\_SYSTEM\_CLASSIFICATION\_PARAM on ConnectorElement and cast to DuctSystemType
  + PipeConnector
    - PipeConnector.LinkedConnector replaced by ConnectorElement.GetLinkedConnectorElement() and ConnectorElement.SetLinkedConnectorElement()
    - PipeConnector.SystemType - query the parameter RBS\_PIPE\_CONNECTOR\_SYSTEM\_CLASSIFICATION\_PARAM on ConnectorElement and cast to PipeSystemType

**Methods and Properties removed**

**Autodesk.Revit.Creation namespace**

**Application**

* + - NewMaterialSet() - replaced by .net Generic collection classes
    - NewElementArray() - replaced by .net Generic collection classes

**Document**

* + - NewAnnotationSymbol(XYZ ,AnnotationSymbolType ,View) - replaced by NewFamilyInstance(XYZ, FamilySymbol, View)
    - NewAreaViewPlan(String,Level,AreaElemType) - replaced by ViewPlan.CreateAreaPlan()
    - NewCurtainSystem(ReferenceArray,CurtainSystemType) - replaced by NewCurtainSystem2(ReferenceArray, CurtainSystemType)
    - NewElectricalSystem(ElementSet,ElectricalSystemType) - replaced by NewElectricalSystem(ICollection<ElementId>, ElectricalSystemType)
    - NewFamilyInstances(List<FamilyInstanceCreationData>) - replaced by NewFamilyInstances2(List<FamilyInstanceCreationData>)
    - NewGroup(ElementSet) - replaced by NewGroup(System.Collections.Generic.ICollection<Autodesk.Revit.DB.ElementId>)
    - NewRooms(Phase, Int32) - replaced by NewRooms2(Phase, Int32)
    - NewRooms(Level,Phase) - replaced by NewRooms2(Level, Phase)
    - NewRooms(Level) - replaced by NewRooms2(Level)
    - NewRooms(List<RoomCreationData>) - there is no single equivalent that creates multiple rooms, this is not needed with Revit API control over regeneration
    - NewSpaces(Phase,Int32) - replaced by NewSpaces2(Phase, Int32)
    - NewSpaces(Level,Phase,View) - replaced by NewSpaces2(Level, Phase, View)
    - NewTextNotes(List<TextNoteCreationData>) - replaced by NewTextNote()
    - NewViewPlan(String,Level,ViewPlanType) - replaced by ViewPlan.Create(Document, ElementId, ElementId)
    - NewView3D(XYZ) - replaced by View3D.CreateIsometric(Document, ElementId)
    - NewViewSection(BoundingBoxXYZ) - replaced by ViewSection.CreateDetail()
    - All Wall creation methods replaced by equivalent Wall.Create() methods:
      * NewWall(CurveArray,WallType,Level,Boolean,XYZ)
      * NewWall(CurveArray,WallType,Level,Boolean)
      * NewWall(CurveArray,Boolean)
      * NewWall(Curve,WallType,Level,Double,Double,Boolean,Boolean)
      * NewWall(Curve,Level,Boolean)
      * NewWalls(List<ProfiledWallCreationData> dataList) - there is no single equivalent that creates multiple walls, this is not needed with Revit API control over regeneration
      * NewWalls(List<RectangularWallCreationData> dataList) - there is no single equivalent that creates multiple walls, this is not needed with Revit API control over regeneration

**FamilyItemFactory**

* + - NewDuctConnector(Reference,DuctSystemType) - replaced by ConnectorElement.CreateDuctConnector()
    - NewPipeConnector(Reference,PipeSystemType) - replaced by ConnectorElement.CreatePipeConnector()
    - NewElectricalConnector(Reference,ElectricalSystemType) - replaced by ConnectorElement.CreateElectricalConnector()

**Autodesk.Revit.DB namespace**

**BaseArray**

* + - CopyMembers - replaced by GetCopiedMemberIds()
    - OrigMembers - replaced by GetOriginalMemberIds()

**CurtainGrid**

* + - UnlockedMullions - replaced by GetUnlockedMullionIds()
    - Mullions - replaced by GetMullionIds()
    - Cells - replaced by GetCurtainCells()
    - UnlockedPanels - replaced by GetUnlockedPanelIds()
    - Panels - replaced by GetPanelIds()
    - VGridLines - replaced by GetVGridLineIds()
    - UGridLines - replaced by GetUGridLineIds()

**CurveElement**

* + - LineStyles - replaced by GetLineStyleIds()

**Document**

* + - Delete(ElementSet elements) - replaced by Delete(System.Collections.Generic.ICollection<Autodesk.Revit.DB.ElementId> elementIds)
    - WorksharingCentralFilename - replaced by ModelPathUtils.ConvertModelPathToUserVisiblePath(Document.GetWorksharingCentralModelPath())
    - PrintSettings - replaced by GetPrintSettingIds()
    - Element/get\_Element - replaced by GetElement method

**Element**

* + - PhaseDemolished - replaced by DemolishedPhaseId
    - PhaseCreated - replaced by CreatedPhaseId

**FamilyInstance**

* + - GetCopings() - replaced by GetCopingIds()
    - SetCopings(ElementSet) - replaced by SetCopingIds(ICollection<ElementId> cutters)
    - SubComponents - replaced by GetSubComponentIds()

**Floor**

* + - SpanDirectionSymbols - replaced by GetSpanDirectionSymbolIds()

**GeometryElement**

* + - Objects - replaced by GetEnumerator()

**Group**

* + - Ungroup() - replaced by UngroupMembers()
    - Members - replaced by GetMemberIds()

**LinearArray**

* + - CopyMembers - replaced by GetCopiedMemberIds()
    - OrigMembers - replaced by GetOriginalMemberIds()

**Material**

* + - GetMaterialAspectPropertySet(MaterialAspect) - replaced by GetStructuralAssetId() and GetThermalAssetId()
    - SetMaterialAspect(MaterialAspect,ElementId,Boolean) - replaced by SetStructuralAssetId() and SetThermalAssetId()
    - SetMaterialAspectToIndependent(MaterialAspect) - replaced by SetStructuralAssetId() and SetThermalAssetId()

**MEPSystem**

* + - Remove(ElementSet) - replaced by Remove(ICollection<ElementId>)

**Part**

* + - ParentDividedElementId - no replacement, concept is removed from Revit
    - OriginalDividedElementId - no replacement, concept is removed from Revit
    - GetDividedParents() - no replacement, concept is removed from Revit

**PartMaker**

* + - IsElementDivided(ElementId elemId) - replaced by IsSourceElement(ElementId)
    - GetDividedElementIds() - replaced by GetSourceElementIds()
    - SetDividedElementIds(ICollection<ElementId>) - replaced by SetSourceElementIds(ICollection<ElementId>)

**PartUtils**

* + - AreElementsValidForDivide(Document, ICollection<ElementId>) - replaced by ArePartsValidForDivide(Document, ICollection<ElementId>)
    - AreElementsValidIntersectingReferences(Document, ICollection<ElementId>) - replaced by PartMakerMethodToDivideVolumes.AreElementsValidIntersectingReferences(Document, ICollection<ElementId>)
    - IsValidSketchPlane(Document, ElementId) - replaced by PartMakerMethodToDivideVolumes.IsValidSketchPlane(Document, ElementId)
    - SetOffsetForIntersectingReference() - replaced by PartMakerMethodToDivideVolumes.SetOffsetForIntersectingReference()
    - GetOffsetForIntersectingReference() - replaced by PartMakerMethodToDivideVolumes.GetOffsetForIntersectingReference()
    - PartMakerUsesReference() - replaced by PartMakerMethodToDivideVolumes.PartMakerUsesReference()
    - IsMaxDivisionDepthReached(Document, ElementId) - no replacement, concept is removed from Revit
    - GetDividedParents(Part) - no replacement, concept is removed from Revit
  + PlanTopology
    - Rooms - replaced by GetRoomIds()

**PropertySetElement**

* + - Create(Document, MaterialAspect) - replaced by Create(Document, StructuralAsset)

**RadialArray**

* + - CopyMembers - replaced by GetCopiedMemberIds()
    - OrigMembers - replaced by GetOriginalMemberIds()

**SpatialFieldManager**

* + - UpdateSpatialFieldPrimitive(Int32,FieldDomainPoints,FieldValues) - replaced by UpdateSpatialFieldPrimitive(Int32, FieldDomainPoints, FieldValues, Int32)
    - SetUnits(IList<string>, IList<double>) - replaced by AnalysisResultSchema.SetUnits() and SetResultSchema()

**View**

* + - ApplyTemplate(View viewTemplate) - replaced by ViewTemplateId/ApplyViewTemplateParameters(View viewTemplate)
    - Hide(ElementSet elemSet) - replaced by HideElements(System::Collections::Generic::ICollection<Autodesk::Revit::DB::ElementId^>^ elementIdSet)
    - Unhide(ElementSet elemSet) - replaced by UnhideElements(System::Collections::Generic::ICollection<Autodesk::Revit::DB::ElementId^>^ elementIdSet)

**View3D**

* + - EyePosition - replaced by ViewOrientation3D.EyePosition/View.Origin
  + ViewSheet
    - AddView(View,UV) - replaced by Viewport.Create(Document, ElementId, ElementId, XYZ)

**Autodesk.Revit.DB.Plumbing namespace**

**PipeType**

* + - Roughness - replaced by Segment.Roughness
    - Material - replaced by PipeSegment.MaterialId

**Autodesk.Revit.DB.Structure namespace**

**AnalyticalMode**l

* + - CanDisable() - no replacement, concept removed from Revit
    - IsValidAnalyticalProjectionType(AnalyticalDirection,AnalyticalProjectionType) - replaced by IsValidProjectionType(AnalyticalElementSelector, AnalyticalDirection, AnalyticalProjectionType)

**AreaReinforcement**

* + - NumBarDescriptions - replaced by GetRebarInSystemIds()
    - BarDescription - replaced GetRebarInSystemIds()
    - Curves - replaced by GetCurveElementIds()

**BeamSystem**

* + - GetAllBeams() - replaced by GetBeamIds()

**PathReinforcement**

* + - BarDescription - replaced by GetRebarInSystemIds()
    - Curves - replaced by GetCurveElementIds()

**Rebar**

* + - GetCenterlineCurves(Boolean) - replaced by GetCenterlineCurves(Boolean,Boolean,Boolean)
    - DistributionPath - replaced by GetDistributionPath()
    - RebarShape - replaced by RebarShapeId
    - Host - replaced by Rebar.GetHostId() and SetHostId(Document, ElementId)
    - BarType - replaced by Element.GetTypeId() and Element.ChangeTypeId(ElementId)

**Major enhancements to the Revit API**

**Worksharing API enhancements**

**Reload Latest**

The method:

* Document.ReloadLatest()

Fetches changes from central (due to one or more synchronizations with central and merges them into the current session. After this call finishes, use

* Document.HasAllChangesFromCentral()

to confirm that there were no Synchronizations with Central performed during execution of ReloadLatest.

**Synchronize with Central**

The method:

* Document.SynchronizeWithCentral()

 performs a reload latest until the model in the current session is up to date and then saves changes back to central.  A save to central is performed even if no changes were made.

**Element ownership**

The following methods affect element and workset ownership:

* WorksharingUtils.CheckoutElements - Obtains ownership for the current user of as many specified elements as possible.
* WorksharingUtils.CheckoutWorksets - Obtains ownership for the current user of as many specified worksets as possible.
* WorksharingUtils.RelinquishOwnership - Relinquishes ownership by the current user of as many specified elements and worksets as possible, and grants element ownership requested by other users on a first come, first served basis.

**Create new local**

The new method:

* WorksharingUtil.CreateNewLocal()

takes a path to a central model and copies the model into a new local file for the current user.

**Enable Worksharing**

The new method Document.EnableWorksharing will enable worksharing in a project.

**Enhancements to interactions with links**

Several improvements have been made to increase API functionality when working with RVT links.

**Identifying links**

The property:

* Document.IsLinked

identifies if a document is in session because it is a linked Revit file.

**Obtaining linked documents**

The method:

* RevitLinkInstance.GetLinkedDocument()

gets the document that corresponds to an Revit Link instance.

**Link creation**

Two functions have been added to RevitLinkOptions, used to specify options for RevitLinkType.Create.

* RevitLinkOptions.GetWorksetConfiguration()
* RevitLinkOptions.SetWorksetConfiguration()

This allows the user to specify which worksets should be opened when creating and loading a new Revit link.

**Link load and unload**

The methods

* RevitLinkType.Load()
* RevitLinkType.LoadFrom()
* RevitLinkType.Unload()

allow a link to be loaded or unloaded, or to be loaded from a new location.   These methods regenerate the document.  The document's Undo history will be cleared by these methods.  As a result, this function and others executed before it cannot be undone.  All transaction phases (e.g. transactions, transaction groups and sub-transactions) that were explicitly started must be finished prior to calling this method.

Link shared coordinates

The methods:

* RevitLinkType.SavePositions()
* RevitLinkType.HasSharedCoordinatesChanges()

support save of shared coordinates changes back to the linked document. While this operation does not clear the document's undo history, you will not be able to undo this specific action, since it saves the link's shared coordinates changes to disk.

**Link path type**

The property:

* RevitLinkType.PathType

allows read and modification of the path type associated with a link.

**Conversion of geometric references**

The new APIs:

* Reference.LinkedElementId
* Reference.CreateLinkReference(RevitLinkInstance)
* Reference.CreateReferenceInLink()

allow conversion between Reference objects which reference only the contents of the link and Reference objects which reference the host. This allows an application, for example, to look at the geometry in the link, find the needed face, and convert the reference to that face into a reference in the host suitable for use to place a face-based instance. Also, they would allow you to obtain a reference in the host (e.g. from a dimension or family) and convert it to a reference in the link, suitable for use in Element.GetGeometryObjectFromReference().

**Room tag creation from linked rooms**

The new method:

* NewRoomTag(LinkElementId roomId, UV point, ElementId viewId)

can create room tags in Revit Links.

**Picking in links**

PickObject() and PickObjects() now allow selection of elements in RVT Links.

**Views & Display**

**Graphic Display options**

These new members expose read and write of graphic display options:

* View.GetBackground()
* View.SetBackground()
* View.ShadowIntensity
* View.SunlightIntensity
* View.SurfaceTransparency
* View.ShowEdges
* View.ShowSilhouettes
* View.SilhouetteLineStyleId

**Category classes override**

Display of category classes may be overridden by the user.  The new properties:

* View.AreModelCategoriesHidden
* View.AreAnnotationCategoriesHidden
* View.AreAnalyticalModelCategoriesHidden
* View.AreImportCategoriesHidden

allow an application to determine if a class of categories has been completely hidden.  Note that Category.Visible[View] and View.GetVisibility(Category) does not look to the category classes when it returns the individual visibility status.

**Category override**

Display of categories can be overridden. This can be done with the new class OverrideGraphicSettings and the new View methods:

* SetCategoryOverrides
* GetCategoryOverrides
* IsOverrideValidForCategory
* IsCategoryOverridable

**Element Override**

Display of elements can be overridden with the new methods:

* View.GetElementOverrides
* View.SetElementOverrides.

**View Filters**

A new set of methods on the View class allow getting, setting, adding, and removing filters. Filters can be created using the ParameterFilterElement class and its Create method which existed in previous versions of the Revit API.

**Non-rectangular crop region**

Views can now be assigned a crop region which is non-rectangular.  The new View members:

* View.GetCropRegionShapeManager()
* View.GetCropRegionShapeManagerForReferenceCallout()
* View.GetReferenceCallouts()

provide access to a ViewCropRegionShapeManager for the view or for a displayed reference callout.

This class allows access and modification of the crop region shape:

* ViewCropRegionShapeManager.GetCropRegionShape()
* ViewCropRegionShapeManager.SetCropRegionShape()
* ViewCropRegionShapeManager.IsCropRegionShapeValid()
* ViewCropRegionShapeManager.RemoveCropRegionShape()
* ViewCropRegionShapeManager.ShapeSet
* ViewCropRegionShapeManager.Valid

The properties:

* View.CropBoxActive
* View.CropBoxVisible

also apply to non-rectangular crop regions which may be assigned to a given view.

**Viewport**

The new property

* Viewport.Rotation

controls the rotation of the viewport on the sheet.

The new method

* Viewport.MoveCenterTo()

moves the viewport so that the center of the box outline (excluding the viewport label) is at a given point.

The new method

* Viewport.GetBoxCenter()

returns the center of the outline of the viewport on the sheet, excluding the viewport label.

**Exploded Views**

The new DisplacementElement class can be used to cause elements to appear displaced from their actual location to create views such as this one where the roof has been moved in the positive Z direction. The DisplacementPath class creates an annotation that depicts the movement of the element from its actual location to its displaced location.

**Revisions on sheets**

The new methods:

* ViewSheet.GetAllProjectRevisionIds()
* ViewSheet.GetAdditionalProjectRevisionIds()
* ViewSheet.SetAdditionalProjectRevisionIds()

provide access to the ids of project revision elements associated to a particular sheet.  GetAllProjectRevisionIds() returns project revisions ordered from lowest to highest by project revision sequence number.  The results include revisions associated to a revision cloud visible on the sheet and revisions that have been additionally included using the Revisions On Sheet parameter.   GetAdditionalProjectRevisionIds() and SetAdditionalProjectRevisionIds() access specifically the additional revisions added using the Revisions On Sheet parameter.

Note that there is no special class for project revision elements, but that they can be accessed as Elements by filtering on category BuiltInCategory.OST\_Revisions.

**User interface customization**

**UIView**

**Zoom operations**

The new methods:

* UIView.ZoomToFit()
* UIView.ZoomSheetSize()
* UIView.Zoom(double zoomFactor)

provide shortcuts to quickly adjust the zoom of the graphical view window.

**Close view**

The new method:

* UIView.Close()

closes a visible view window. Note that the last open view for a given document cannot be closed by this method.

**PreviewControl**

The new property:

* PreviewControl.UIView

returns a UIView handle to the preview view. This allows an application to manipulate the zoom and pan settings of the preview view.

The new property:

* PreviewControl.ScrollbarVisibility

accesses the visibility setting for the preview view scrollbars.

**Command API**

**Command event**

The event

* AddInCommand.BeforeExecuted

occurs before the command associated with an AddInCommandBinding executes.  This event is read-only, an application can react to this event but cannot make changes to documents, or affect the invocation of the command in any way.

**Command posting**

The method

* UIApplication.PostCommand()

posts a command to Revit.  Revit will invoke it when control returns from the current API context.   Only certain commands can be posted using this method:

1. Commands listed in the Autodesk.Revit.UI.PostableCommand enumerated type
2. External commands created by any add-in

This restriction prevents posting of contextual commands (because Revit's command framework cannot directly access the accessibility of some contextual commands).

Note that only one command may be posted to Revit at a given time.  If a second command is posted from any API application, the method throws an InvalidOperationException.

The command must be accessible to be executed.  This is determined only at the point where Revit returns from the API context, and therefore a failure to execute the command because the command is not currently accessible will not be reported directly back to the application that posted the command.

To use this API, the application must obtain a RevitCommandId handle for the command.  This can be done using either

* RevitAddInCommand.LookupCommandId(String)
* RevitAddInCommand.LookupPostableCommandId(PostableCommand)

The method

* UIApplication.CanPostCommand

identifies if the given command can ever be posted (that is, it is a member of PostableCommand or an external command).  It does not identify the command can be posted at the given timeframe (that is, it will not attempt to determine if the command is currently accessible).

**Dockable Dialog Panes**

Revit now allows addins to register WPF dialogs to dock into the top, left, right, and bottom of the main Revit window, as well as to be added as a tab to an existing system pane, such as the project browser.  Dialogs can be registered with UIApplication.RegisterDockablePane and a user-implementation of the IDockablePaneProvider interface.  Dockable panes can later be retrieved, shown, and hidden through UIApplication.GetDockablePane() and DockablePane.Show() and Hide().

**Dimensions & annotations API**

**Multi-reference annotations for rebar**

Revit now supports annotations pointing to more than one reference, consisting of a dimension and associated tag These annotations can be used to label and dimension Rebar elements, and are labeled in the user interface as “Multi-rebar annotations.”

The API exposes this through:

* MultiReferenceAnnotation - the annotation object instance. This class includes a reference to the associated dimension and tag element.
* MultiReferenceAnnotationType - the annotation type.  The type specifies the tag and dimension types to be used in the multi-reference annotation, as well as associated display settings.
* MultiReferenceAnnotationOptions - an options class used to create a new MultiReferenceAnnotation, including specification of the associated elements and options for the dimension and tag placement.
* IndependentTag.MultiReferenceAnnotationId  - returns the ElementId of the MultiReferenceAnnotation that owns a tag.
* Dimension.MultiReferenceAnnotationId - returns the ElementId of the MultiReferenceAnnotation that owns a dimension.

**Dimension alternate units**

New properties of DimensionStyle provide access to alternate units settings:

* DimensionStyle.AlternateUnits
* DimensionStyle.GetAlternateUnitsFormatOptions()
* DimensionStyle.SetAlternateUnitsFormatOptions()
* DimensionStyle.AlternateUnitsPrefix
* DimensionStyle.AlternateUnitsSuffix

**Dimension unit type**

The property:

* DimensionStyle.UnitType

returns the type of units supported by this dimension style (length, angle, or slope).

**Add-ins and macros**

**Automatic load of add-ins without restarting Revit**

Revit now automatically loads addins from newly added .addin manifest files without restarting Revit.

After placing a new .addin manifest file into one of the dedicated addins folders, the running Revit session will attempt to load the corresponding addin.  Loaded ExternalCommands will be added to the External Tools pulldown menu.  ExternalApplication and ExternalDBApplication OnStartup methods will be executed upon loading.  An addin's installer may leverage this feature - but it must do so by creating the new .addin file at the end of installation to the dedicated Revit addin folders (the installer must ensure that the addin's assembly is already deployed to the target machine and the assembly path can be resolved in the add-in manifest file).

Add-ins may decline the ability for Revit to load the .addin in the middle of a session.  To decline this options, add an <AllowLoadingIntoExistingSession> tag (set to “NO”) in the add-in manifest file.

Note that when Revit starts an add-in in the middle of the session, some add-in logic may not function identically because of the different interactions with the session.  Specifically:

* If the application's goal is to prevent something from happening, the application may not be able to handle the fact that this activity has already happened in the existing session.
* If the application's goal is to manage external information in synch with documents loaded in the session, the application may not be able to handle documents that were loaded before the application started.
* If the application's logic depends on the ApplicationInitialized event, this event has already been called before the add-in was loaded.

Also, some add-ins may not be able to fully initialize when loading in the middle of the session.  This is because some activities must take place at the start of the Revit session:

* Creation of custom failure definitions
* Establishment of a system-wide IFailureProcessor to handle all failures.
* Registering ExternalServices.

Revit also offers a new UI API method

* UIApplication.LoadAddIn()

to programmatically load the add-in(s) listed in the provided add-in manifest file.   Typically, this API would not be needed because Revit is already loading new .addin files automatically.

**MacroManager API**

Revit now support an API for listing, creating, removing, editing, debugging, and running macros through several supporting classes and enums

* MacroManager  
  + Available at the DB.Document or ApplicationServices.Application level; manages the querying, creation, and removal of macro modules
* UIMacroManager  
  + Available at the UIDocument or UIAppliciation level, manages adding, removing, and editing UI-level modules.
* MacroModule  
  + Manages the querying, creation, removal, and running of specific macro methods
* ModuleSettings
  + A collection of top-level properties of a MacroModule
* Macro  
  + Represents a single, runnable macro method.
* UIDocumentMacroOptions  
  + Represents security options for UIDocument-level macros.
* DocumentMacroOptions  
  + Represents security options for DB.Document-level macros.
* ApplicationMacroOptions  
  + Represents security options for Application-level macros
* MacroLanguageType  
  + Represents the language of a given macro, C#, VB.NET, Ruby, or Python
* MacroLevel  
  + Represents whether a macro is associated with a document the Revit application
* ModuleStatus
  + Represents the compiled, loaded, or built status of a MacroModule.

 The MacroManager APIs are all in the Autodesk.Revit.DB.Macros namespace and are available in RevitAPIMacros.dll and RevitAPIUIMacros.dll

**Macro Attributes**

The attributes

* Autodesk.Revit.UI.Macros.AddinAttribute
* Autodesk.Revit.UI.Macros.VendorIdAttribute

and the interface

* Autodesk.Revit.UI.Macros.IEntryPoint

have been moved to the namespace Autodesk.Revit.DB.Macros.

**Parameters**

**Shared parameter - create with specified GUID**

The new method

* Definitions.Create(String, ParameterType, bool, GUID)

supports creation of new ExternalDefinition objects (shared parameter definitions) which use the designated GUID instead of a random GUID. This allows an application to programmatically create consistent definitions for shared parameters without a shared parameter file or copying elements from one project to another.

**Dimension.Label**

The property

* Dimension.Label

Has been replaced by a new property

* Dimension.FamilyLabel

As this label applies only to dimension in families and their corresponding family parameter.

**Family Parameters**

The new property

* FamilyParameter.IsShared

identifies if the family parameter is a shared parameter.

**Geometry APIs**

**JoinGeometryUtils**

Revit now supports APIs for joining, unjoining, querying join state, and changing join order of elements in a model through the JoinGeometryUtils class.

**Extensible Storage**

**ExtensibleStorage API changes**

The method

* Element.GetEntitySchemaGuids()

returns the Schema Guids of any Entities present on an Element.

The methods

* Schema.GetField()
* Schema.ListFields()

are now restricted based on the read permission defined in the Schema.

**ExtensibleStorageFilter**

An ExtensibleStorageFilter is a new ElementQuickFilter for finding elements that contain an extensible storage entity of a given Schema Guid.

**Translation**

**Export to Navisworks**

The new function:

* Document.Export(String, String, NavisworksExportOptions)

exports a Revit project to the Navisworks .nwc format.  Note that in order to use this function,you must have a compatible Navisworks exporter add-in registered with your session of Revit.  If there is no compatible exporter registered, the function will throw OptionalFunctionalityNotAvailableException.  Use

* OptionalFunctionalityUtils.IsNavisworksExporterAvailable()

to check if there is an exporter registered.

**Import/Link SAT**

The functions:

* Document.Import(String, SATImportOptions, View)
* Document.Link(String, SATImportOptions, View)

import or link an SAT file into the document.

**Import/Link SketchUp**

The functions:

* Document.Import(String, SKPImportOptions, View)
* Document.Link(String, SKPImportOptions, View)

import or link an SKP file into the document.

**Import DWF Markups**

The function:

* Document.Import(String, DWFImportOptions)

imports DWF markups into the document.

**Export tables**

The new classes

* ExportLayerTable
* ExportLinetypeTable
* ExportPatternTable
* ExportFontTable
* ExportLineweightTable

expose read and write access to the tables used for mapping on export to various formats such as DWG, IFC and DGN.

**Site**

**Editing a TopographySurface**

Editing the points in a TopographySurface now requires establishment of an edit scope. The class

* TopographyEditScope

allows an application to create and maintain an editing session for a TopographySurface. Start and end of a TopographyEditScope will also start and end a transaction group. After the TopographyEditScope is started, an application can start transactions and edit the topography surface. Individual transactions the application creates inside TopographyEditScope will not appear in the undo menu.  All transactions committed during the edit mode will be merged into a single one which will bear the given name passed into TopographyEditScope constructor.

The method:

* TopographySurface.AddPoints()

has been modified to operate with the edit scope, and thus cannot be called outside of an edit scope.  The function no longer regenerates the document.

The new methods:

* TopographySurface.DeletePoints()
* TopographySurface.MovePoint()
* TopographySurface.MovePoints()
* TopographySurface.ChangePointElevation()
* TopographySurface.ChangePointsElevation()

provide the ability to modify and delete existing points in the TopographySurface.  Multiple calls to these functions can be included in the same edit scope.

**Reading points from a TopographySurface**

The new method:

* TopographySurface.ReadPoints()

returns a list of the points stored in the surface.

The method:

* TopographySurface.FindPoints()

filters and returns only the points of the topography surface which lie within the input bounding box.

The method:

* TopographySurface.ContainsPoint()

identifies if the input point is found in points stored in the surface.

The methods:

* TopographySurface.GetBoundaryPoints()
* TopographySurface.IsBoundaryPoint()

identify points which are a part of the boundary of the surface.

**Validation**

Several new validation methods have been added to the TopographySurface class:

* IsValidRegion() - Identifies whether the points can construct a valid region for a topography surface.
* ArePointsDistinct() - Identifies whether the input points are distinct in XY location.

These methods are newly used in AddPoints() to prevent problematic inputs from causing issues.

**SiteSubRegion**

The class SiteSubRegion represents a proxy class exposing the interfaces needed to access details of a subregion.   In the Revit database, both TopographySurface elements and subregion elements are represented by the same TopographySurface element subclass, however, in the Revit API, this SiteSubRegion class exists to separate the interfaces for subregions from those of topography surfaces.  The SiteSubRegion class offers these APIs:

* SiteSubRegion.Create() - creates a new subregion given a boundary consisting of one or more curve loops.
* SiteSubRegion.IsValidBoundary() - validates the input boundary for a new subregion
* SiteSubRegion.GetBoundary() - returns the boundary curves
* SiteSubRegion.SetBoundary() - sets the boundary for the subregion
* SiteSubRegion.TopographySurface - the TopographySurface element which this subregion represents
* SiteSubRegion.HostId - the identifier of the topography surface hosting this subregion

**BuildingPad**

The class BuildingPad represents a building pad element in Revit.  It inherits from CeilingAndFloor and thus offers inherited capabilities from HostObject as well. The API exposes the following specific capabilities around BuildingPads:

* BuildingPad.Create() - Creates a new BuildingPad given a boundary, type and level.
* BuildingPad.GetBoundary() - Returns the sketched boundary curves.
* BuildingPad.SetBoundary() - Assigns a new boundary to the BuildingPad.
* BuildingPad.HostId - The element id of the topography surface hosting this BuildingPad.

The type of the building pad is represented by BuildingPadType.  This inherits from HostObjAttributes and offers inherited capabilities from this parent, including access to the CompoundStructure of the type.  The API exposes these new capabilities for BuildingPadTypes:

* BuildingPadType.CreateDefault() - Creates a new default BuildingPadType in the document.

**MEP**

**Externalized Calculations**

Several new interfaces and classes based on external services are supported for pipe and duct calculations.

* Several new interfaces deriving from IExternalServer to support friction, flow, and pressure drop for pipes and ducts have been added.
  + IPipePlumbingFixtureFlowServer
  + IPipePressureDropServer
  + IDuctPressureDropServer
  + IDuctFittingAndAccessoryPressureDropServer
  + IPipeFittingAndAccessoryPressureDropServer
  + IDuctFittingAndAccessoryPressureDropUIServer
  + IPipeFittingAndAccessoryPressureDropUIServer
* Several new classes implementing IExternalData to support friction, flow, and pressure drop for pipes and ducts have been added.
  + PipePlumbingFixtureFlowData
  + PipePressureDropData
  + DuctPressureDropData
  + DuctFittingAndAccessoryConnectorData
  + PipeFittingAndAccessoryConnectorData
* Several new classes implementing ISingleServerService to support friction, flow, and pressure drop for pipes and ducts have been added.  
    
  + PipePlumbingFixtureFlowService
  + PipePressureDropService
  + DuctPressureDropService
  + DuctFittingAndAccessoryPressureDropService
  + PipeFittingAndAccessoryPressureDropService
  + DuctFittingAndAccessoryPressureDropUIService
  + PipeFittingAndAccessoryPressureDropUIService

**ElectricalLoadClassificationData**

ElectricalLoadClassificationData has several new string properties corresponding to the load classification section of the electrical panel schedule.

* LoadSummaryDemandFactorLabel
* PanelConnectedLabel
* PanelEstimatedLabel
* PanelConnectedCurrentLabel
* PanelEstimatedCurrentLabel
* ActualElecricalLoadNameLabel

**CSV Fitting Parameter Removal**

Because CSV files are no longer used to drive MEP fitting parameters, Revit supports a new set of APIs to manage fitting parameters through several classes

* FamilySizeTableManager - manages importing and exporting of legacy CSV data and size tables.
* FamilySizeTable - manages specific sizes of fittings.
* FamilySizeTableColumn - manages a specific dimension of a given size in a size table
* FamilySizeTableErrorInfo - reports any errors when importing an file with CSV size table into a FamilySizeTable

**Fitting Angle Settings**

The members:

* DuctSettings.FittingAngleUsage
* DuctSettings.GetSpecificFittingAngles()
* DuctSettings.GetSpecificFittingAngleStatus()
* DuctSettings.SetSpecificFittingAngleStatus()
* PipeSettings.FittingAngleUsage
* PipeSettings.GetSpecificFittingAngles()
* PipeSettings.GetSpecificFittingAngleStatus()
* PipeSettings.SetSpecificFittingAngleStatus()
* ElectricalSetting.FittingAngleUsage
* ElectricalSetting.GetSpecificFittingAngles()
* ElectricalSetting.GetSpecificFittingAngleStatus()
* ElectricalSetting.SetSpecificFittingAngleStatus()

provide access to fitting angle usage settings for ducts, pipes, cable trays and conduits.

**Duct Settings**

Duct settings for size prefixes and suffixes, annotations, and  air density and viscosity may now be set through the DuctSettings class.

**Curve Creation**

Pipes, Ducts,  FlexPipes, and FlexDucts can now be created with a  Pipe, Duct, FlexPipe, or FlexDuct SystemTypeId as a parameter on their respective static Create() methods.

**ConnectorElement**

The methods

* ConnectorElement.CreatePipeConnector(Document, Document, PipeSystemType, Reference, Edge)
* ConnectorElement.CreateDuctConnector(Document, DuctSystemType, ConnectorProfileType, Reference, Edge)
* ConnectorElement.CreateElectricalConnector(Document, ElectricalSystemType, Reference, Edge)
* ConnectorElement.CreateConduitConnector(Document, Reference, Edge)
* ConnectorElement.CreateCableTrayConnector(Document, Reference, Edge)

allow creation of connector elements in families on centered on internal loops of a given face.

The property

* ConnectorElement.SystemType

accesses the MEPSystemType of the connector

The members:

* ConnectorElement.Direction
* ConnectorElement.FlipDirection()

access the direction of the connector.

**Connect Air Terminal to Duct**

The new method:

* MechanicalUtils.ConnectAirTerminalOnDuct()

connects an air terminal to a duct directly (without the need for a tee or takeoff). The current location of the air terminal will be projected to the duct centerline, and if the point can be successfully projected, the air terminal will be placed on the most suitable face of the duct.

**General**

* CableTray.CurveNormal returns the "up" direction vector of a cable tray segment.
* RoutingPreferenceGroup now supports a new rule type for Cap fittings.
* Caps can be automatically be placed on all open connections of a pipe with PlumbingUtils.PlaceCapOnOpenEnds(Document, ElementId, ElementId)

**Reinforcement API**

There are many new reinforcement and rebar-related methods and classes in Revit 2014.

AreaReinforcement has several new methods and properties.

* AreaReinforcement.Create – creates a new AreaReinforcement object from curves.
* AreaReinforcement.IsUnobscuredInView - checks if Area Reinforcement is shown unobscured in a view.
* AreaReinforcement.SetUnobscuredInView - sets Area Reinforcement to be shown unobscured in a view.

PathReinforcement has several new methods and properties.

* PathReinforcement .Create – creates a new PathReinforcement object from curves.
* PathReinforcement .AdditionalOffset - gets and sets length offset.
* PathReinforcement .IsUnobscuredInView - checks if PathReinforcement is shown unobscured in a view.
* PathReinforcement .SetUnobscuredInView - sets PathReinforcement to be shown unobscured in a view.

FabricArea has several new methods and properties.

* FabricArea.Create - now supports an origin point of direction parameter.
* FabricArea.GetReinforcementRoundingManager - returns an object for managing reinforcement rounding override settings.

FabricSheet has several new methods and properties.

* FabricSheet.Create() - new static factory method for creation.
* FabricSheet.GetLocalCoordinateSystem()
* FabricSheet.SetLocalCoordinateSystem()
* FabricSheet.PlaceInHost()
* FabricSheet.GetReinforcementRoundingManager() - returns an object for managing reinforcement rounding override settings.
* FabricSheet.HostId
* FabricSheet.FabricLocation - the FabricSheet location in the host.
* FabricSheet.CoverOffset - the additional cover offset of the FabricSheet.
* FabricSheet.IsCoverOffsetValid() - identifies if the specified value is valid for use as a cover offset.

FabricSheetType

* FabricSheetType.GetReinforcementRoundingManager() - returns an object for managing reinforcement rounding override settings.

Rebar has several new methods and properties.

* Rebar.ComputeDrivingCurves() - compute the driving curves.
* Rebar.GetRebarConstraintsManager() - returns an object for managing the external constraints on the Rebar element.
* Rebar.GetReinforcementRoundingManager()  - returns an object for managing reinforcement rounding override settings.
* Rebar.HookAngleMatchesRebarShapeDefinition() - checks that the hook angle of the specified RebarHookType matches the hook angle used in the Rebar's RebarShape at the specified end of the bar.
* Rebar.CanUseHookType - checks if the specified RebarHookType id is of a valid RebarHookType for the Rebar's RebarBarType.
* Rebar.ConstraintsCanBeEdited - returns true, if the Rebar element's external constraints are available for editing.
* Rebar.GetHookOrientation() - returns the orientation of the hook plane at the start or at the end of the rebar with respect to the orientation of the first or the last curve and the plane normal.
* Rebar.SetHookOrientation() - defines the orientation of the hook plane at the start or at the end of the rebar with respect to the orientation of the first or the last curve and the plane normal.

RebarType has several new methods and properties.

* RebarType.SetHookTangentLength() - identifies the hook tangent length for a hook type.
* RebarType.GetReinforcementRoundingManager() - returns an object for managing reinforcement rounding override settings.

RebarBendData

* RebarBendData – new constructor to specify hook orientation.

RebarHandleType - a new enum to represent  the various types of handles on a Rebar instance that can be joined to References.

RebarConstrainedHandle  - a new class to represent a handle on a Rebar that can be joined to a reference.

RebarConstraintType  - a new enum to represent the various types of constraints that can be applied to a RebarConstrainedHandle.

RebarConstraintTargetHostFaceType - a new enum  to identify the individual face on a host element to which a Rebar handle is constrained.

RebarConstraint - a new class representing a constraint on the position of a Rebar Element or one of its handles.

RebarConstraintsManager -a  new class used to obtain information about the constrained shape handles (RebarConstrainedHandles) on a Rebar element.

RebarDeformationType - new enum representing the type of deformation of rebar.

RebarHookType.HookAngleInDegrees - a new property representing the angle of a rebar hook.

RebarInSystem

* RebarInSystem.getReinforcementRoundingManager - returns an object for managing reinforcement rounding override settings.
* RebarInSystem.setUnobscuredInView - sets the element to be shown unobscured in a view.

RebarShape

* RebarShape.GetDefaultHookAngle - get the hook angle, expressed as an integral number of degrees (common values are 0, 90, 135, and 180).
* RebarShape.GetDefaultHookOrientation - gets the default rebar hook orientation.

ReinforcementRoundingSource - a new enum listing the possible sources for reinforcement rounding overrides.

ReinforcementRoundingManager - a new class providing access to reinforcement rounding overrides for structural elements.

RebarRoundingManager - new class providing access to element reinforcement roundings overrides.

FabricRoundingManager - a new class providing access to element reinforcement roundings overrides.

ReinforcementSettings

* ReinforcementSettings.GetRebarRoundingManager - returns an object for managing reinforcement rounding override settings used by RebarBarTypes, Rebar and RebarInSystem elements.
* ReinforcementSettings.GetFabricRoundingManager -returns an object for managing reinforcement rounding override settings used by FabricSheetType and FabricSheet elements.

**Custom Export**

The Custom Export API provides access to the rendering output pipeline through which Revit sends a processed model (its graphics 3D representation) to an output device. In the case of Custom Export, the "device" is represented by a context object that could be any kind of a device, even a file (in the most common case, actually.) A client of Custom Export provides a context and invokes rendering of a model, upon which Revit starts processing the model and sends graphic data out via methods of the context. The data describes the model exactly as it would have appeared in Revit when the model is rendered. The data includes all geometry and material properties.

The following classes have been made available so far:

**CustomExporter**

A class that allows exporting 3D views via a custom export context. The Export method of this class triggers standard rendering process in Revit, but instead of displaying the result on screen or printer, the output is channeled through the given custom context that handles processing the geometric as well as non-geometric information.

**IExportContext**

An interface of which interface is used in a custom export of a Revit model. The instance of this class is passed in as a parameter of a CustomExporter. The methods are then called by Revit at times of exporting entities of the model.

**Render Node Classes**

Classes of which instance are sent to an export context during a custom export.

* RenderNode - base class for all output nodes
* ViewNode - represents a View
* GroupNode - base class for family and link instances
* InstanceNode - represents an instance of a family
* LinkNode - represents an external link
* ContentNode - base class from RPC, Light, and Daylight nodes
* RPCNode - represents an RPC object
* DaylightPortalNode - represents a daylight portal
* LightNode - represents a light
* MaterialNode - represents a material change

**CameraInfo**

A class that describes information about projection mapping of a 3D view to a rendered image. An instance of this class can be obtained via a property of ViewNode.

**Paint API**

The paint tool is now supported in the API

* Document.Paint
* Document.RemovePaint
* Document.IsPainted
* Document.GetPaintedMaterial

**Component repeater API**

Component repeaters can be used to replicate (repeat) elements hosted on repeating references. The result of the repeating operation is a collection of slots. Each slot contains one repeated component. The ComponentRepeater class provides the repeating functionality and access to the slots.

Each repeating reference is capable of hosting one point of an adaptive component. An initial pattern can be created by populating one or more repeating references with such points. Component repeaters can then be used to replicate the pattern to fill the rest of the repeating references in the particular repeating reference source.

The repeating references in repeating reference source are arranged in one or two dimensional arrays, allowing for different kinds of repeating:

* One dimensional source allows for repeating along a path.
* Two dimensional source allows for repeating across a grid.
* It is also possible to host a point on a zero dimensional reference (a point). This point will be shared by all slots. A zero dimensional source allows for repeating around a single point. It should not be used alone, but together with at least one other repeating reference source (typically one dimensional.) The point hosted on the zero dimensional source serves as a central point around which other points can be repeated on their respective repeating reference sources.

Multiple adaptive components may be hosted on one repeating reference source, and different points of one adaptive component may be hosted on different repeating reference sources, effectively allowing different points of an adaptive component to be repeated using different patterns.

The following classes provide access to the functionality of Component repeaters:

* Autodesk.Revit.DB.RepeatingReferenceSource
* Autodesk.Revit.DB.RepeaterBounds
* Autodesk.Revit.DB.RepeaterCoordinates
* Autodesk.Revit.DB.ComponentRepeater
* Autodesk.Revit.DB.ComponentRepeaterSlot

**Small enhancements & API interface changes**

**API validation**

**No transactions from outside threads**

Calling into the Revit API from outside threads and outside modeless dialogs has never been supported, but it was not strictly prohibited, meaning there would be no immediate exceptions when someone tries to modify model from outside of the supported API workflows. That has been changed. It is no longer possible to start a transaction unless the caller is inside a legitimate API call, such as an external command, event, updater, call-back, etc. An exception will be thrown if such attempt is made.

**IsValidObject property**

If a Revit native object contained within an API wrapper is destroyed, or creation of the corresponding native object is undone, the managed API object containing it is no longer valid. API methods cannot be called on invalidated wrapper objects. The property IsValidObject (added to many API classes) can be used to identify if the corresponding native object has gone out of scope.

**Enumerated type validation**

Automatic validation has been introduced for enumerated type inputs to API methods and properties. If an value is improperly cast to an enum and is not a valid member of that enum, an ArgumentOutOfRangeException will be thrown.

**Elements**

**Copy & paste elements**

The new methods:

* ElementTransformUtils.CopyElements(Document, ICollection<ElementId>, Document, Transform)
* ElementTransformUtils.CopyElements(View, ICollection<ElementId>, View, Transform)
* Transform ElementTransformUtils.GetTransformFromViewToView(View, View)

support copy and paste of arbitrary elements.  The first overload supports copy within documents, or from document to document.  The second overload also support copying within one document or between two documents, but specifically supports copy and paste of view-specific elements.

**Materials**

Materials applied with the Paint tool and their areas can be found by specifying "true" for the 'usePaintMaterial' argumement in Element.GetMaterialIds and Element.GetMaterialArea

**Geometry**

**FreeForm element**

A FreeFormElement is a form sub-type that contains non-parametric geometry created from an input solid outline. The element can be added to families, and can participate in joins and void cuts with other combinable elements.  Planar faces of the element can be offset interactively and programmatically in the face normal direction.

* FreeFormElement.Create() - creates a new FreeForm element.
* FreeFormElement.SetFaceOffset() - offsets a planar face a certain distance in the face normal direction.

**Solid & curve intersection**

The new method

* Solid.IntersectWithCurve()

calculates the intersection between a closed volume Solid and a curve. There is an option to return details about the segments inside the volume, or outside. Both the curve segments and the parameters of the segments are available in the results.

**Face/Face Intersection**

The method

* Face.Intersect()

calculates the intersection of the input face with this face and returns the intersection results.  The method can output the intersection geometry if the intersection consists of a single curve.

**ReferenceIntersector & RVT Links**

The new option findReferencesInRevitLinks allows ReferenceIntersector to find geometry in RVT links.

**Rulings of RuledFace**

The new method

* RuledFace.RulingsAreParallel

determines if the rulings of the ruled surface are parallel.

**Detail elements**

**Detail element draw order**

The class

* DetailElementOrderUtils

now includes routines to BringToFront, BringForward, SendToBack, SendBackward multiple detail members.  The draw order of the members does not change relative to one another.

**Stairs**

**StairsRunJustification**

New values LeftExterior and RightExterior have been added to this enum to support jusitification to the left and right supports.

**StairsLanding**

The new members:

* CreateAutomaticLanding(Document, ElementId, ElementId)
* CanCreateAutomaticLanding(Document, ElementId, ElementId)

provide automatic landing(s) creation and creation validation between two stairs runs.

**StairsRun**

The new properties:

* StairsRun.ExtensionBelowRiserBase
* StairsRun.ExtensionBelowTreadBase

represent the extension/trim value for the run, depending upon whether the run begins with a riser or tread.

These replace the deprecated property:

* StairsRun.ExtensionBelowBase

which now accesses whichever property listed above depending on the run's configuration.

**StairsComponentConnection**

The new methods:

* StairsRun.GetConnections()
* StairsLanding.GetConnections()

provide information about connections among stairs components (run to run, or run to landing).

**Parameters & filters**

**Parameter.AsValueString()**

This method previously was implemented for only Integer and Double values. It now can also be used with Enums and ElementIds. Optionally it can accept a FormatOptions object to specify units formatting of the returned string.

**Parameter.Definition.UnitType**

This new property provides access to the UnitType of a parameter definition.

**Parameter variance among group instances**

The new members:

* InternalDefinition.VariesAmongGroups
* InternalDefinition.SetVariesAmongGroups(Document)

support read and write to the option that the parameter value can vary across groups.  This can be changed only for non-built-in parameters.

**FilterCategoryRule**

The new class FilterCategoryRule can be used in the definition of a ParameterFilterElement.   It represents a filter rule that matches elements of a set of categories.

The related method:

* ParameterFilterElement.AllCategoriesFilterable()

has been replaced by

* FilterCategoryRule.AllCategoriesFilterable()

**Miscellaneous**

**ThermalAsset.SpecificHeat**

This new property provides the specific heat value of a thermal asset in feet per Kelvin, squared-second.

**AreaVolumeSettings**

This new class provides access to settings related to volume and area computations. The old VolumeCalculationSetting class is obsolete.

**Document.Delete()**

This method previously returned null if the element or elements could not be deleted. It now will throw an exception in this situation.

**Document level updaters**

The new methods:

* UpdaterRegistry.IsUpdaterRegister(UpdaterId, Document)
* UpdaterRegistry.UnregisterUpdater(UpdaterId, Document)

provide the ability to inquire about and to unregister updaters that are associated to specific documents.

**UIThemeManager**

The static properties of UIThemeManager provide access to the current UI theme and the default theme.

**Families & content**

**Family category**

The property

* FamilyBase.FamilyCategory

can now be set.  This allows the category of an family being edited to be changed.

**SpatialElementCalculationLocation**

The class SpatialElementCalculationLocation is used to specify the room or space where an element should be considered as placed.

This is a base class with currently subtypes of calculation location:

* SpatialElementCalculationPoint
* SpatialElementFromToCalculationPoints

A SpatialElementCalculationLocation can be added to the family by turning on the family's ROOM\_CALCULATION\_POINT parameter.   Once the location has been added, you can access the marker position using the MarkerPosition property.

Note that the MarkerPosition property is no longer settable - this position is determined automatically.

**SpatialElementFromToCalculationPoints**

SpatialElementFromToCalculationPoints is a subclass of SpatialElementCalculationLocation used to specify the search points for a family instance that connects two rooms or spaces, such as a door or window. The points determine which room or space is considered the "from" and which is considered the "to".    The properties ToPosition and FromPosition govern the coordinates for these points.

**Arc through points**

The method

* CurveByPointsUtils.CreateArcThroughPoints()

creates an arc curve through three input ReferencePoints.

**Events**

**DocumentChangedEventArgs**

For the methods GetAddedElementIds()/GetModifiedElementIds() - internal Revit element ids that are not visible to UI and API are now excluded from the return set.

**GetAddElementIds(ElementFilter)/GetModifiedElementIds(ElementFilter)**

The new methods:

* DocumentChangedEventArgs.GetAddedElementIds(ElementFilter)
* DocumentChangedEventArgs.GetModifiedElementIds(ElementFilter)

only return ElementIds that pass the input element filter. This helps applications detect only specific changes of interest.

**Reinforcement API**

**Reinforcement Length Tolerance**

The new classes:

* RebarRoundingManager
* FabricRoundingManager

contain settings for rounding tolerance for rebar and fabric reinforcement elements.  They can be assigned at the element instance level (Rebar, RebarInSystem, FabricArea, FabricSheet), at the type level (RebarType, FabricSheetType), or at the ReinforcementSettings level.   Lower level settings override the setting of higher levels.