Autodesk® Revit® Construction Modeling and API for Builders

Harry Mattison – Autodesk
Enrique Galicia – ICABIM

CP5211 Autodesk® Revit® Construction Modeling and API for Builders

Learning Objectives

At the end of this class, you will be able to:

- Apply Revit construction modeling functionality
- Describe how the Revit API can be used to automate design and analysis for construction
- Explain how Revit can work effectively with external databases
- Describe benefits of building information modeling (BIM) when applied to building construction

About the Speakers

Harry Mattison is a Principal Engineer for Autodesk on the Revit® API team. Harry joined Revit Technology Corp. in 1999 and was the Quality Assurance Manager for AutoCAD® Revit Architecture and Autodesk® Revit Structure for eight years. Since joining the Revit API team, Harry has worked on large projects including the Family API and Massing API for the 2010 release, the Analysis AutoCAD Visualization Suite Framework, Dynamic Model Update, and Failure API projects in 2011, and the Construction Modeling API in Revit 2012. Harry has contributed to design, development, testing, documentation, and customer support for the Revit API and is excited by the great potential that this technology offers to Autodesk customers.

Enrique Galicia is the VDC Manager of ICABIM. Enrique joined the ICABIM Team on 2009 and since his start has worked developing 4D Simulations, Project Management, Clash Detection and Scanning Tools processes for BIM. On April 2010 went to Autodesk® Revit API Training Course on Warsaw, Poland to develop BIM construction processes through external commands and applications. Since joining the ICABIM Team has worked on over 40 large projects including, Damps, Hospitals, Elevated Highways, Housing Developments, and Office Buildings. Enrique has contributed developing BIM Processes on fields as Project Management, Simulations, Scanning Techniques, Mobility, and Quality Management for ICABIM and believes that programming is the quantum Leap for developing BIM.

Introduction

Revit contains a wide variety of functionality that can be useful throughout the construction process. Scheduling, estimating, and coordination are a few of these areas, and there are many opportunities to use the Revit API to automate tasks related to construction modeling. The Revit API can help you perform construction modeling faster and more accurately.

Applying Revit construction modeling functionality

This handout focuses on the new Construction Modeling features in Revit 2012 that support workflows for improved 4D sequencing, cost estimating, quantification, and other construction-related tasks by allowing you to create parts, assemblies, and assembly views in your Revit model.

Parts

Parts are dividing elements that can be created in Revit hosts such as walls, ceilings, floor slabs, and roofs. Dividing one of these elements creates new elements called “parts which are associated with the element used to create them. This “parent” element is not deleted or altered by the creation of the parts – the parent continues to exist and can be edited after parts have been created from it.

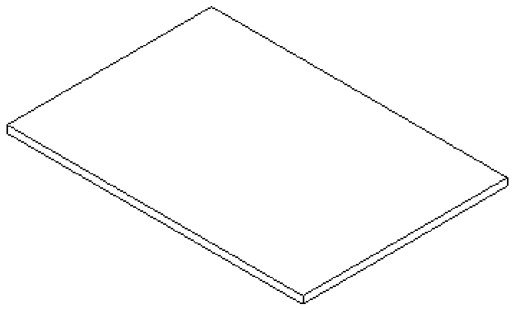
The importance of parts is that they allow your Revit model to reflect the difference between how objects are designed and how they are constructed. For example, a large floor slab may be modeled as a single floor element in Revit, but it may not be constructed as a single element at a single time during the construction schedule. By dividing the single floor into multiple parts, the model can better reflect the realities of the construction process and more construction data can be captured.

Parts can be divided into sub-parts using the UI or API by specifying curves that define the division or by providing a set of intersecting references such as levels, reference planes, or grids. Curves that divide parts can be created with either closed loops or open loops that intersect the boundaries of the parent element. These parts will allow you to improve the design intelligence in your Revit model so that it can be used to produce more granular and accurate schedules, quantity takeoffs, and graphical views of the model.

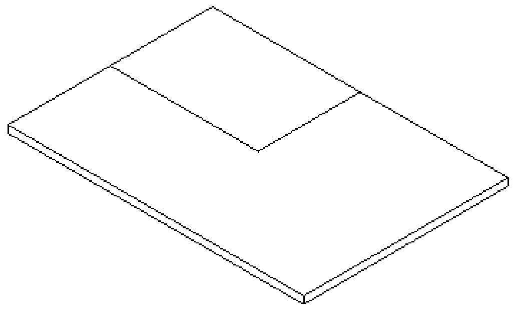
After parts are created and divided, many different attributes of these parts can be modified to accurately indicate how the building will be built. For example, the material of a part created from a floor slab may have a different material than the remainder of the slab.

After creating parts, part list schedules can be created to more accurately track cost estimation and quantification data in the BIM model.

This example shows how parts can be created and divided.



Before

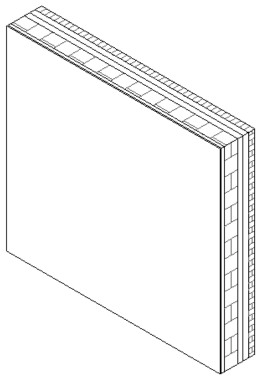
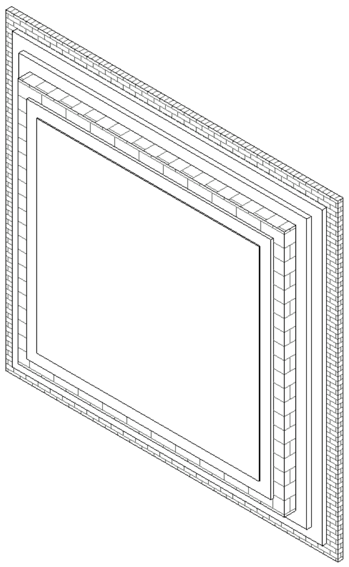


After

```
{
    Autodesk.Revit.ApplicationServices.Application app = commandData.Application.Application;
    Document doc = commandData.Application.ActiveUIDocument.Document;
    UIDocument uidoc = commandData.Application.ActiveUIDocument;

    IList<ElementId> idList = new List<ElementId>();

            // Select an element to use for part creation. Can be a host element (floor, roof, etc) or a 
part created from one of these elements.
            Element pickedElement = doc.GetElement(uidoc.Selection.PickObject(ObjectType.Element));
            idList.Add(pickedElement.Id);

            // Determine which elements to divide.
            ICollection<ElementId> elementIdsToDivide = new Collection<ElementId>();
            if (PartUtils.AreElementsValidForCreateParts(doc, idList))
            {
                // AreElementsValidForCreateParts returned true, so the selected element is not a part 
but it is an element that can be used to create a part.
                Transaction createPartTransaction = new Transaction(doc, "Create Part");
                createPartTransaction.Start();
                PartUtils.CreateParts(doc, idList); // create the parts
                createPartTransaction.Commit();

                elementIdsToDivide = PartUtils.GetAssociatedParts(doc, pickedElement.Id, true, true); // 
get the id of the newly created part
            }
            else if (pickedElement is Part)
            {
                // The selected element is a part, so that part will be divided.
                elementIdsToDivide.Add(pickedElement.Id);
            }

            // Create geometry that will be used to divide the part. For this example, a new part will 
be divided from the main part that is one quarter of the face. More complex intelligence could be coded
```

```
to divide the part based on construction logistics or the properties of the materials being used to
create the part.
            XYZ pointRight = null;
            XYZ pointTop = null;
            XYZ pointCorner = null;
            XYZ pointCenter = null;

            SketchPlane sketchPlane = null;

            Options opt = new Options();
            opt.ComputeReferences = true;
            Autodesk.Revit.DB.GeometryElement geomElem = pickedElement.get_Geometry(opt);
            foreach (GeometryObject geomObject in geomElem.Objects)
            {
                if (geomObject is Solid) // get the solid geometry of the selected element
                {
                    Solid solid = geomObject as Solid;
                    FaceArray faceArray = solid.Faces;
                    foreach (Face face in faceArray)
                    {
                        // find the center of the face
                        BoundingBoxUV bbox = face.GetBoundingBox();
                        UV center = new UV((bbox.Max.U - bbox.Min.U) / 2 + bbox.Min.U , (bbox.Max.V -
bbox.Min.V) / 2 + bbox.Min.V);
                        XYZ faceNormal = face.ComputeNormal(center);
                        if (faceNormal.IsAlmostEqualTo(XYZ.BasisZ)) // this example is designed to work
with a floor or other element with a large face whose normal is in the Z direction
                        {
                            Transaction sketchPlaneTransaction = new Transaction(doc, "Create Sketch
Plane");

                            sketchPlaneTransaction.Start();
                            sketchPlane = doc.Create.NewSketchPlane(face as PlanarFace);
                            sketchPlaneTransaction.Commit();

                            pointCenter = face.Evaluate(center);

                            UV top = new UV((bbox.Max.U - bbox.Min.U) / 2 + bbox.Min.U, bbox.Max.V);
                            pointTop = face.Evaluate(top);

                            UV right = new UV(bbox.Max.U, (bbox.Max.V - bbox.Min.V) / 2 + bbox.Min.V);
                            pointRight = face.Evaluate(right);

                            UV corner = new UV(bbox.Max.U, bbox.Max.V);
                            pointCorner = face.Evaluate(corner);

                            break;
                        }
                    }
                }
            }
            // Create the curves that will be used for the part division.
            IList<Curve> curveList = new List<Curve>();
            Curve curve1 = app.Create.NewLine(pointCenter, pointRight, true);
            curveList.Add(curve1);
            Curve curve2 = app.Create.NewLine(pointRight, pointCorner, true);
            curveList.Add(curve2);
            Curve curve3 = app.Create.NewLine(pointCorner, pointTop, true);
            curveList.Add(curve3);
            Curve curve4 = app.Create.NewLine(pointTop, pointCenter, true);
            curveList.Add(curve4);

            // intersectingReferenceIds will be empty for this example.
            ICollection<ElementId> intersectingReferenceIds = new Collection<ElementId>();

            // Divide the part
            Transaction dividePartTransaction = new Transaction(doc, "Divide Part");
            dividePartTransaction.Start();
            PartMaker maker = PartUtils.DivideParts(doc, elementIdsToDivide, intersectingReferenceIds, 
curveList, sketchPlane.Id);
            dividePartTransaction.Commit();
            ICollection<ElementId> divElems = maker.GetDividedElementIds(); // Get the ids of the 
divided elements

            // Set the view's "Parts Visibility" parameter so that parts are shown
            Parameter partVisInView = 
doc.ActiveView.get_Parameter(BuiltInParameter.VIEW_PARTS_VISIBILITY);
            Transaction setPartVizTransaction = new Transaction(doc, "Set View Parameter");
            setPartVizTransaction.Start();
            partVisInView.Set(0); // 0 = Show Parts, 1 = Show Original, 2 = Show Both
            setPartVizTransaction.Commit();

            return Result.Succeeded;
        }
```

Faces of parts (which correspond to layers of a compound host object) can be offset via the API to achieve the same result that can be obtained in the UI by dragging the part’s shape handles. This functionality provides customized control beyond the default layer configuration created by Revit.

	
Wall that has been converted to parts	Parts after running command to offset faces

```
Autodesk.Revit.DB.GeometryElement geomElem = part.get_Geometry(new Options());
foreach (GeometryObject geomObject in geomElem.Objects)
{
    if (geomObject is Solid)
    {
        Solid solid = geomObject as Solid;
        FaceArray faceArray = solid.Faces;
        foreach (Face face in faceArray)
        {
            if (part.CanOffsetFace(face))
                part.SetFaceOffset(face, offsetCtr);
            offsetCtr += 0.1;
        }
    }
}
```

Assemblies and Assembly Views

Assemblies are collections of elements that can be used in the Revit model to create shop drawings for prefabricated building components such as pre-cast walls, columns, beams, and floors. Assemblies can be scheduled, visually isolated, and tagged.

Every assembly has a primary element that determines the assembly's category. Each assembly is shown in the Project Browser and can be used to create assembly views that display only that assembly.

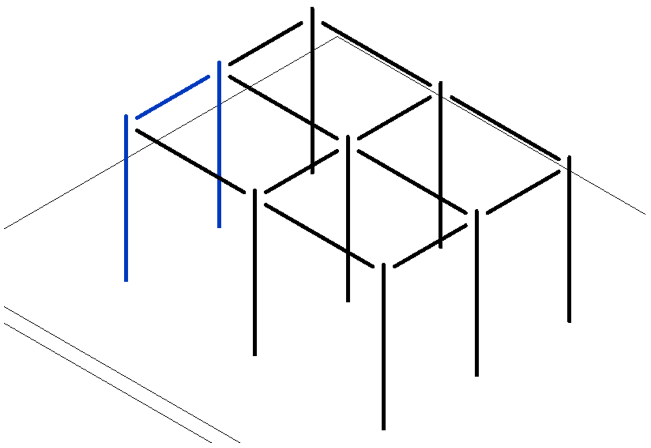
Read, write and create access to assemblies in the Revit environment is provided through the classes:

- Autodesk.Revit.DB.Assembly.AssemblyInstance

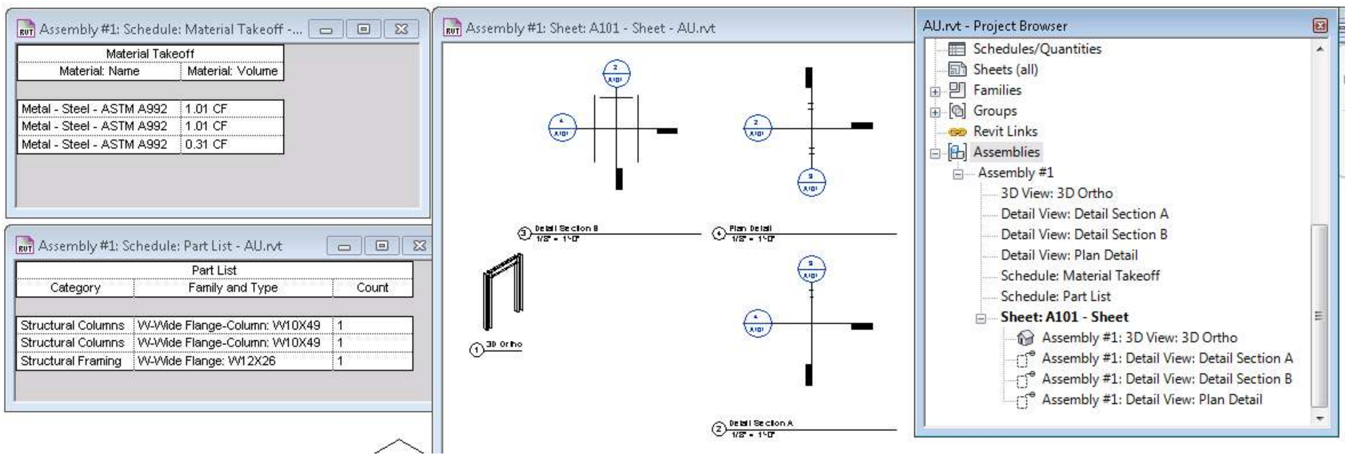
- Autodesk.Revit.DB.Assembly.AssemblyType

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

Assembly Views display only the elements in the assembly and are created with methods of the AssemblyViewUtils class. The code below shows how assemblies and assembly views can be created with the API.



Two columns and a beam which are selected before running the command



Assembly #1: Schedule: Material Takeoff -...

Material Takeoff	
Material: Name	Material: Volume
Metal - Steel - ASTM A992	1.01 CF
Metal - Steel - ASTM A992	1.01 CF
Metal - Steel - ASTM A992	0.31 CF

Assembly #1: Schedule: Part List - AU.rvt

Part List		
Category	Family and Type	Count
Structural Columns	W-Wide Flange-Column: W10X49	1
Structural Columns	W-Wide Flange-Column: W10X49	1
Structural Framing	W-Wide Flange: W12X26	1

Assembly #1: Sheet: A101 - Sheet - AU.rvt

AU.rvt - Project Browser

```
Schedules/Quantities
Sheets (all)
Families
Groups
Revit Links
Assemblies
  Assembly #1
    3D View: 3D Ortho
    Detail View: Detail Section A
    Detail View: Detail Section B
    Detail View: Plan Detail
    Schedule: Material Takeoff
    Schedule: Part List
    Sheet: A101 - Sheet
      Assembly #1: 3D View: 3D Ortho
      Assembly #1: Detail View: Detail Section A
      Assembly #1: Detail View: Detail Section B
      Assembly #1: Detail View: Plan Detail
```

After running the command

```
        Document doc = commandData.Application.ActiveUIDocument.Document;
        UIDocument uidoc = commandData.Application.ActiveUIDocument;

        ElementId categoryId =
doc.get_Element(uidoc.Selection.GetElementIds().FirstOrDefault()).Category.Id;
```

```
            if (AssemblyInstance.IsValidNamingCategory(doc, categoryId, 
uidoc.Selection.GetElementIds()))
            {
                Transaction transactionA = new Transaction(doc, "Create Assembly");
                transactionA.Start();
                AssemblyInstance assemblyInstance = AssemblyInstance.Create(doc, 
uidoc.Selection.GetElementIds(), categoryId);
                transactionA.Commit(); // need to commit the transaction to complete the creation of 
the assembly instance so it can be accessed in the code below

                Transaction transactionB = new Transaction(doc, "Create Assembly Views");
                transactionB.Start();
                assemblyInstance.AssemblyTypeName = "Assembly #1"; // rename the assembly
                if (assemblyInstance.AllowsAssemblyViewCreation()) // check to see if views can be 
created for this assembly
                {
                    ElementId titleblockId = 
doc.TitleBlocks.Cast<FamilySymbol>().First<FamilySymbol>().Id; // find a titleblock
                    // create a sheet, 3d view, sections, a material takeoff, and parts list for this 
assembly
                    ViewSheet viewSheet = AssemblyViewUtils.CreateSheet(doc, assemblyInstance.Id, 
titleblockId);
                    View3D view3d = AssemblyViewUtils.Create3DOrthographic(doc, assemblyInstance.Id);
                    ViewSection detailSectionA = AssemblyViewUtils.CreateDetailSection(doc, 
assemblyInstance.Id, AssemblyDetailViewOrientation.DetailSectionA);
                    ViewSection detailSectionB = AssemblyViewUtils.CreateDetailSection(doc, 
assemblyInstance.Id, AssemblyDetailViewOrientation.DetailSectionB);
                    ViewSection detailSectionH = AssemblyViewUtils.CreateDetailSection(doc, 
assemblyInstance.Id, AssemblyDetailViewOrientation.HorizontalDetail);
                    View materialTakeoff = AssemblyViewUtils.CreateMaterialTakeoff(doc, 
assemblyInstance.Id);
                    View partList = AssemblyViewUtils.CreatePartList(doc, assemblyInstance.Id);

                    // Add graphical views to the newly created sheet. Schedules (the Parts List and 
Material Takeoff) cannot be added to sheets with the 2012 API.
                    viewSheet.AddView(view3d, new UV(1, 1));
                    viewSheet.AddView(detailSectionA, new UV(1.3, 1));
                    viewSheet.AddView(detailSectionB, new UV(1, 1.3));
                    viewSheet.AddView(detailSectionH, new UV(1.3, 1.3));
                }
                transactionB.Commit();
            }
            return Result.Succeeded;
```