AS323523

How to document a tower with forty different floor plates?

Maciej Wypych
BVN

Adam Walmsley
Autodesk

Learning Objectives

- Learn how to overcome modeling issues with concept models
- Learn about best practice methodologies for undertaking complex Dynamo scripting
- Learn how to use the power of computational design to model complex geometries
- Learn about data-management processes with Dynamo

Description

This course will present a case study of a residential tower in Sydney, Australia. A unique design with completely different floors required a new approach to documentation. Dynamo had been extensively used to create and manage all the unique elements. More than 45% of all Revit elements were created using automated workflows. This course will focus on understanding the challenges in automation of such a vast amount of Revit elements; learning how to choose between an intelligent or brute force approach; and what we have learned by documenting the same building twice. Understand how a variety of multiple Dynamo definitions had been used to create specific elements. Learn efficiency differences between groups and links. The design of a multistory tower can be challenging without every floor being different. Learn how we overcame this challenge and optimized our workflows to enable multiple design changes in the process. This class will focus on ways of improving efficiency through automation.

Speakers

Maciej Wypych is a Design Technology Coordinator at BVN. Prior to joining BVN Maciej was a Studio BIM Manager for Warren and Mahoney in Sydney. Maciej is also a sessional Tutor at University of New South Wales. He is a committee member and frequent speaker at Dynamo User Group Sydney as well as BUILT ANZ, Wellington Digital Design User Group and other conferences. He has over 15 years' experience in the architecture and building industry in Australia and UK.

Adam Walmsley is a Civil Infrastructure Technical Specialist for Autodesk based in Sydney, Australia. He focuses on enabling the industry to design, document and construct Civil Infrastructure projects using improved connected workflows. His passion lies in using computational design processes to help automate and improve design.
Prior to joining Autodesk in April 2019, Adam had worked for 12 years in the structural design and civil construction industry, holding lead Digital Engineering roles on challenging major projects in Australia.

Initial model split

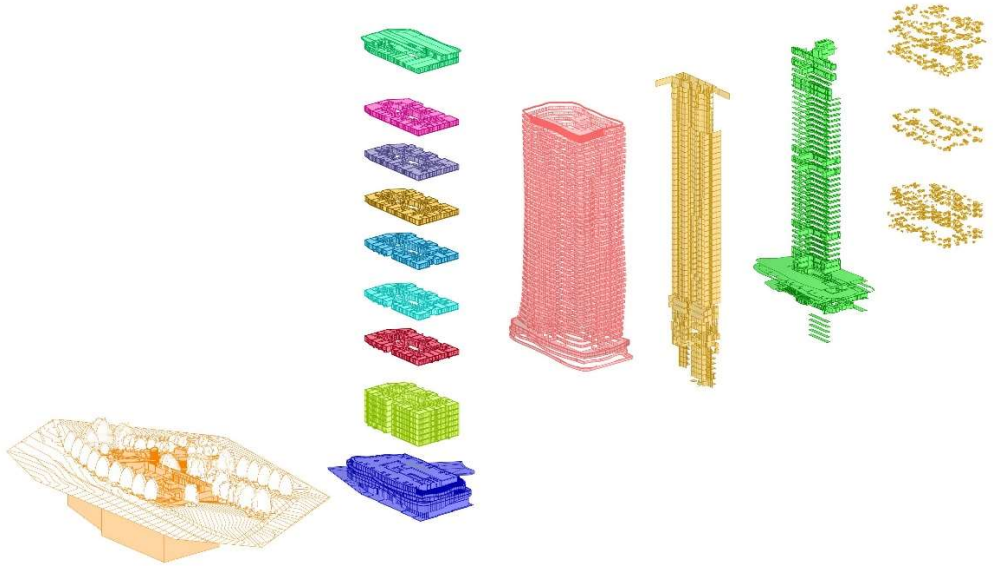


Figure 1 Initial model split

Our model was initially created using multiple links. The links included site, typical floor levels, core, interior, façade and furniture on typical levels. All of these elements were combined in a federated model. We've also had a container model with all families to maintain consistency.
During documentation phase individual models were performing reasonably well, however each of them had several links loaded in. A team member editing a typical floor would have at least the façade and the core loaded in and more often the floor above below and above for reference.
Due to the tight and intense project program, all models were rapidly changing.
This resulted in multiple issues. First, users had to reload other links to coordinate between models. This removed any speed benefits of this split. Also, continuous changes and reloads resulted in annotation elements dropping off.

As the project had to be completely redesigned, it was an unique opportunity to change the strategy. This time we've decided to minimize the number of models and use groups to model apartments and other items repeated across multiple levels

Final model split

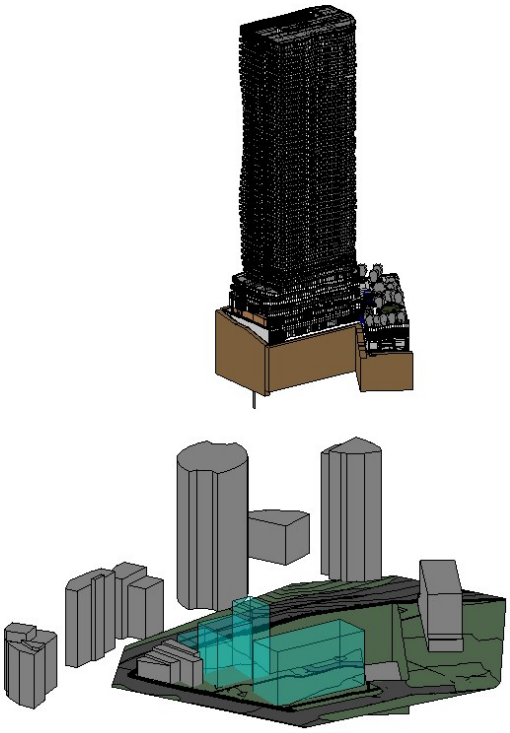


Figure 2 Final model split

We've applied the best practice rules for using groups to maintain consistency and minimize errors based on the best practices listed in RevitForum

Planning:

- Think carefully and plan how you want to break up the building into groups. Sometimes it is better not to put some elements into a group.
- Disallow joins in all walls on the outer extent of the group – use join tool instead
- The larger the groups, the slower the file will perform and the greater the damage if there is a problem.
- The greater the number of instances of a group, the slower the file will perform.
- Preferably have a group that is just the core/floor containing lift, corridor, façade etc. and then have groups for each apartment types.
- Don't put the apartment groups in the floor group.
- Don't Exclude items from groups. It's impossible to know if it's intentional.
- Be aware that groups do not have a Phase, but the objects within a group all have their own Phases – which can vary and can vary across group instances.
- Don't add elements to groups if they generate warnings
- Don't use "Fix Groups" option, create duplicates and replace with original.
- Parameters that are not geometric can vary by group instance: Text, URL, Materials, Currency. Make sure that you select

Isolation:

- Ensure all elements in the group are hosted to the same Level.
- Don't group elements without the elements they are hosted or constrained to. Eg. put the Doors in the same group as the Wall they hosted in.
- Don't pin groups to elements outside the group.
- Don't pin or lock elements within the group to elements outside the group.
- Ensure no walls are joined to walls outside the group. Use Disallow Join to ensure walls in the group kiss walls outside the group but do not join them.
- It can be better not to set the Top Constraint of a Wall to be the Level Above. Instead, use the Current Level plus Unconnected Height as appropriate

Elements:

- Try to avoid using Line-based families.
- Be careful with Face-based families. It may be the case that each Face-Based item must be hosted to an individual Wall or Reference Plane. In other words, you can host one Face-Based item (eg basin) to a wall, but you may/will have to create an individual Reference Plane for each other Face-Based items (eg toilet, towel rail etc) you wish to host to that wall.

- If your groups need to be copied up/down (such as in a multi-level building) then ensure that the vertical extents of any Reference Plane's used for Face-Based families (see above) are within a single level's Floor to Floor extents so that there is no overlapping when the copying occurs.
- Avoid using WorkPlane-based families in groups. If you must, then ensure that the item or plane they are hosted to is also in the group.
- Be cautious of putting floors or stairs in groups as copying to new levels can be tricky. If you do have a floor in the group, don't lock the floor sketch lines to other objects.
- Be careful if you have a floor waste in a bathroom group: it may lock people out from editing or dimensioning a concrete setout plan.
- Ensure grouped elements are in the same Workset. Group instances also belong to a Workset – ensure they are all on the same one.
- When creating families to go into groups, preference should be given to un-hosted objects, although we do acknowledge that we do use many Face-based items.
- If you are using a family wall, floor or ceiling hosted ensure that host is within the group, otherwise create and select reference plane or use level as a host

Manipulation:

- Mirroring and rotating groups can be problematic, especially with faced-based families. Check your results after you have performed an operation. If there is an issue, it can be better to edit the flipped version of the group and fix the problem there.
- Be careful when you Copy and Paste, or Create Similar as the new elements sometimes don't end up on the same host Level. Note also, that you can't Re-host an element when in the Group Edit mode: you have to remove it from the group, re-host it and then put it back in the group. This is especially true for Line-based and Face-based families.
- If you have a nested group with a level problem, you may have to remove it from all the groups it is hosted into (such as a table and chair group nested into several apartment types), fix the problem and then put it back into all the other groups.
- One way to check on the group Level problem is to export a group to a new empty file and see what levels the items end up on. If there are problems, fix then in this new file, delete the excess levels, then replace the group back in your project.
- Each group has an origin and Revit can sometimes get confused about what Level it is on.

You can find more information in Revit Forum - https://www.revitforum.org/showthread.php/6859-Best-Practices-for-Groups-in-High-rise-Tower-Models?highlight=groups+tower

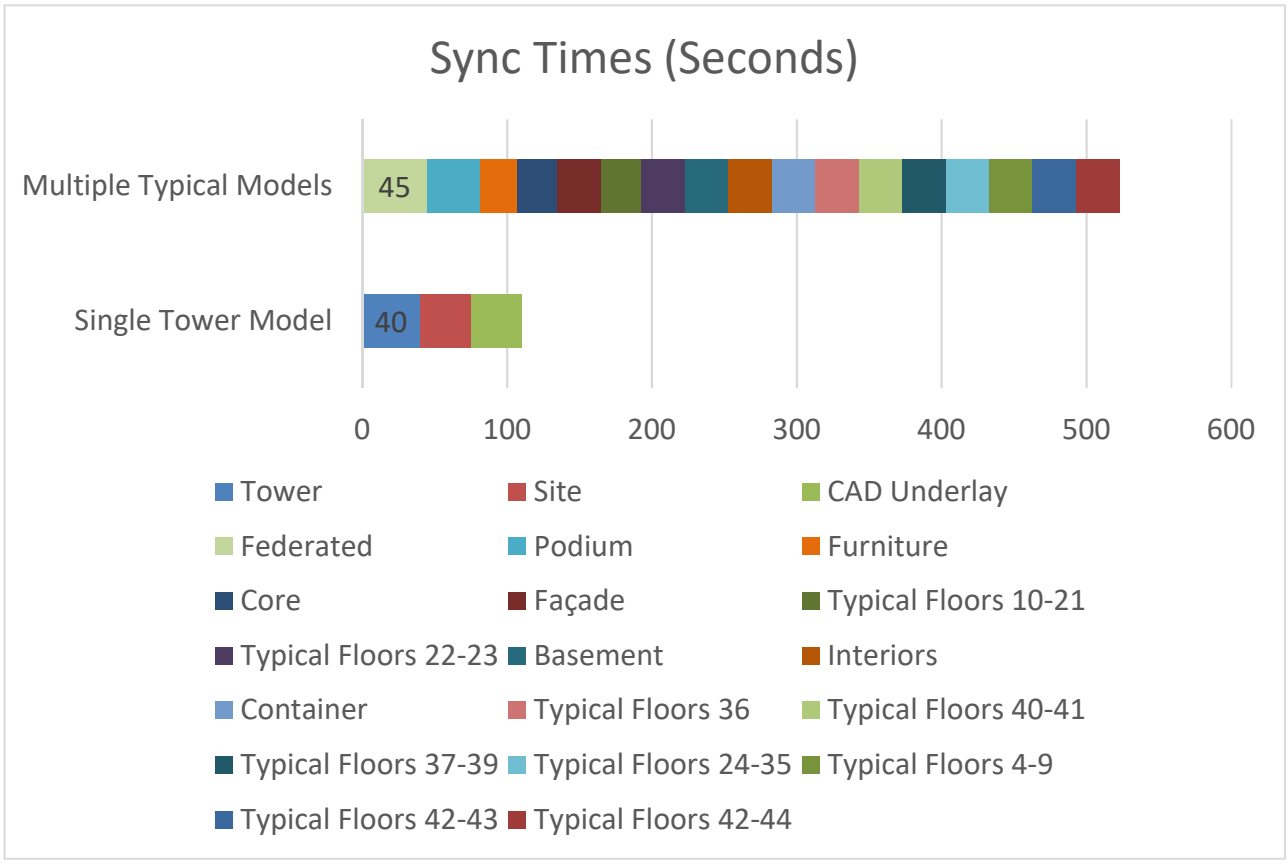


Figure 3 Sync time comparison between multiple models and a single model

When you compare link vs group workflow it's important to consider synchronization times.
When using the Revit link workflow, each of the individual files will sync faster than one large file of the group workflow, but you'll need to open federated file as well as the individual file, then reload and synchronize again.
When using the group workflow, you'll have one large file that will synchronize slightly slower than individual floor file, however 5 seconds faster than the federated file, and you won't have to open multiple files and reload.

Using Dynamo to create model elements

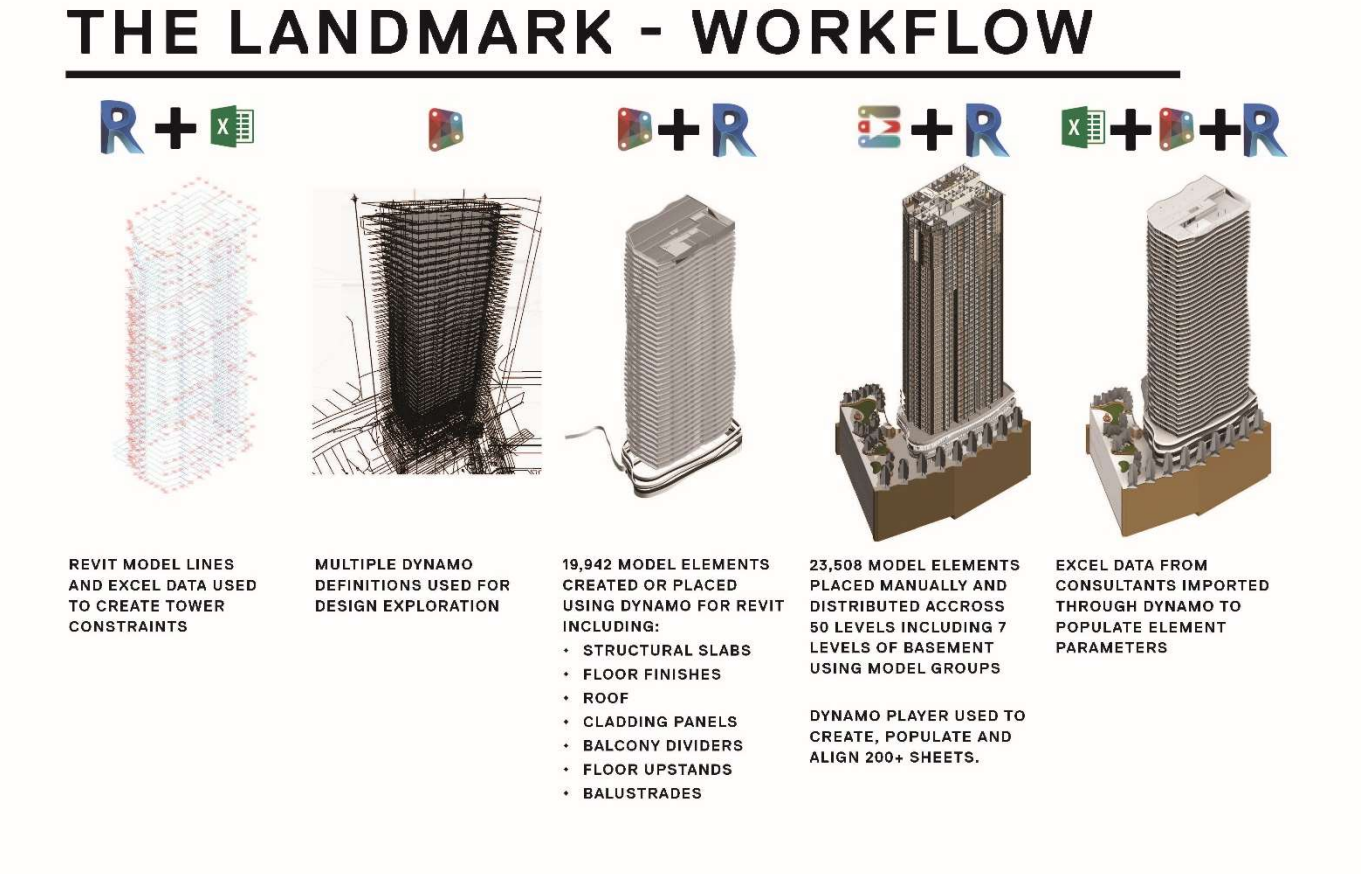


Figure 4 Workflow overview

1. Floors and other elements created using Dynamo

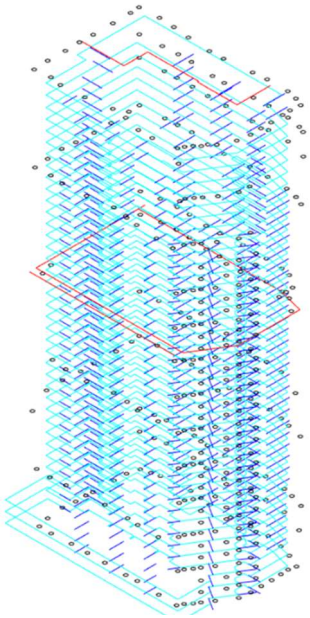


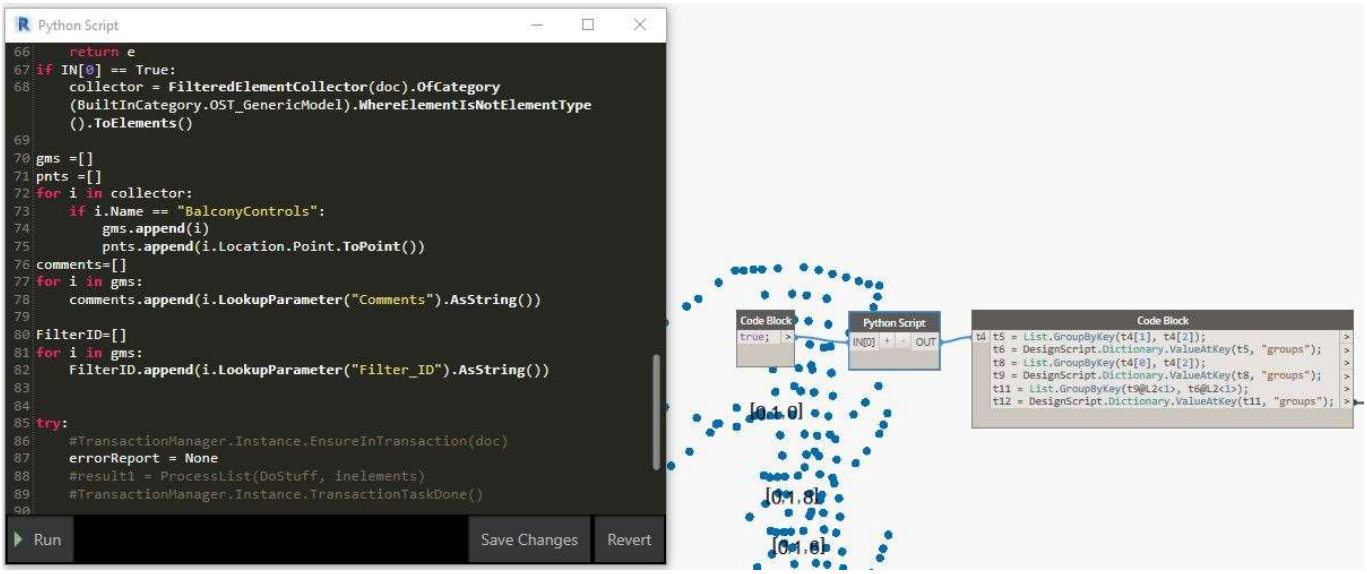
Figure 5 Generic Model elements as control points and line constraints

The process started with creating a set of constrains using model lines and circles to define façade outline, balcony separation and key points defining overall mass.

To help with reducing errors when lines were added or removed, each constrain had been created using a separate line style

Initially points defining the balcony constraints have been selected using the *SELECT MODEL ELEMENTS* node, to define groups of points used in *NURBS.BYPOINTS* node. However, this was later replaced by Generic Model families and python script with Filtered Element Collector.

Filtered Element collector allows you to select specific elements based on selected criteria. In this instance the collector selects all elements of *BUILTINCATEGORY.OST_GENERICMODEL*
Then I've used a for loop to iterate through the collector and add location points of elements where name equals to "BalconyControls" to a list. Additional for loops are getting the Comments and Filter_ID parameters that are used later for grouping points together.



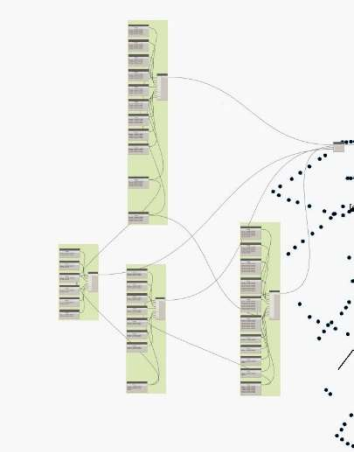


Figure 6 – Same selection required a lot of Select Model Element nodes

This workflow is more robust than selecting elements using the Select Model Elements node. If new elements are added they are automatically picked by the script. Also, if the script is opened and saved in a different active model, the selection is not affected, like it would be in the 'traditional' method.

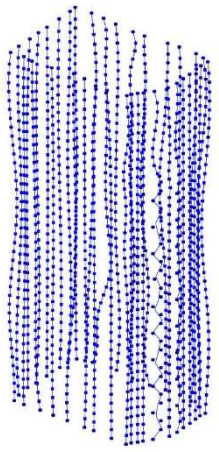


Figure 7 - Nurbs Curves Intersected with Level Planes

The Nurbs curves have been then used to intersect with planes defined by each level.
This created a list of points on each level defining the curves one each side of the building. The easiest way of crating smooth curves would be to connect all of these points using Nurbs curves however, they would be difficult to document and would have a limited functionality down the track, when placing Revit elements.
Instead we've created a custom definition, that was later on converted into a python script and eventually a zero touch node, to improve performance.

This definition is taking the first three points to define an initial arc

and connect other points using tangent arcs.

All of these curves are then combined into a polycurve for each side of the building and a combined polycurve for the complete outline.


```python
import clr
clr.AddReference('ProtoGeometry')
from Autodesk.DesignScript.Geometry import *
#The inputs to this node will be stored as a list in the IN variables.
dataEnteringNode = IN
import sys
pyt_path = r'C:\Program Files (x86)\IronPython 2.7\Lib'
sys.path.append(pyt_path)
pnts = IN[0]


try:
    errorReport = None
    crvs = []
    for e in pnts:

        arcs = []
        a1=Arc.ByThreePoints(e[0],e[1],e[2])
        arcs.append(a1)


        for ind, i in enumerate(e[3:]):
            try:
                afnd = Arc.ByStartPointEndPointStartTangent(arcs[ind].EndPoint,i,arcs
                [ind].TangentAtParameter(1))
                arcs.append(afnd)
            except:
                afnd = Line.ByStartPointEndPoint(arcs[ind].EndPoint,i)
                arcs.append(afnd)

        crvs.append(PolyCurve.ByJoinedCurves(arcs))
except:
    # if error accurs anywhere in the process catch it
    import traceback
    errorReport = traceback.format_exc()
#Assign your output to the OUT variable
if errorReport == None:
    OUT = crvs
else:
    OUT = errorReport
```

Figure 8 - Custom Python script to create smooth PolyCurves from list of points

```
namespace Geometry
{
    public class Polycurve
    {
        private Polycurve()
            //used to remove addtional menu level in when loaded into dynamo
        {
        }
        public static Autodesk.DesignScript.Geometry.PolyCurve TangentContinuousByPoints(List<Autodesk.DesignScript.Geometry.Point> points)
        {
            int i = 0;
            //starting i value
            List<Autodesk.DesignScript.Geometry.Curve> curves = new List<Autodesk.DesignScript.Geometry.Curve>();
            //empty list of curves
            foreach (Autodesk.DesignScript.Geometry.Point pnt in points)
            {
                if (i == 0)
                    //starting curve
                {
                    try
                    {
                        //create a curve from first three points
                        Autodesk.DesignScript.Geometry.Arc curv = Autodesk.DesignScript.Geometry.Arc.ByThreePoints(points[i], points[i + 1], points[i + 2]);
                        //add curve to list
                        curves.Add(curv);
                        //increment i to move to next step
                        i = 2;
                    }
                    catch
                    {
                        //if can't create curve, create a line
                        Autodesk.DesignScript.Geometry.Line curv = Autodesk.DesignScript.Geometry.Line.ByStartPointEndPoint(points[i], points[i + 2]);
                        //add curve to list
                        curves.Add(curv);
                        //increment i to move to next step
                        i = 2;
                    }
                }
                else if (i <= points.Count - 2)
                {
                    try
                    {
                        //create a curve from end tangent and endpoint of previous and next point
                        Autodesk.DesignScript.Geometry.Arc curv = Autodesk.DesignScript.Geometry.Arc.ByStartPointEndPointStartTangent(points[i], points[i + 1], curves[i - 2].TangentAtParameter(1));
                        //add curve to list
                        curves.Add(curv);
                        //increment i
                        i++;
                    }
                    catch
                    {
                        //if can't create curve, create a line
                        Autodesk.DesignScript.Geometry.Line curv = Autodesk.DesignScript.Geometry.Line.ByStartPointEndPoint(points[i], points[i + 1]);
                        //add curve to list
                        curves.Add(curv);
                        //increment i
                        i++;
                    }
                }
            }
        }
    }
}
```

Figure 9 - Smooth PolyCurve created using a zero-touch node

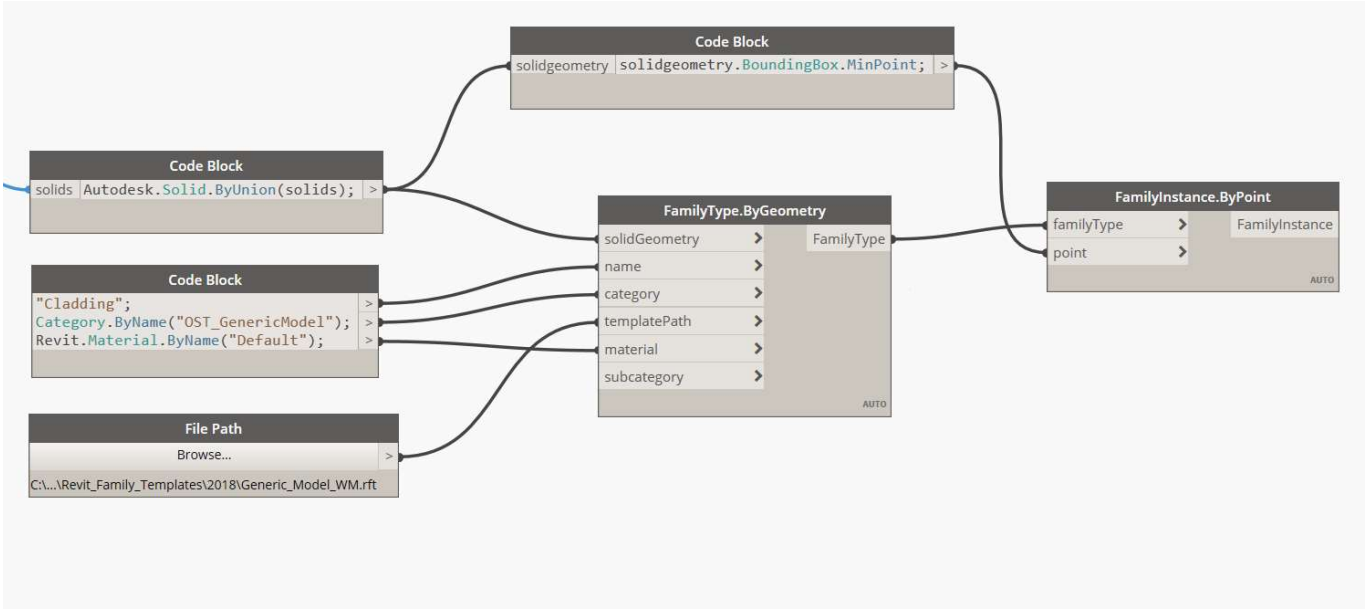
This outline is being used to create a concrete floor on each level as well as a concreate hob on the parameter of the floor using the *WALL.BYCURVEANDHEIGHT* node.
And at the early stages of the design a curtain wall representing the balustrade.
The final balustrade element is being created within Dynamo to allow for a detailed setout of each panel and fixing elements.

To create the individual floors for floor finishes on balconies, I've used the outline of the main building defined by a model lines drawn on each typical level. This was then subtracted from the outline of the building alongside with the placement lines for balcony separation walls.
These sets of outlines have been used to create individual floors using *FLOOR.BYOUTLINETYPEANDLEVEL*

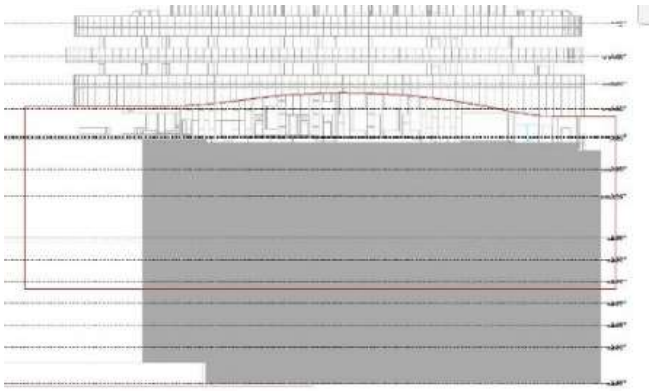
As each of the balcony separation walls have different lengths, so we've created them longer than the maximum extent of the outline and intersected them with the outline of the floor. This allowed us to get the individual lengths and apply this as a length parameter to the balcony separation family.

Part of the façade with direct shape families

Part of the façade is a form curved in two directions. This would be difficult to create using OOTB Revit elements. To create the form I've used Dynamo to create overall form and individual panels. Panels were created initially as individual families using the *SPRINGS.FAMILYINSTANCE.BYGEOMETRY node. Later on it became apparent that the combination of the two OOTB nodes, the FAMILYTYPE.BYGEOMETRY and FAMILYINSTANCE.BYPOINT work better than the Springs node.*



It's because you can check if geometry of any of the elements is being repeated and only create a few types. Then just place the instances. The placement point of the newly created families is the defined by the *MINPOINT* of the *BOUNDINGBOX* of the family geometry.



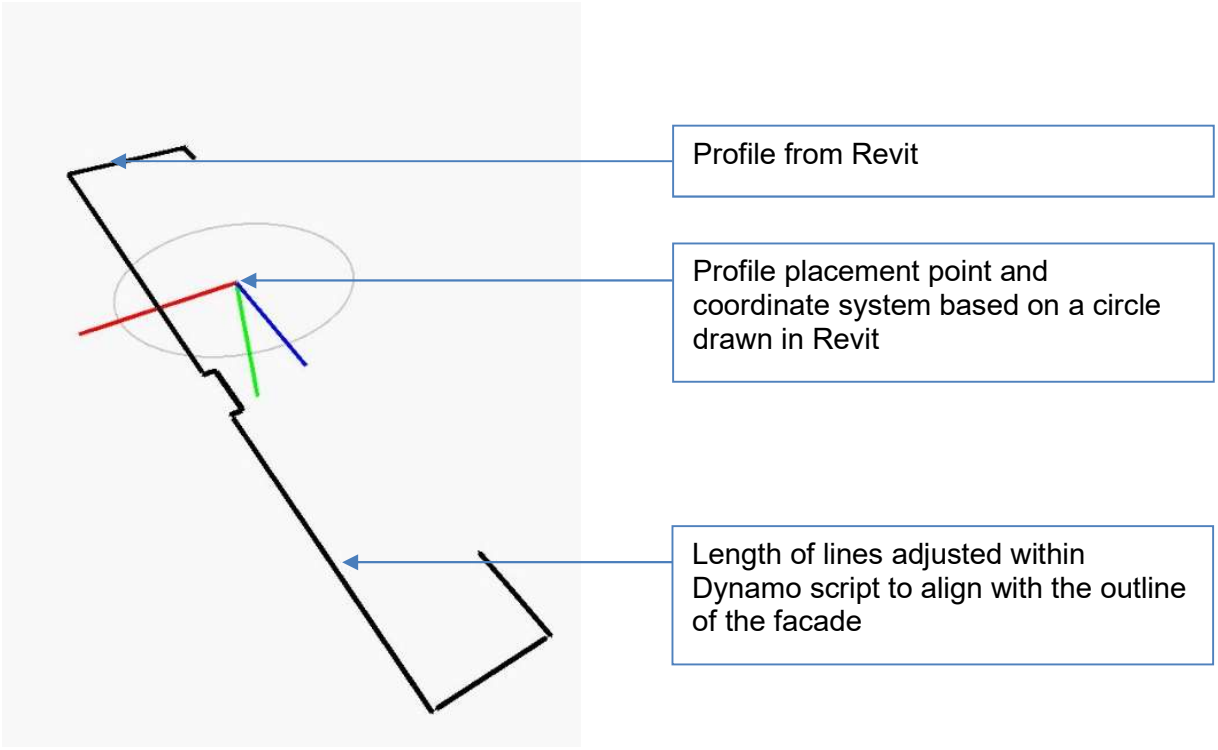
The outline was based on the sketch curves of the selected floors. To define vertical curvature, detail curves placed on selected sections were used as a base to create solids forms that were cut out of the overall form.

Figure 10 Facade outline indicated in a section view

This form then was used to get heights of the panels. The profiles of the panels were created using detail lines in a drafting view.
This allowed for easy modification and input from members of the team that were not using Dynamo.

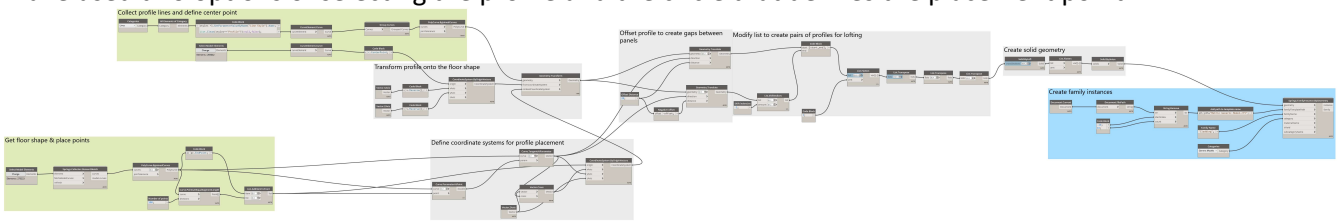
In the initial phases of the project the façade panels have been created as one object. And later split into individual items for tagging and scheduling.

Lower parts of the façade are curving not only following the curvature of the floor, but also vertically. To accommodate this, the variable parts of the profile had to be recreated

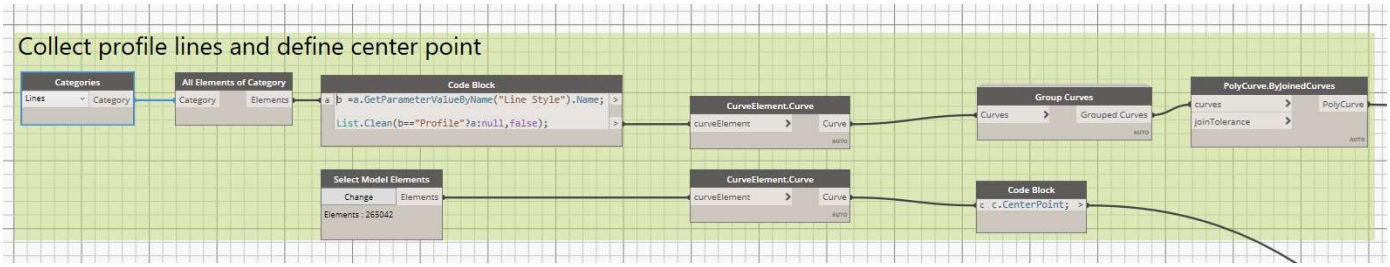


First, we need to define a profile for the façade and a placement point. You can draw the profile in any view, using model or detail lines.

I have used two options of selecting the profile and the circle that defines the placement point.



The circle is selected using the *SELECT MODEL ELEMENTS* node, followed by *CURVEELEMENT.CURVE* to extract the geometry from Revit element. Then I've used a code block to get the *CENTERPOINT* from the circle.



For getting profile curves we can use the same method, however in this example I've collected all lines from the project and filtered a specific line style. This could be helpful if you are constantly modifying your profile and don't want to keep selecting new lines every time.

To get the lines we can use All Elements of Category and select Lines as the category.
Then a code block

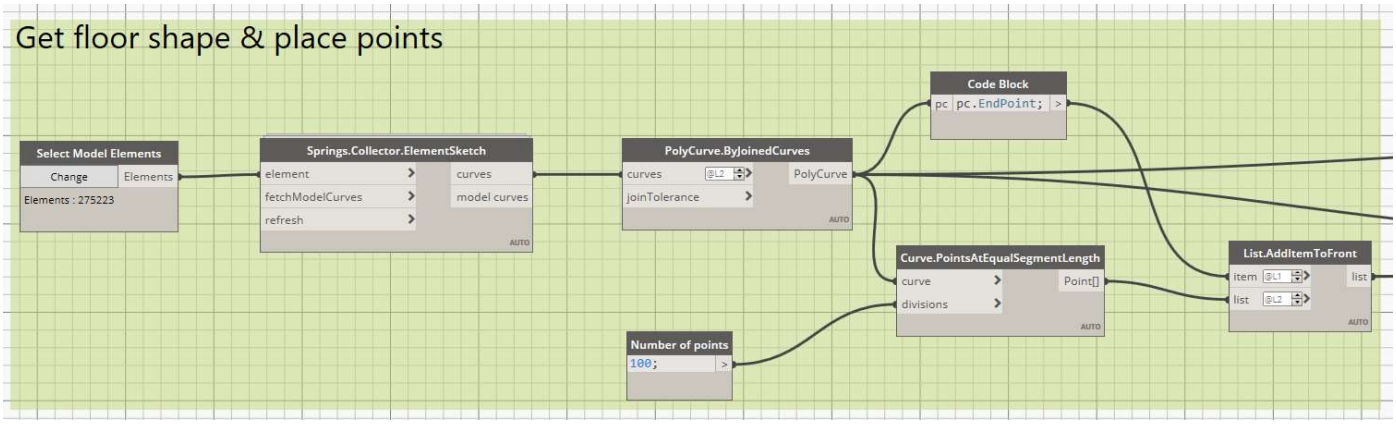
STYLES =L.GETPARAMETERVALUEBYNAME("LINE STYLE").NAME;

We are defining styles as a variable that equals to the name of a line style for each line

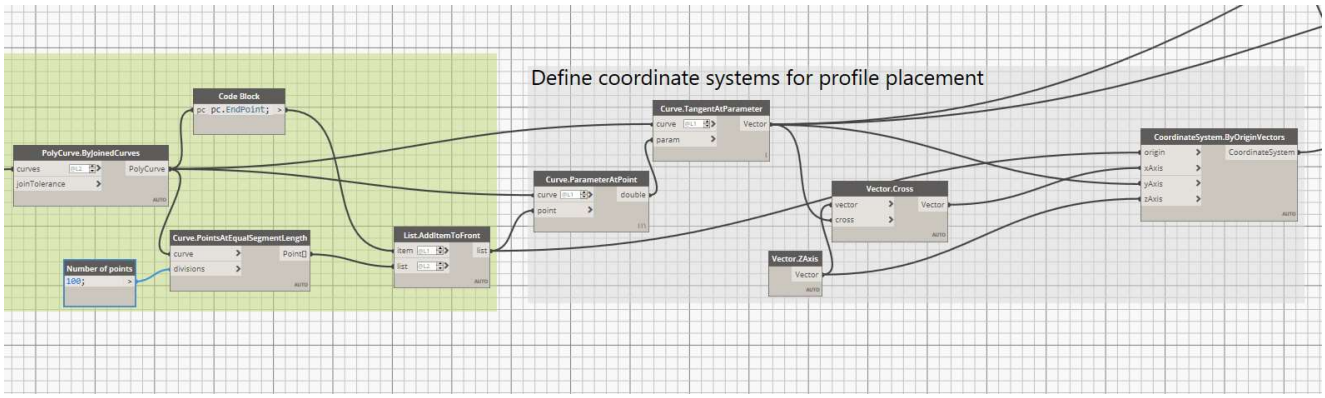
LIST.CLEAN(STYLES=="PROFILE"?L:NULL,FALSE);

Then we can filter our list using the if statement notation and *LIST.CLEAN* function.
If the style equals to "Profile" we're getting the line otherwise is a null value. Then null values are removed using the *LIST.CLEAN* with false flag for the *PRESERVEINDICES*.

Now we can extract the curve geometry using *CURVEELEMENT.CURVE* and Group Curves from archilab packages to group connected curves, so we can use Polycuve.ByJoinedCurves to create polycurves.

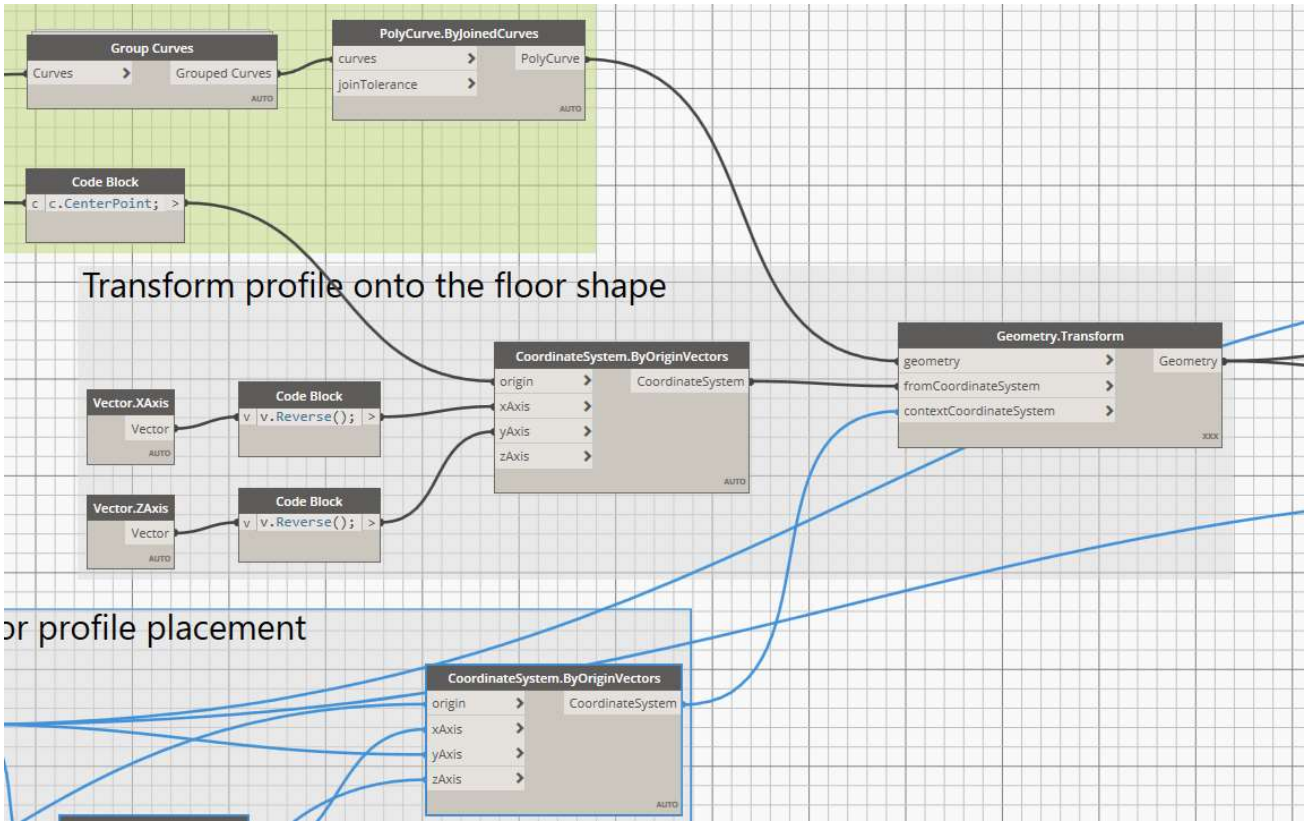


Next step is to get the outline of the floor. We can use the Select Model Elements to select the floor and *SPRINGS.COLLECTOR.ELEMENTSKETCH* node from Springs package to get sketch curves.
To define where the profiles will be placed we can use the *CURVE.POINTSATEQUALSEGMENTLENGTH*. However, we will need to add the endpoint of the Polycurve to the list using *LIST.ADDITEMTOFRONT* node, to get an equal spread of points.

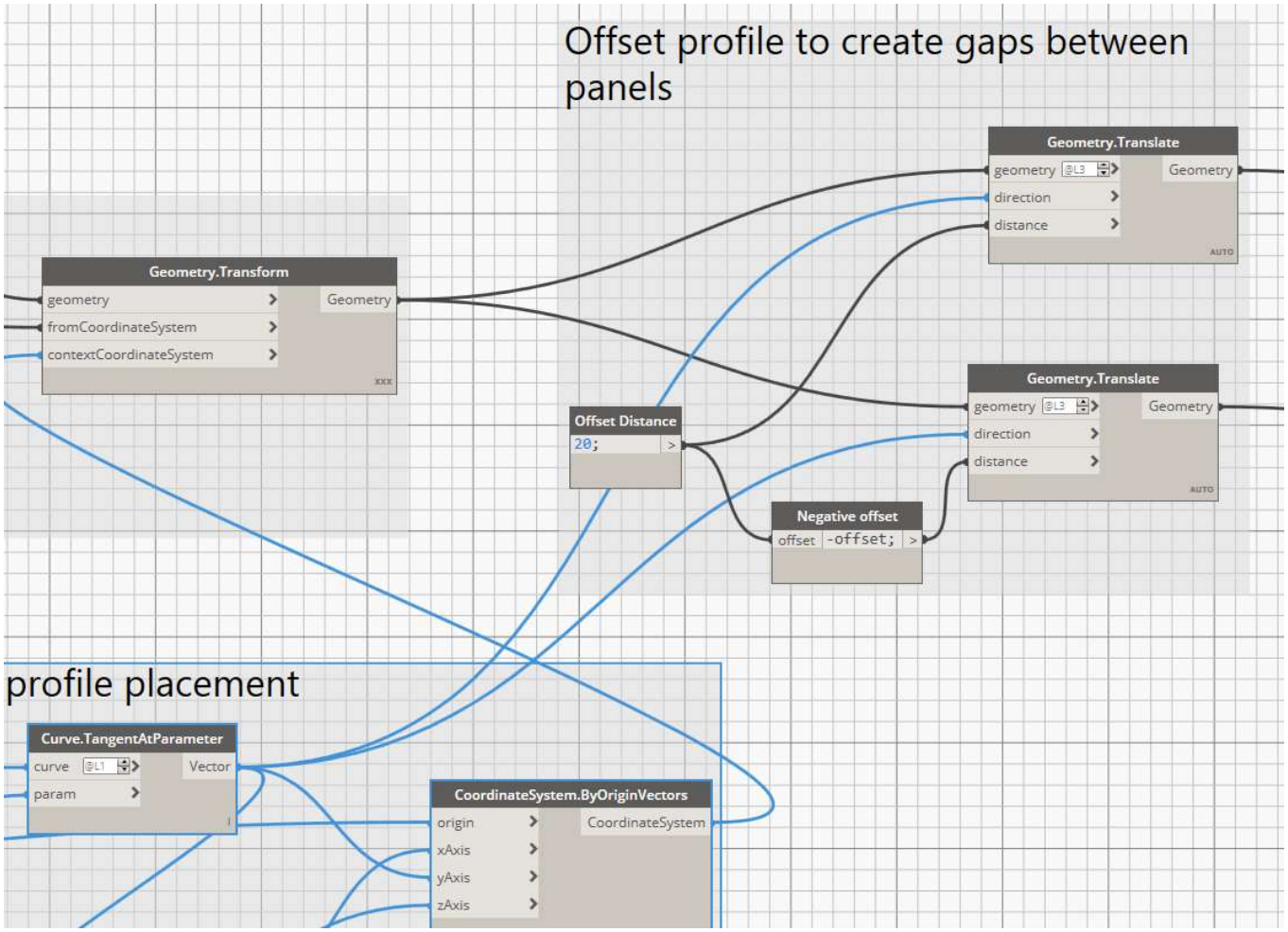


Now we can use the points to create Coordinate systems that will be used to correctly place cladding profiles.

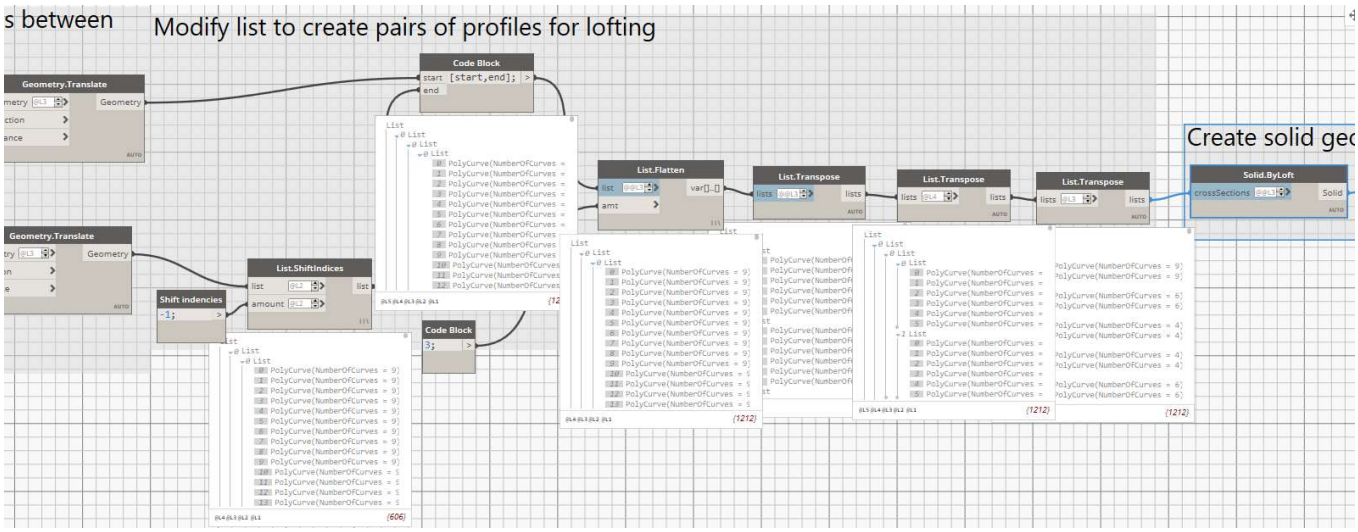
We'll start with converting points to parameters, that will be used later. We'll use *Curve.ParameterAtPoint* to do that. Then we can use the *Curve.TangentAtParmater* to get tangents at each point. We'll use them as the Y axis of the new coordinate systems. We'll use the *Vector.ZAxis()* as the Z axis. Finally, we can use the *Vector.Cross* function to get the X axis.
Now, we can plug in the points and the vectors to the *CoordinateSystem.ByOriginVectors* node to create the coordinate systems



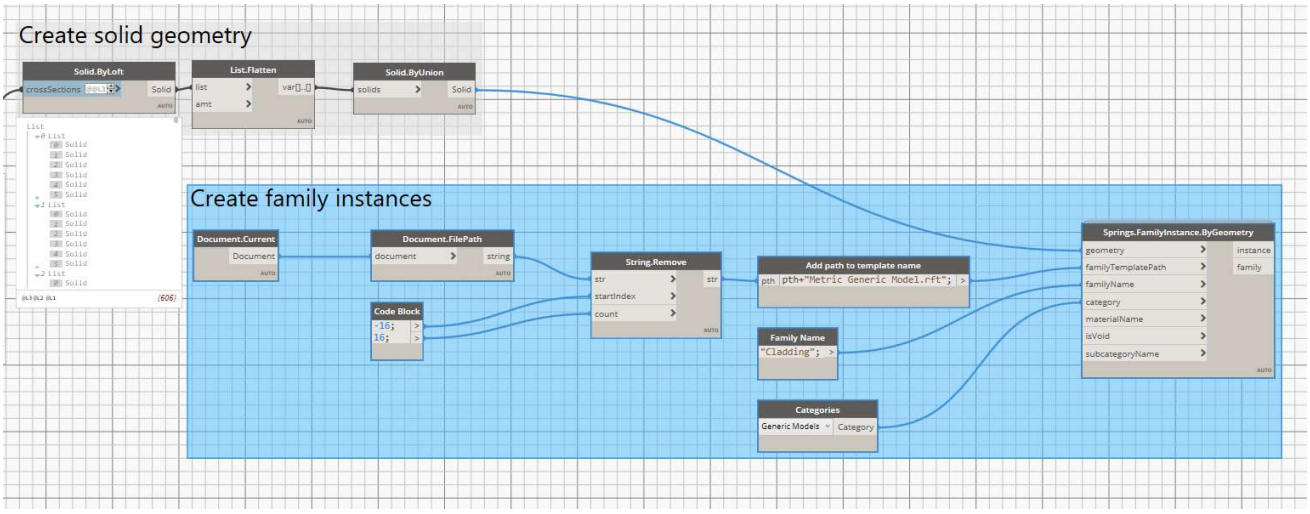
To position profiles on the coordinate systems, we'll use *Geometry.Transform* node. Connect profile curves to the geometry and coordinate systems to *contextCoordinateSystem* input. For the *fromCoordinateSystem*, we'll create *CoordinateSystem.ByOriginVectors* using the circle center point as origin and reversed *Vector.XAxis()* and *Vector.ZAxis()* to orient the profile.



Now we can use *GEOMETRY.TRANSLATE* to create gaps between panels. Use the *CURVE.TANGENTATPARAMETER* as the direction. And positive and negative value offset as the distance, to equally offset the profile in both directions.



We'll need to use *LIST.SHIFTINDICIES* node to create panels, otherwise the profiles would connect between gaps and not the panel. Then create a list using two profile lists. Finally, flatten the list and transpose it three times using different level settings to get lists containing pairs of profiles, which then is used in the *SOLID.BYLOFT* node.



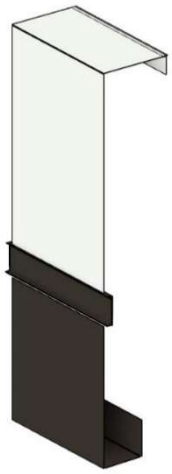


Figure 11 Individual panel

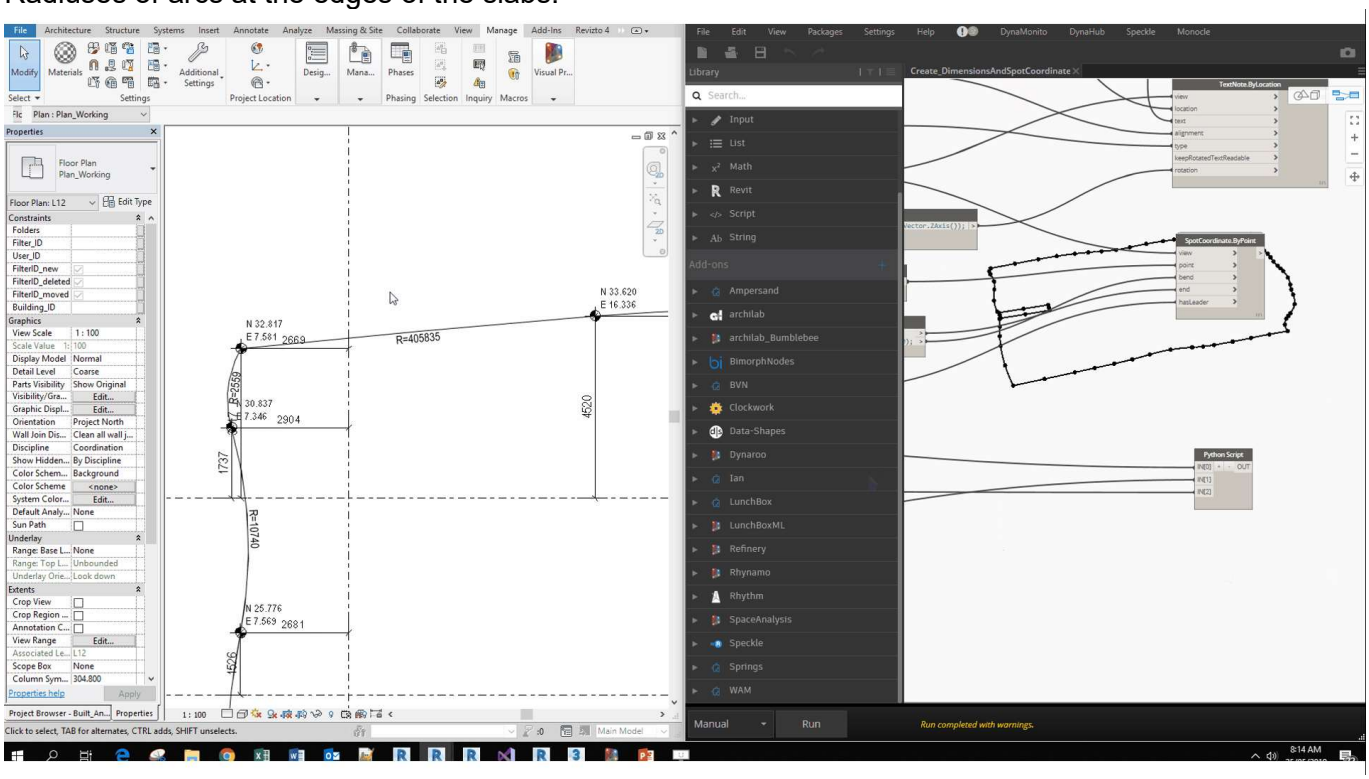
The newly created list of solids needs to be flattened, so we can use the *SOLIDS.BYUNION* node (unless you want to create panels as individual families)

To create the families we can use the *SPRINGS.FAMILYINSTANCE.BYGEOMETRY*. You can input the file path simply as a string or using the *FILEPATH* node.

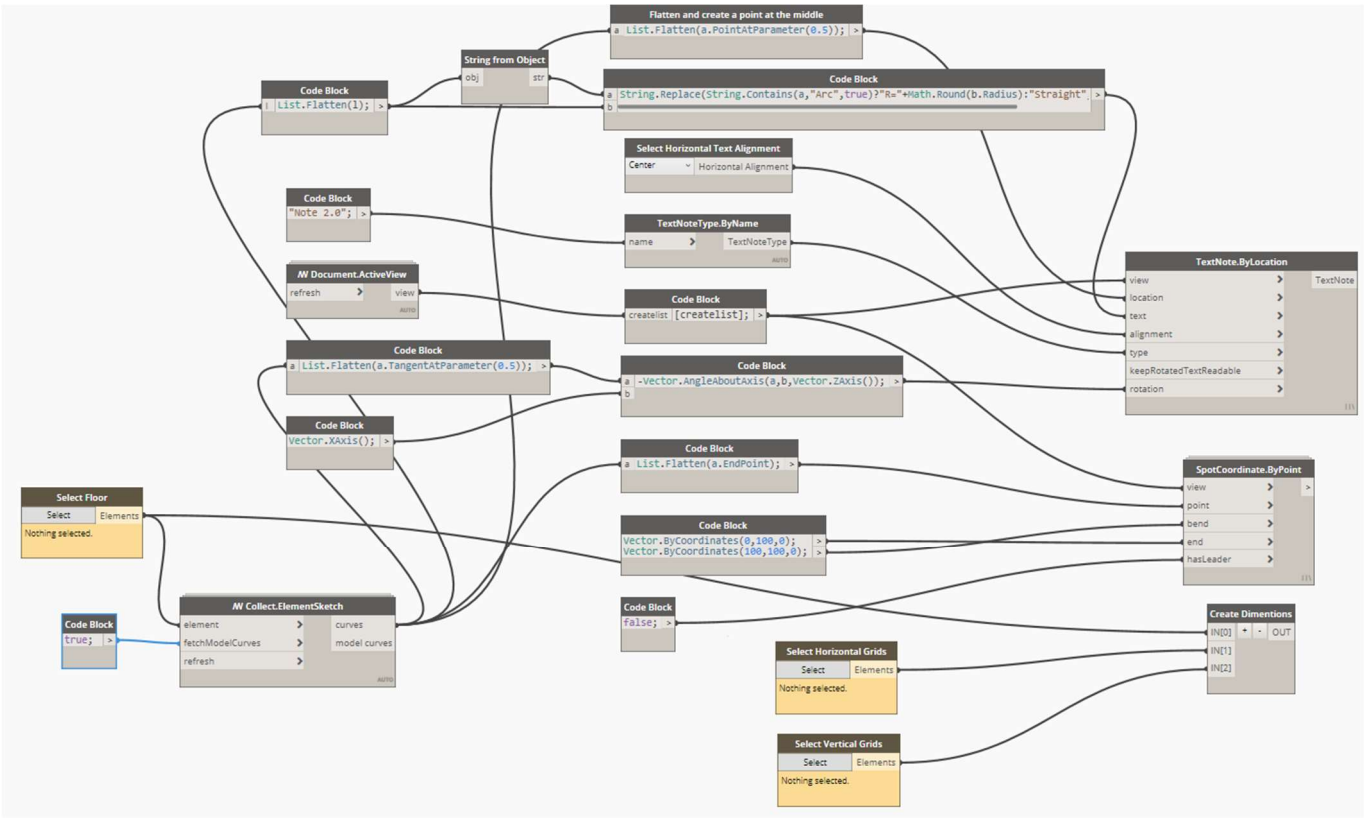
Automated dimensioning of curved slabs

To accommodate future changes in the overall form of the building and make the workflow of documenting all slabs with curved edges, I've had to come up with a more efficient solution.

Unfortunately, Revit API does have it's limitations for creating dimensions automatically. You can only create linear dimensions. To accommodate this, the Dynamo definition creates text notes to describe the Radiuses of arcs at the edges of the slabs.



Additionally, the definition, creates spot coordinates at the endpoints of all edges.



I’ve created a Python script to create dimensions. The script gets the floor geometry and creates dimensions between the endpoint of a floor edge and the nearest grid.

The first part of the script is used to import all required modules. For most of your scripts you can use pretty much the same modules. It may not elegant coding, but it's a good idea to add your most often modules to your Python template in Dynamo.

For interacting with Revit you need to import RevitNodes, RevitAPI and RevitAPIUI. If you need to create any Revit elements, you have to create them inside a transaction. This Requires importing the DocumentManager and TransactionManager from RevitServices.

```
4
5  import clr
6  # Import Element wrapper extension methods
7  clr.AddReference("RevitNodes")
8
9  # Import DesignScript Geometry - uncomment below if necessary
10 clr.AddReference("ProtoGeometry")
11 from Autodesk.DesignScript.Geometry import *
12
13 clr.AddReference("System")
14 from System.Collections.Generic import List
15
16 # Import Revit elements module
17 import Revit
18 clr.ImportExtensions(Revit.Elements)
19
20 #Import Conversion libraries between Revit and DesignScript - uncomment below if necessary
21 clr.ImportExtensions(Revit.GeometryConversion)
22
23 # Import DocumentManager and TransactionManager
24 clr.AddReference("RevitServices")
25 import RevitServices
26 from RevitServices.Persistence import DocumentManager
27 from RevitServices.Transactions import TransactionManager
28
29 #An object that represents an open Autodesk Revit project.
30 doc = DocumentManager.Instance.CurrentDBDocument
31
32 #Represents an active session of the Autodesk Revit user interface, providing access to UI customization methods, events, and the active document.
33 uiapp = DocumentManager.Instance.CurrentUIApplication
34 #The Application Creation object is used to create new instances of utility objects.
35 app = uiapp.Application
36 #An object that represents an Autodesk Revit project opened in the Revit user interface.
37 uidoc = DocumentManager.Instance.CurrentUIApplication.ActiveUIDocument
38
39 # Import RevitAPI
40 clr.AddReference("RevitAPI")
41 clr.AddReference('RevitAPIUI')
42 import Autodesk
43 from Autodesk.Revit.DB import *
44 from Autodesk.Revit.UI import *
45
46 #Import sys module
47 import sys
48 #Specify location path for IronPython install
49 pyt_path = r'C:\Program Files (x86)\IronPython 2.7\Lib'
50 sys.path.append(pyt_path)
51
52 getDocUnits = doc.GetUnits()
53 getDisplayUnits = getDocUnits.GetFormatOptions(UnitType.UT_Length).DisplayUnits
54 unitConversion = UnitUtils.ConvertFromInternalUnits(1, getDisplayUnits )
55
56 #The inputs to this node will be stored as a list in the IN variable.
```

Next the script defines the inputs for the floor and horizontal and vertical grids.

- Create options required for creating dimensions
- Get solid of floor element
- Get edges of the solid
- Discard edges underside of slab
- Get Edge start points
- Calculate Distance between curve start point and each grid

```
dataEnteringNode = IN

def ProcessList(_func, _list):
    return map( lambda x: ProcessList(_func, x) if type(x)==list else _func(x), _list )

def UnwrapNestedList(e):
    return UnwrapElement(e)

if isinstance(IN[0], list):
    inelements = ProcessList(UnwrapNestedList, IN[0])
else:
    inelements = [UnwrapElement(IN[0])]
if isinstance(IN[1], list):
    hgridcurves = ProcessList(UnwrapNestedList, IN[1])
else:
    hgridcurves = [UnwrapElement(IN[1])]

if isinstance(IN[2], list):
    vgridcurves = ProcessList(UnwrapNestedList, IN[2])
else:
    vgridcurves = [UnwrapElement(IN[2])]

try:

    errorReport = None
    TransactionManager.Instance.EnsureInTransaction(doc)
    dimensions=[]
    opt = Options()
    opt.ComputeReferences=True
    opt.IncludeNonVisibleObjects=True
    opt.View = doc.ActiveView
    for element in inelements:
        geometry = element.get_Geometry(opt)
        for solid in geometry:
            edges= solid.Edges
            bbx= element.get_BoundingBox(doc.ActiveView)
            mid = Line.CreateBound(bbx.Min,bbx.Max).Evaluate(0.5,True)
            #Count number of edges and get half
            n=(edges.Size-1)/2
            for i, edge in enumerate(edges):
                #Check if edges are flat and contained in second half of the enumerator
                if edge.Evaluate(1).Z == edge.Evaluate(0).Z and i>=n:
                    #Create start point
                    pnt =edge.Evaluate(0)
                    #create lists for distance measurement for vertical and horizontal grids
                    disth=[]
                    distv=[]
                    #create list of distances between curve start point and grid horizontal
                    for h in hgridcurves:
                        disth.append(h.Curve.Distance(pnt))
                    #get first item of the above list after sorting the values
                    minh = sorted(disth)[0]
                    #create list of distances between curve start point and grid horizontal
                    for v in vgridcurves:
                        distv.append(v.Curve.Distance(pnt))
                    #get first item of the above list after sorting the values
                    minv = sorted(distv)[0]
                    #create horizontal lines from point for dim placement
                    if pnt.X >= mid.X:
                        dirh = 4
                    else:
                        dirh = -4
```

- Create reference for grid at the minimal distance to point
- Add items to Reference Array
- Create dimension

```
                    if pnt.Y >= mid.Y:
                        dirv = 4
                    else:
                        dirv = -4
                    lh=Line.CreateUnbound(XYZ(pnt.X+dirh,pnt.Y,pnt.Z),XYZ(0,1,0))
                    for h in hgridcurves:
                        #check if the distance to grid is equal to minimum
                        if h.Curve.Distance(pnt) == minh:
                            refgh=h
                            break
                    lv=Line.CreateUnbound(XYZ(pnt.X,pnt.Y+dirv,pnt.Z),XYZ(1,0,0))
                    for v in vgridcurves:
                        #check if the distance to grid is equal to minimum
                        if v.Curve.Distance(pnt) == minv:
                            refgv=v
                            break
                    #Create reference arrays
                    elementsRefh=ReferenceArray()
                    elementsRefv=ReferenceArray()
                    #ger reference to curve start point
                    pntref = edge.GetEndPointReference(0)
                    #add references to arrays
                    elementsRefh.Append(pntref)
                    elementsRefh.Append(Reference(refgh))
                    elementsRefv.Append(pntref)
                    elementsRefv.Append(Reference(refgv))
                    #create dimensions
                    dimh= doc.Create.NewDimension(doc.ActiveView,lh,elementsRefh)
                    dimv= doc.Create.NewDimension(doc.ActiveView,lv,elementsRefv)
                    dimensions.append([dimh,dimv,dirh,dirv])

    TransactionManager.Instance.TransactionTaskDone()

except:
    # if error accurs anywhere in the process catch it
    import traceback
    errorReport = traceback.format_exc()

#Assign your output to the OUT variable
if errorReport == None:
    OUT = dimensions,mid.ToPoint()
else:
    OUT = errorReport
```

Tips & Tricks

- **Tip 1: Careful planning of model split will save you time down the track**
- **Tip 2: Use selection by parameters in Dynamo to maintain consistency**
- **Tip 3: Groups work well if you follow best practice rules. Make sure that your team knows them all.**
- **Tip 4: Make your Dynamo workflows inclusive – Drive Dynamo geometry with Revit Geometry.**