

# Automate Everyday Revit Tasks with Dynamo

**Paul F. Aubin**—is the author of many Revit book titles including the Aubin Academy Series and his "deep dive" into the Revit family editor: Renaissance Revit. He also authors dozens of video titles at lynda.com (now LinkedIn Learning) covering Revit topics from beginner to advanced. Paul is an independent architectural consultant providing Revit content creation, implementation, training and support services. His career of nearly 30 years, includes experience in design, production, CAD management, coaching and training. Paul is an active member of the Autodesk user community and has been a top-rated speaker at conferences such as Autodesk University, BILT and Midwest University for many years; and he recently gave the keynote address to the Campus FM Technology Association annual conference. Paul is an associate member of the AIA, an Autodesk Expert Elite and an Autodesk Certified Professional. He lives in Chicago with his wife and their three children currently attending universities around the country.

Visit: [www.paulaubin.com](http://www.paulaubin.com)

Follow: @paulfaubin

## Class Description

Do you find yourself frequently having to perform the same task several times over the course of a project? Are there workflows that your firm relies on that are not easily achieved with native Revit tools and techniques? Do you ever find yourself thinking that your current procedures are just not maximizing the promise or potential of your BIM? It's times like these where a little Dynamo can be just the thing. In this session, we will walk through some complete workflows to automate common repetitive tasks and more importantly, give end users the confidence they need to know that the resulting data they see is correct and accurate. This session will explore using Dynamo to design a workflow that solves a simple repetitive task in Revit. We'll discuss the problem, walk through the design of the solution, and explore the Dynamo graph piece by piece. Don't worry if you are new to Dynamo or programming, we'll keep it simple, approachable, understandable and free from complex code.

## Learning Objectives

- Learn how to identify a good task for Dynamo
- Learn how to frame the problem and break down the steps
- Understand how to approach building a graph even if you are not a programmer
- Learn to organize and document your graphs so they are easy to understand and modify

## Software Version

This session will use **Dynamo for Revit** and will use **Dynamo version 2.0.2** running in **Revit 2019.2** (see Figure 1). More information on Dynamo versions and installation in the appendix.

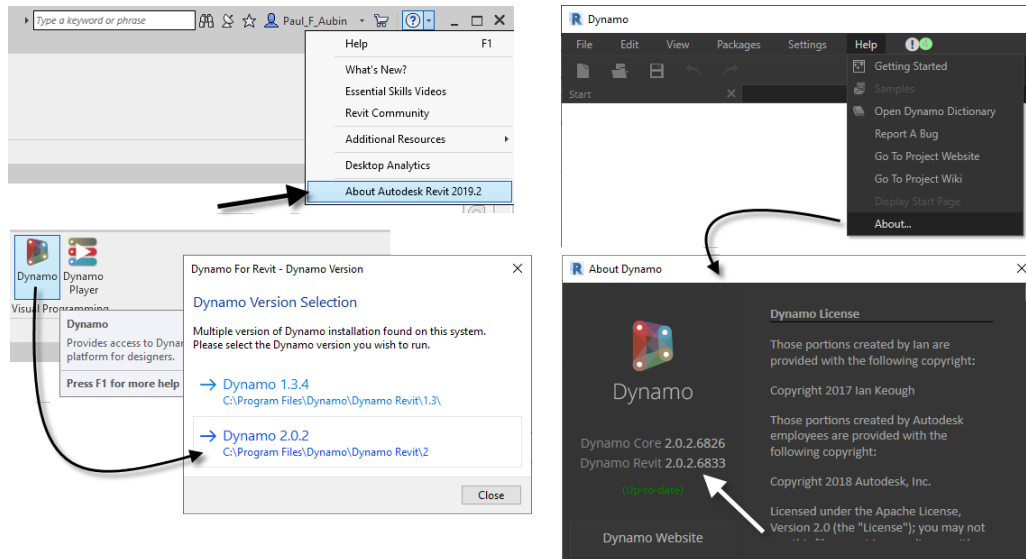


Figure 1 –Dynamo and Revit version used in this session

## Additional Resources

In this session, I am going to assume that this is not your first time using Dynamo. Therefore, the basics of launching Dynamo, creating a graph, adding nodes and connecting wires are all assumed. If you are not familiar with these tasks or are brand new to Dynamo, there are many online resources that can help you on journey to learn more about Dynamo. There is a list in the appendix that you can consult before and after the session. But we would like to share a few useful resources with you right away:

**LinkedIn Learning** (powered by lynda.com content) has a growing collection of courses that are well worth your time; including some by yours truly! (Dynamo: Practical) Check them out!

Visit: [www.linkedin.com/learning/](http://www.linkedin.com/learning/) to learn more and then search for: **Dynamo** (see Figure 2).

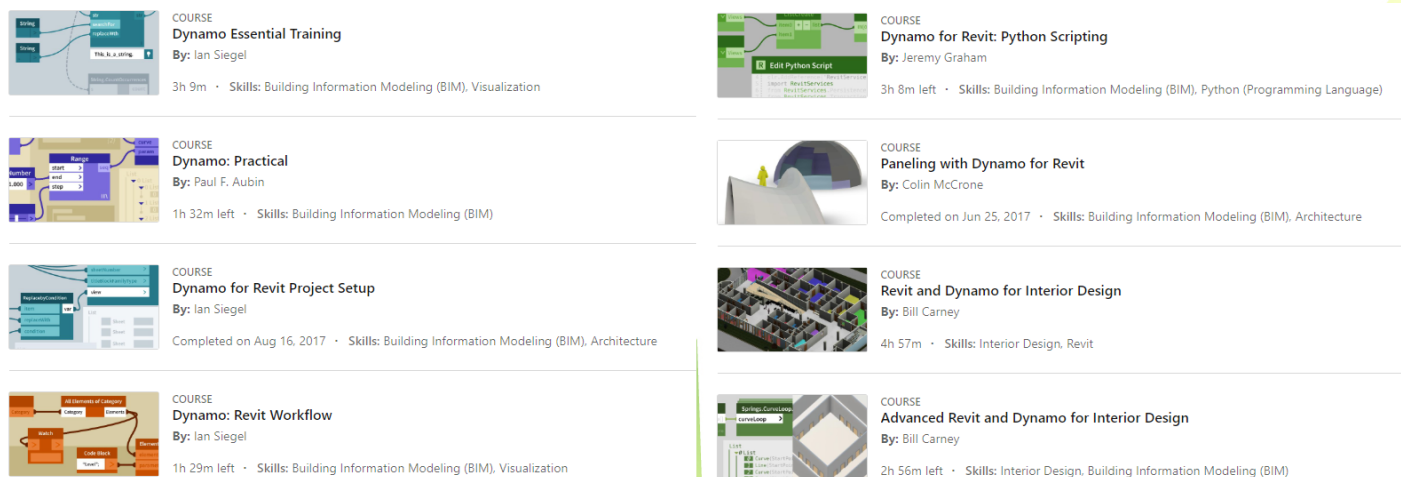


Figure 2 – Video courses on LinkedIn Learning (lynda.com)

Other popular resources:

[dynamobim.org/](http://dynamobim.org/) - the Dynamo home page.

<http://primer.dynamobim.org/en/> - an introductory tutorial on getting started with Dynamo.

[dictionary.dynamobim.com](http://dictionary.dynamobim.com) – this is a searchable database for Dynamo functionality.

[dynamonodes.com](http://dynamonodes.com) – all about dynamo nodes.

## Introduction

Got some mundane repetitive Revit task to do? Let Dynamo do it! This session explores the identifying of repetitive Revit tasks that are good candidates for Dynamo and explores a couple sample solutions in detail. Learn to create repeatable results with no complex coding required. In this session and handout, we will walk through two quick and practical graphs that you can build to solve real-world problems back at the office. These include:

- Calculating Room Occupancy Values based on building code guidelines
- Automatically determine and place Wall Type Details in use in a project

## Calculating Room Occupancy Values

In this example, we will use Dynamo to calculate the occupancy load values for several rooms in a project based on the code requirements for each type of space. For this example, we will be using values taken from the International Building Code, 2015 edition.

**IMPORTANT**—If you adapt this graph to your own use, please double-check that all values match the actual code requirements for the municipality having jurisdiction over your project.

For the purposes of this demo, the values in Table 1 will be used.

Table 1 - Occupancy Values per Room Type

Room Name	SF required per occupant	SM required per occupant	Room Name	SF required per occupant	SM required per occupant
Administration	100	9.2900	Lounge	15	1.3935
Advisors	100	9.2900	Media Review	50	4.6450
Cafeteria	15	1.3935	Men	100	9.2900
Computer Lab	50	4.6450	Office	100	9.2900
Conference	100	9.2900	Open Office	100	9.2900
Copy/Print	100	9.2900	Prep/Dish	200	18.5800
Corridor	100	9.2900	Sprinkler	300	27.8700
Drafting	100	9.2900	Stair	100	9.2900
Dry Storage	300	27.8700	Storage	300	27.8700
Electrical	50	4.6450	Toilet	100	9.2900
Instruction	20	1.8580	Vest.	100	9.2900
Library	50	4.6450	Women	100	9.2900
Lobby	5	0.4645			

The out-of-the-box Revit Architectural Advanced Sample Project will be used in the demonstration. The Room names in the table come from this project and have been compared to the National Building Code to come up with the values listed. Once again, please check the values against your own resources to ensure accuracy and compliance.

#### *Why Use Dynamo for this task?*

Calculating Occupancy Values is not difficult to do. You simply take the area of the room or space, divide it by the number of required square feet [or square meters] per occupant and round down to the nearest whole value. This is your maximum occupant load for that space.

So why not just do this calculation directly in Revit? You theoretically could do this, but it would require a good deal of manual input. At minimum, you would be required to input the values from Table 1 above somewhere in the model. The most likely place would be in a custom parameter assigned to the Room Category. However, there would be no easy way to input this data based on room function. So, you would be forced to manually input it for each room. And since room elements do not have types, you would be required to input the value for *every* room individually. Now clever use of schedule grouping and sorting could speed this process up if you group and sort by like functions. But consider just the simple example presented here, Table 1 includes 7 unique occupancy values used by 25 unique room names.

This means you would first need to create a custom project parameter to receive the occupancy value input, then create a schedule sorted by the 25 room names and finally make 25 separate edits to input those 7 occupancy values into the custom field. While making 25 edits is arguably not a tremendous amount of effort, it is certainly more than most of us wish to do manually; not to mention the potential for error when performing such repetitive input tasks. Furthermore, many projects (if not most) will have far more room types than the 25 used here.

Dynamo to the rescue! Dynamo is great for repetitive tasks like this one.

#### *Framing the problem*

So, what is involved in getting Dynamo to do this task for you? Well you must frame the problem in a way that Dynamo will understand and be able to process regardless of the input you feed in.

This means thinking about how data flows through a Dynamo graph and ensuring that you anticipate any issues or errors that are likely to occur along the way. The best thing you can do is keep it simple and stay focused on the main task. Here's what we want to do:

Have Dynamo gather all rooms in our project file, organize them by room type/usage and then assign the correct occupancy value to each one based on the table and calculation outlined above.

## Explore the Graph

The sample graph provided here is grouped into logical chunks. In this topic, we will go through each group and explore how data flows through the graph. This sample uses the room names from the Advanced Sample Project. If you adapt this for your own use, you will need to export your list of room names from your project first and then compile a table in Excel that has the desired occupancy values for each room name in use. This can be done using another Dynamo graph, or by simply exporting a Revit room schedule.

Groups in this graph are numbered. The explanations here will refer to them by number.

#### *Group 1*

Group 1 contains three nodes (See Figure 3).



## 1. Select Rooms

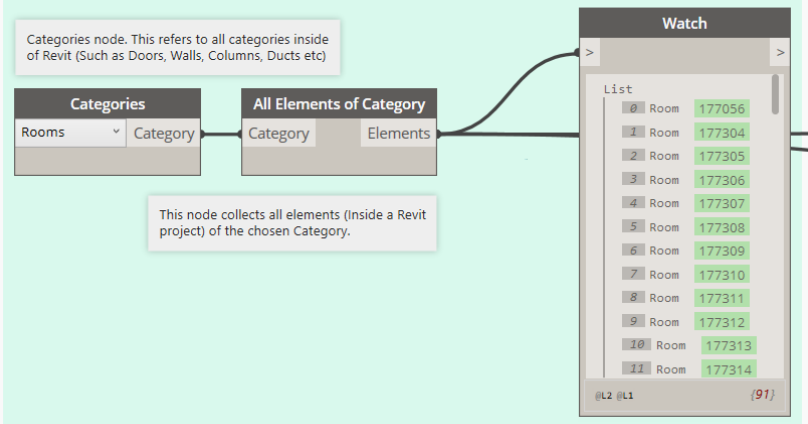


Figure 3 – Selecting all of the room elements in the model

This group makes the selection of Revit elements. The first node chooses the category of elements we want; Rooms in this case. And then the second node gathers all elements in the model that belong to this category. In other words, it selects all the room elements in the entire model. The **Watch** node is optional. But it can be nice to give feedback on the graph as you work through it. In this case, it is listing the selection and shows us that Dynamo has selected a total of 91 rooms. If you wanted to verify this, you could make a room schedule in Revit and it would yield the same quantity of rooms.

### Group 2

Group 2 also contains three nodes, but really there is one main one: **Element.GetParameterValueByName** (See Figure 4).

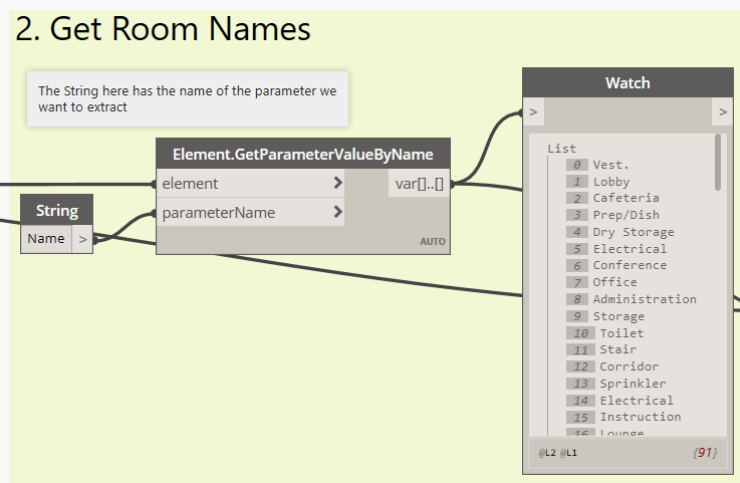


Figure 4 – Retrieve the room names from the selection of rooms

The list of room names from group 1 feeds into the **element** port of the **Element.GetParameterValueByName** node here in group 2. This is a very useful Revit node. It allows you to grab the value of any parameter on a selection of Revit elements. In this case, we are using a **String** node to feed in the name of the parameter we want into the **parameterName** input. A **String** node is a text node. The computer science word for text values are “strings.” The results here show up in another **Watch** node.

### Group 3

Group 3 contains five nodes (See Figure 5). There are two “ByKey” nodes. Keys are used to sort and group lists. In the first case, **List.SortByKey**, the node takes a list of values (91 total Revit Room elements from group 1) and sorts them based on the list of room names from group 2. This generates two outputs: a sorted list and a list of sorted keys.

These outputs are next fed into the **List.GroupByKey**. This node organizes the list into a “structured” list. A structured list is a list of lists. In other words, there is a main list that contains sub lists as its items. Here we end up with a sub list for each unique room name (the keys used for sorting and grouping) and then the values on these sub lists are the unique rooms corresponding to those name values. So, for example, the first list (0 List) contains three items. These are the three rooms named: Administration. You can see this structured list in the first **Watch** node.

### 3. Sort Rooms by name values, group by names and then extract list of unique names

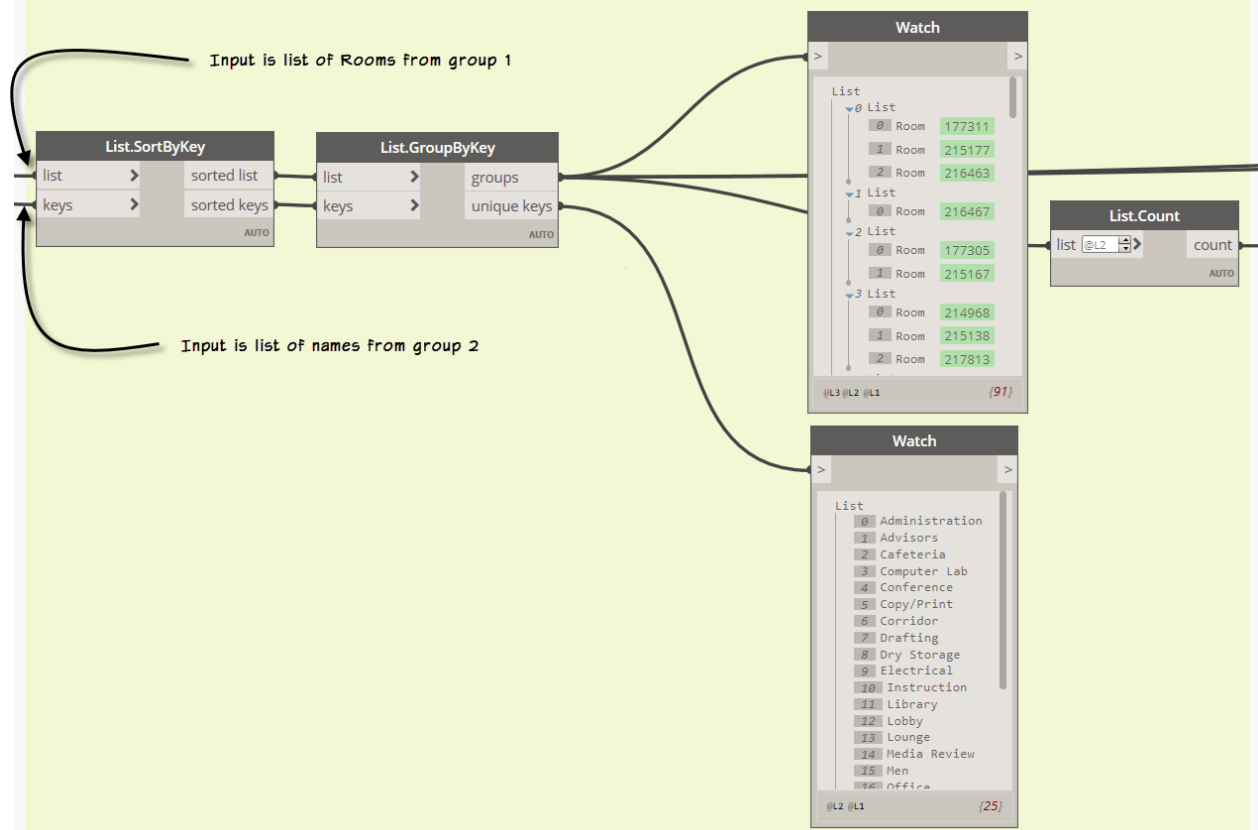


Figure 5 – Sort and group the list as a structured list using room name as the organizing key

The second Watch node gives us the output of the unique keys from the **GroupByKey** node. This shows the 25 unique room names used in the project (and noted above). The final node in this group gives us a count of each sub list. To achieve this, you must use “list at levels.” This is the small chevron icon next to the input. You click this, check the Use Levels checkbox and then set which level you want the node to apply at. Using **@L2** in this case gives us a list that counts each sub list instead of the main list and therefore gives us 25 counts (corresponding to the quantity of sub lists). If you applied the list at levels to either **@L1** or **@L3**, you would get very different results (see Figure 6).



Figure 6 – Use List at Levels to control at which list in the nested structure the node function should apply

To summarize where we are at in the first three groups, we started by selecting all the rooms in the model, querying their room names and then sorting and grouping the original collection of rooms by those room name values and then counting the total number of instances of each room name. Next, we want to match up our list of occupancy values with the rooms of each room type (based on names); onto group 4.

#### Group 4

Group 4 shows 7 nodes (See Figure 7). Part of being successful in Dynamo (and any programming endeavor) is knowing what to do in Dynamo (and what to do elsewhere). It is certainly possible to build the building code table directly in Dynamo. Doing so would require more nodes and make a more complex graph. Later editing of the values should the code change would require reworking the Dynamo graph. Instead, this group does something much simpler. It uses Excel. Excel is well known by most users and a great way to build a list of values. We can then simply have Dynamo read the data directly from the Excel file and use the data in Dynamo and Revit. Furthermore, since the data lives in Excel, it is easy to update when required.

### 4. Import occupancy multipliers from Excel

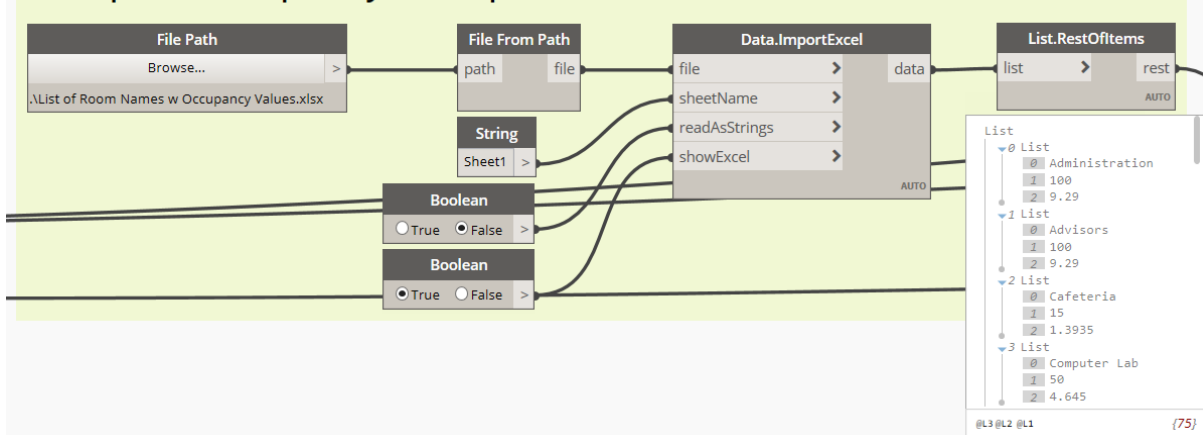


Figure 7 – Read an Excel file into your Dynamo graph

The **Data.ImportExcel** node takes four inputs. The first two are required, the last two are optional. The last two are Booleans (true or false). Attaching a **Boolean** node to these inputs allows you to change the default value if you choose. If you leave them as-is, or unconnected, they will use their respective defaults.

The first input: **file**, takes two nodes in series. Wire a **File Path** into a **File From Path** and then into the **file** input. If you skip the **File From Path** node, it will fail. So, make sure you have both. The Browse button on the **File Path** node lets you point to an Excel file you want to read. The final input: **sheetName** tells Dynamo which sheet in the Excel workbook you want to read. A simple **String** node allows you to type in whatever this name is. We end this group with a **List.RestOfItems** node. This node drops the first item on the list. This is useful here to eliminate the headers that are often included in the first row of an Excel file. If you expand the preview bubble on this node and look at the output, it will be a structured list. Each sub list was a row of data in Excel with the values on the list being the columns of data.

#### Group 5

Moving on to Group 5, the first node is **Transpose**. Often you want to organize the Excel data by columns instead of rows. This is what transpose does. Compare the two preview bubbles in the first two nodes shown in Figure 8 to visualize this. The remaining five nodes in Group 5 are devoted to matching up the correct occupancy multipliers with each room name value.

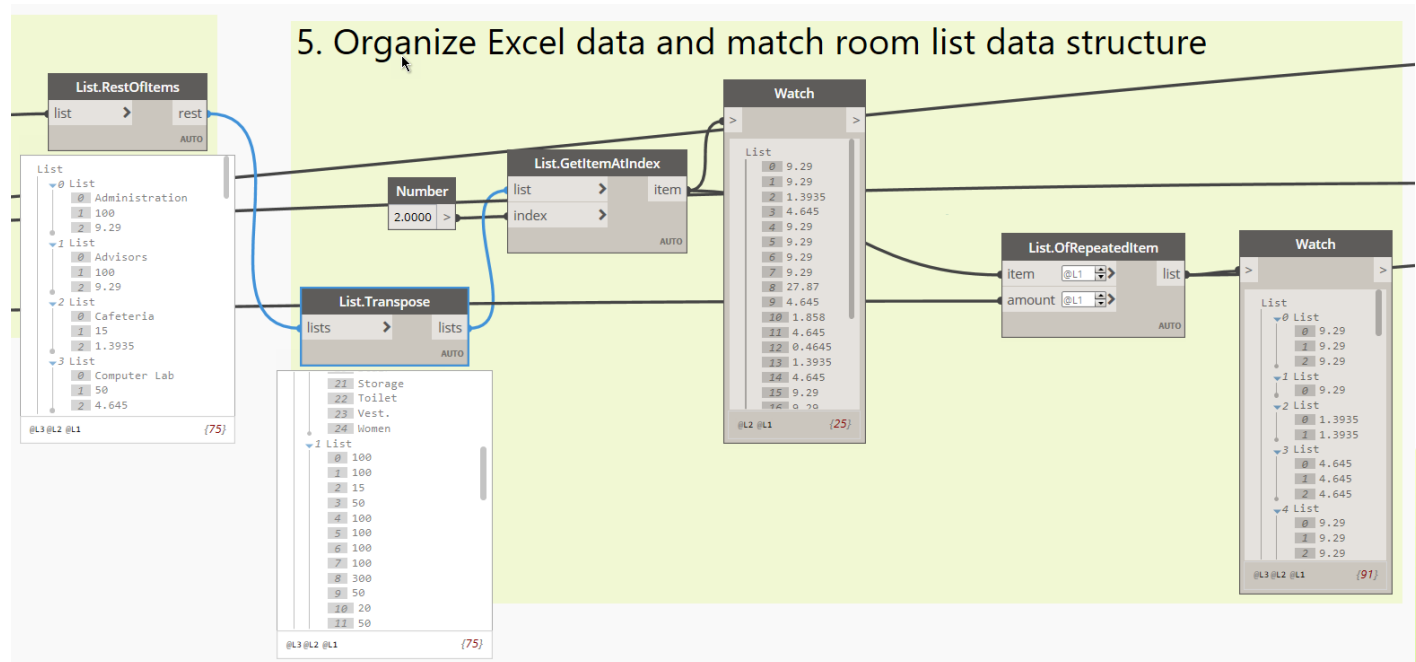


Figure 8 – Organize and match the Excel data to the rooms

Let's dig into the logic a bit. We start with a **List.GetItemAtIndex** node. This node simply uses the index value you feed it (from a **Number** node) to extract the corresponding value from a list. In this case, we are feeding in the transposed list of Excel data, but since we are only asking for index 2, we are getting just the metric occupancy multipliers. You can see this in the first **Watch** node. But this is only 25 values since the Excel file only contains one value for each unique room name. In order to make this work correctly with our model, we must ensure that we match the original list and its data structure.



Remember our example above, the first list contained three instances of rooms called Administration. We need *each* of these rooms to use the 9.29 occupancy multiplier. This gets us back to why we wanted the count of each sub list in Group 3 above. Using that list of counts we feed it into the **List.OfRepeatedItem** node. This node will repeat one or more values fed into it using the quantity fed into the amount input. So here, we feed out list of unique room occupancy multipliers from Excel (25 total) and repeat each one the number of times coming from the **List.Count** node above. We will end up with the structured list show in the final Watch node and if you compare its structure to the previous structured lists, it is a perfect match and we are back to 91 items!

#### Group 6

Group 6 contains four nodes (See Figure 9).

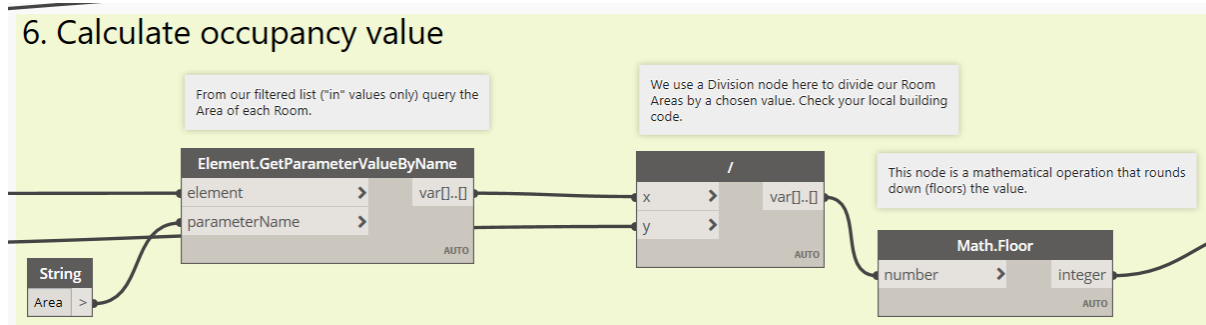


Figure 9 – Perform the calculations

Here we are going back to the original list of Rooms and getting another parameter; the Area this time. With this information, we can use the occupancy multipliers and calculate our final required occupancy values. A simple division node (/) is used here for this purpose. **Math.Floor** is similar to rounding, but eliminates any fractional value and rounds down to the nearest whole number.

#### Group 7

Group 7, our final group has just three nodes (See Figure 10).

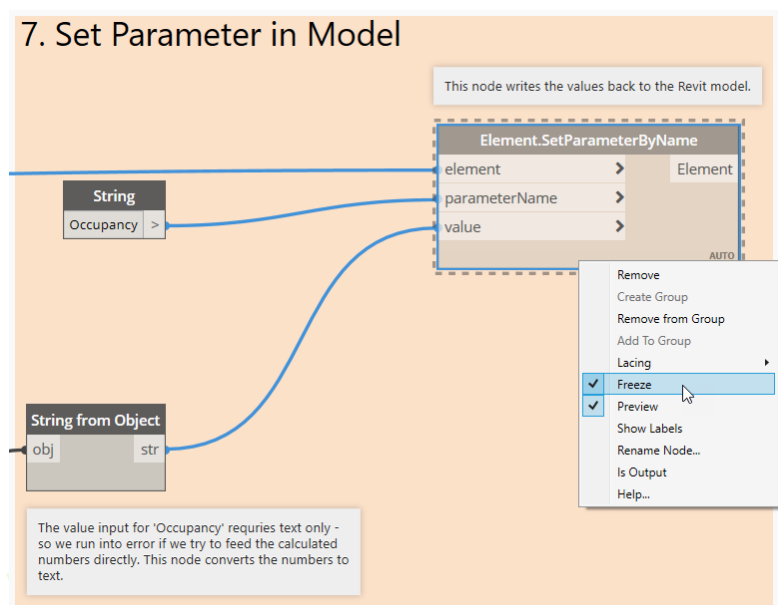


Figure 10 – Set the room occupancy values in the model

Earlier we used two “Get Parameter” nodes to read data from Revit. Here we will be modifying the Revit model using a “Set” node. The **Element.SetParameterByName** node writes data back to Revit. So, with our list of calculated values, this node will update all our rooms and write the required occupancy values to them. Two things to note in this group. First is the **String from Object** node. The Occupancy field in Revit is a text parameter. Therefore, we must convert our list of numbers to text before writing to Revit, or it will fail. This node does that. Second, the final node is “frozen.” This means that you must unfreeze it before it will execute this final node. This is a nice way to run part of a graph and wait till you are ready to complete the results. So be sure to right-click and choose Freeze to uncheck it and then run the graph to complete the task. Once you have done this, you can select any room in the model, and it will have an occupancy value added to it. Or you can create a schedule that shows the occupancy values to see them all at once.

## Automatically Determine and Place Wall Type Details

Many firms have standard wall type details that are used in each project. These are often already included in the project template. In this example, we will use Dynamo to determine which wall types are currently in use in a project and then use that information to place the required wall type details.

### *Why Use Dynamo for this task?*

Placing views on sheets is an easy task in Revit. The reason that this is a good task for Dynamo is that as your list of partition types grows, you will find it better if we can reliably ensure that all details called out in the project are properly represented on the partition type sheet. This is not hard to do if there are only a few, but when there are many, it can become a tedious and possibly error prone task. That’s why Dynamo can help.

### *Framing the problem*

So, what is involved in getting Dynamo to do this task for you? As we did above, you must frame the problem in a way that Dynamo will understand and be able to process regardless of the input you feed in. Here we need Dynamo to look at our model, figure out which wall types we are using, then make a sheet and place the required details on that sheet.

## Explore the Graph

To start this example, we have a simple Revit model and a starter Dynamo graph. Like the previous example, we’ll go through the graph one group at a time. The first group is incomplete. In this example, we’ll add the missing nodes and wire them together using the notes included in the group for some first-hand practice (see Figure 11).

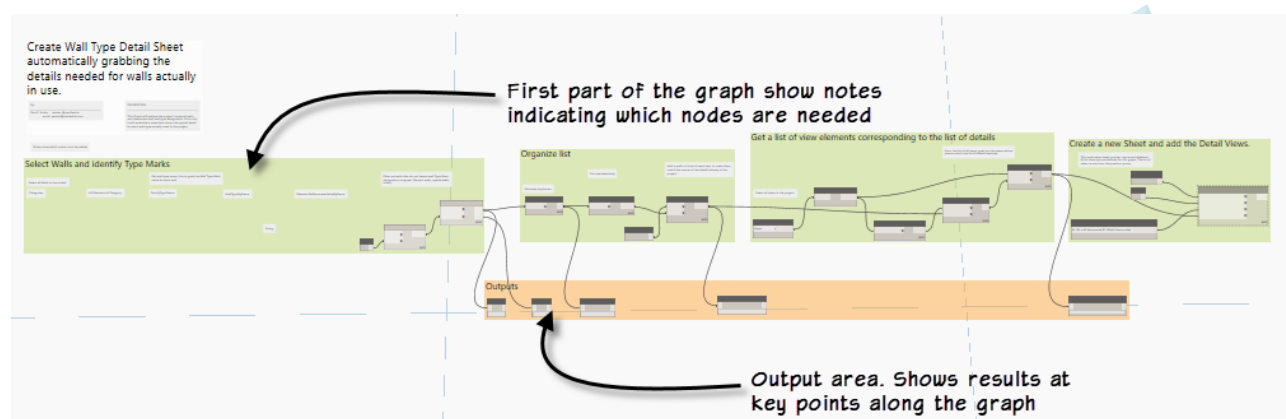


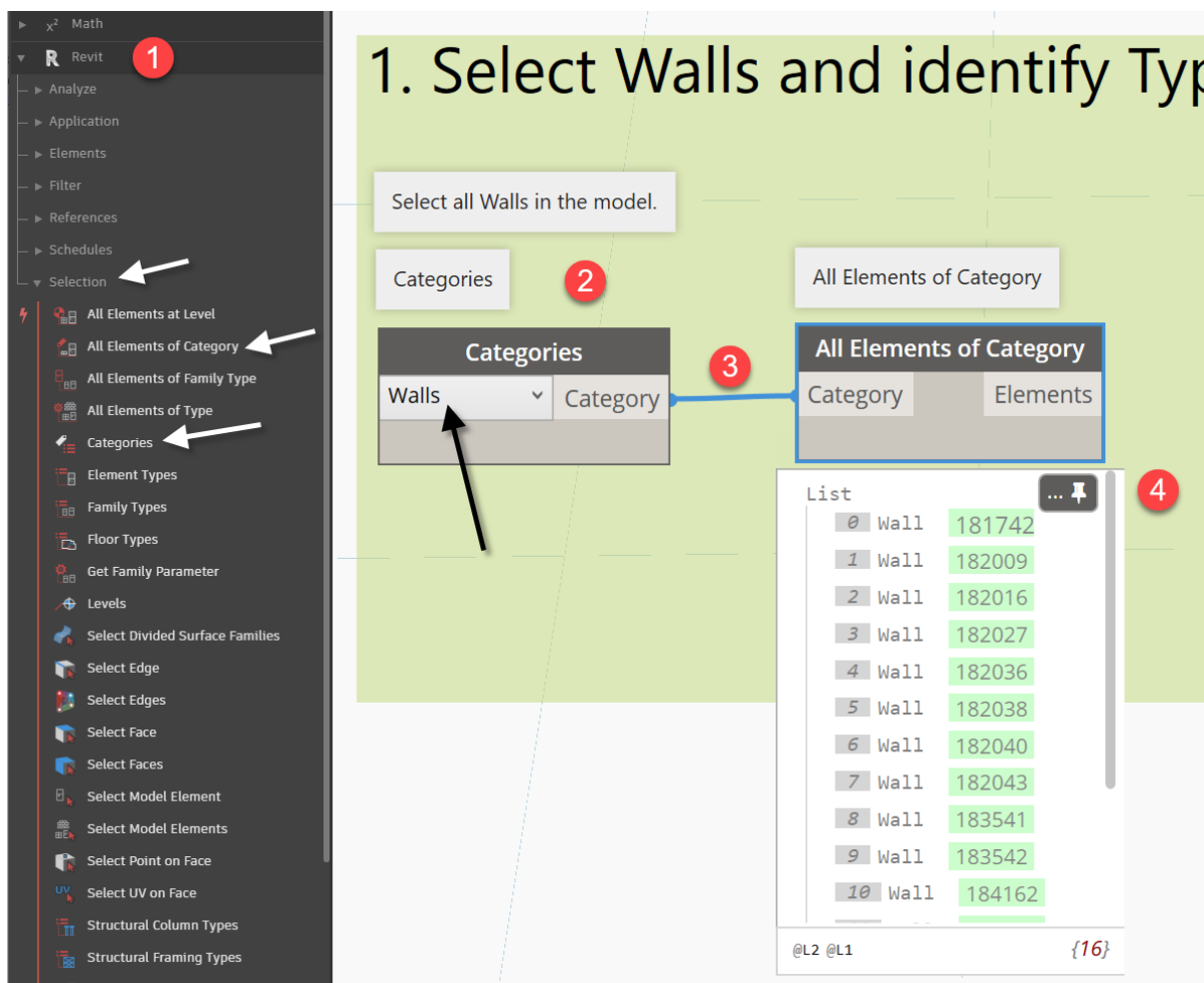
Figure 11 – Starter graph

### Build Group 1

The first note in group 1 indicates selection of all walls in the model. This is accomplished with two nodes, both indicated by separate notes: **Categories** and **All Elements of Category**. You can find both nodes in the library beneath *Revit > Selection*.

Click on the name of each node in the library to add it to the canvas. Position them beneath the notes in group 1. From the **Categories** node, choose **Walls**, then wire its output to the input of **All Elements of Category**.

At the bottom of the Dynamo window, click the Run button to execute the graph. Then hover over the **All Elements of Category** node. The preview bubble will appear. Click the small pushpin icon to pin this open. It will indicate that 16 walls have been selected. Note the green highlighted element IDs next to each wall. These are the Revit element IDs and you can click on them to zoom the Revit window directly to this item in the model. But another important factor of having those green IDs is that they indicate that the item is a Revit element (see Figure 12).



1. Select Walls and identify Type

Select all Walls in the model.

Categories

All Elements of Category

Categories

Walls

Category

All Elements of Category

Category

Elements

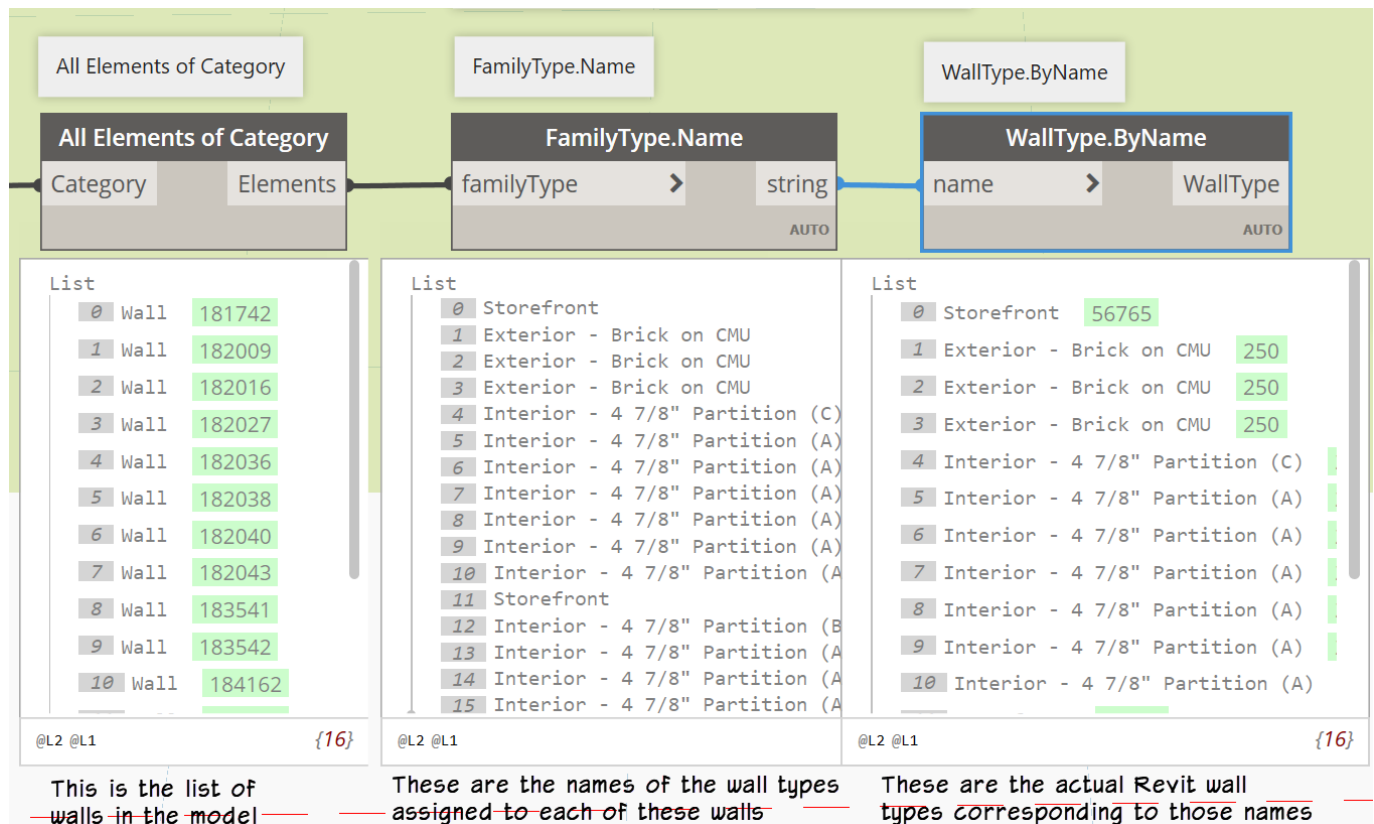
List

0	Wall	181742
1	Wall	182009
2	Wall	182016
3	Wall	182027
4	Wall	182036
5	Wall	182038
6	Wall	182040
7	Wall	182043
8	Wall	183541
9	Wall	183542
10	Wall	184162

@L2 @L1 {16}

Figure 12 – Select all the walls in the project with two simple nodes

From this list of individual wall elements, we next want to get the Revit Type name of each wall. The **FamilyType.Name** node is used for this purpose and can be found on the *Revit > Elements > FamilyType* branch of the library or by searching. Wire the output of **All Elements of Category** to this node. Next, from the *Revit > Elements > WallType* branch (or by searching), place the **WallType.ByName** node. Wire this to the output of the last node (see Figure 13).



It might not be clear why all these nodes are required. But the outputs of each node give some clues. The **All Elements of Category** outputs individual model elements. The next node: **FamilyType.Name** gives the names (as text, or “strings”) of the types used by each of those elements. The final node selects the actual Revit Wall Types based on the names in the middle list.

From the *Revit > Elements > Element* branch we next need an **Element.GetParameterValueByName** node. As we have seen before, this allows us to grab the value of the named parameter. A String node will allow us to indicate the parameter name we need. In this case, that parameter is: “Type Mark.” It is case sensitive, so be sure to type is correctly. Type Mark values are what show in our wall tags and how we indicate each wall type in the model and their corresponding details (see Figure 14).



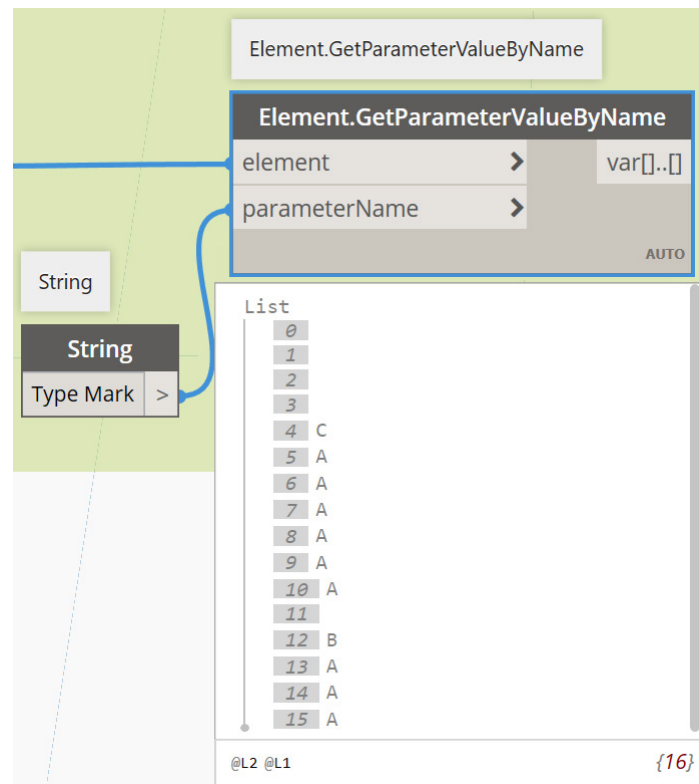


Figure 14 – Grab the Type Mark values of each wall type

After you run the graph and expand the preview bubble, you will note that some walls do not have a Type Mark value. We'll filter those out next. A **FilterByBoolMask** node is already in the graph for this purpose. The String that we will use for filtering is empty. The **!=** node checks if the values fed into the x port are not equal to the one fed into the y port. This gives a list of true and false values. This can be used to branch our output into two lists with the boolmask (see Figure 15).

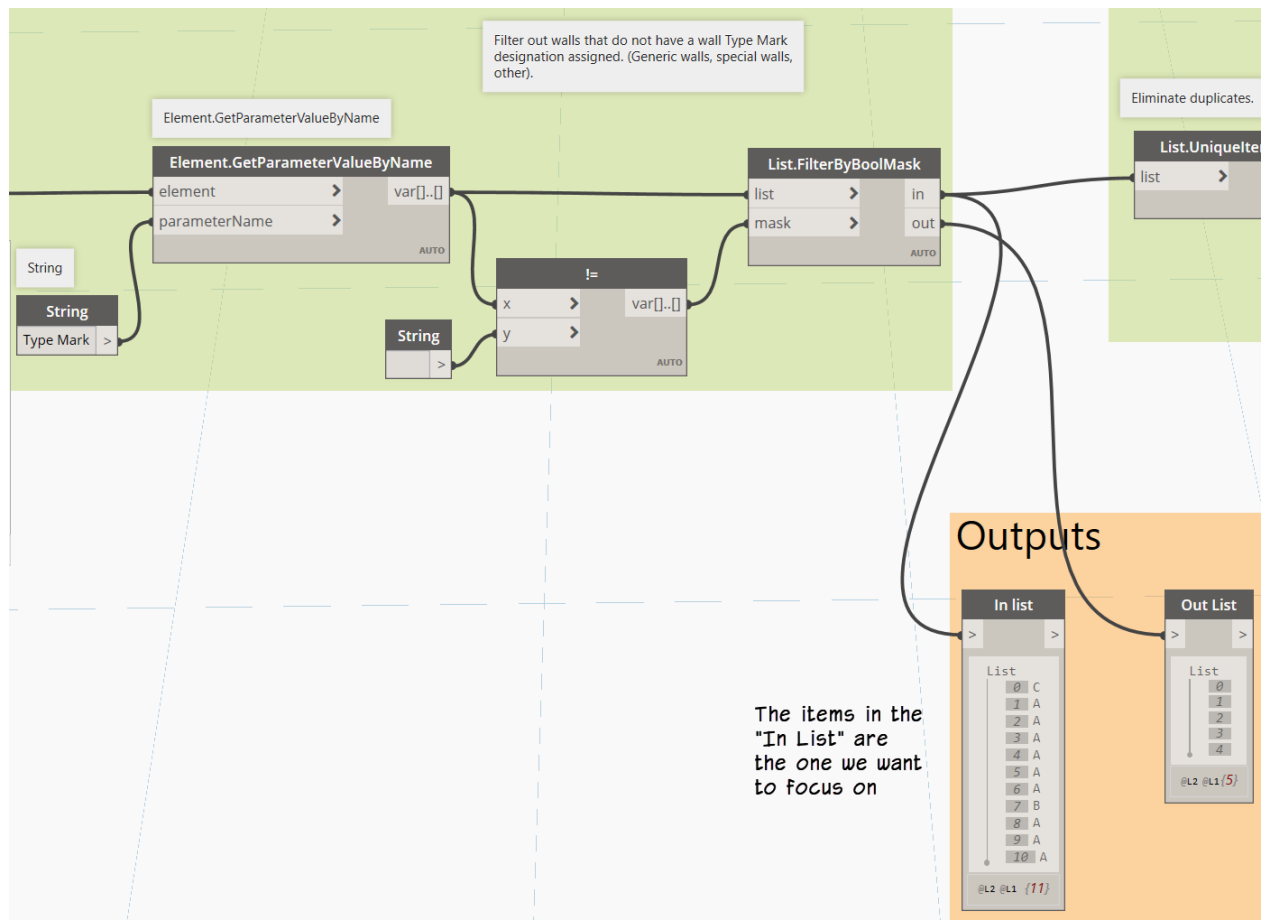


Figure 15 – Filter the list into two lists, those with Type Marks and those without

The **List.FilterByBoolMask** node has two outputs. I will frequently use **Watch** nodes to report the values when there are multiple outputs. This just keeps it cleaner. You can see those in the orange group called "Outputs." These are just for information only and are optional. The remainder of the graph is already built in the starter file. Let's walk through the rest of it now.

### Group 2

Group 2 has three nodes. The first takes the list of "In" elements from the BoolMask and finds only the unique items. Next this list is sorted, and then using a **+** node, we add a prefix to the front of the names to make them match the names used for the drafting details back in our Revit project file (see Figure 16).



On to group 3. Here we will match up the detail views in the project with the names we just constructed. First, using **Categories** and **All Elements of Category** again, we select all the Views this time. From the list, we grab their names (**Element.Name**) and then use the **List.IndexOf** node to figure out where each view name occurs on the list. This is asking for the specific index (or address) of the named item on the list of view names (see Figure 17).

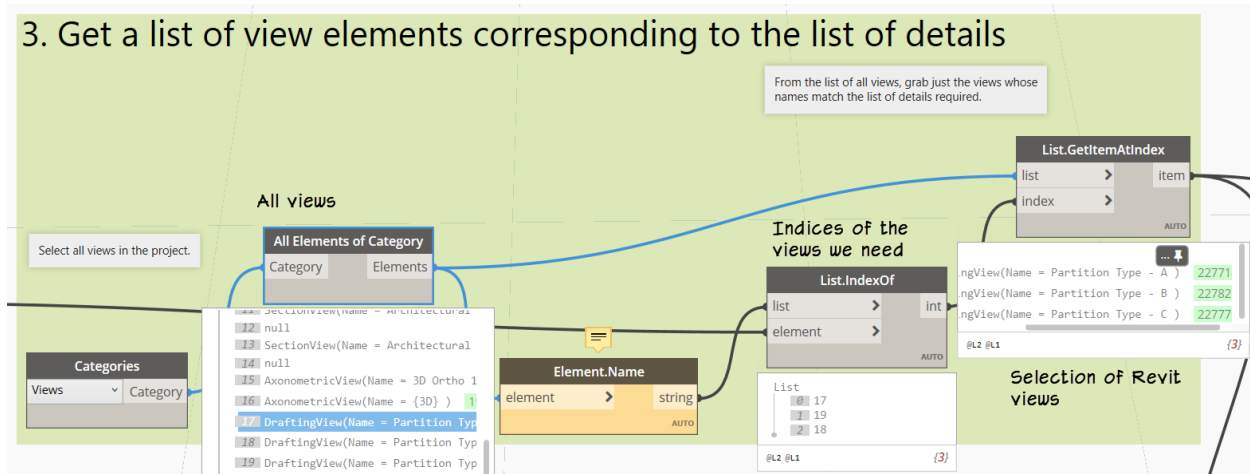


Figure 17 – Locate the required detail views from the list of all project views

#### Group 4

The final group uses the **Sheet.ByNameNumberTitleBlockAndViews** node. This node creates a sheet and adds a list of views. Satisfying the first two inputs is accomplished with simple **String** nodes. The **titleBlockFamilyType** input is satisfied with a **Family Types** node. To satisfy the list of **views**, we simply feed in our list from group 3. The final node is frozen, so be sure to right-click it and un-freeze before running.

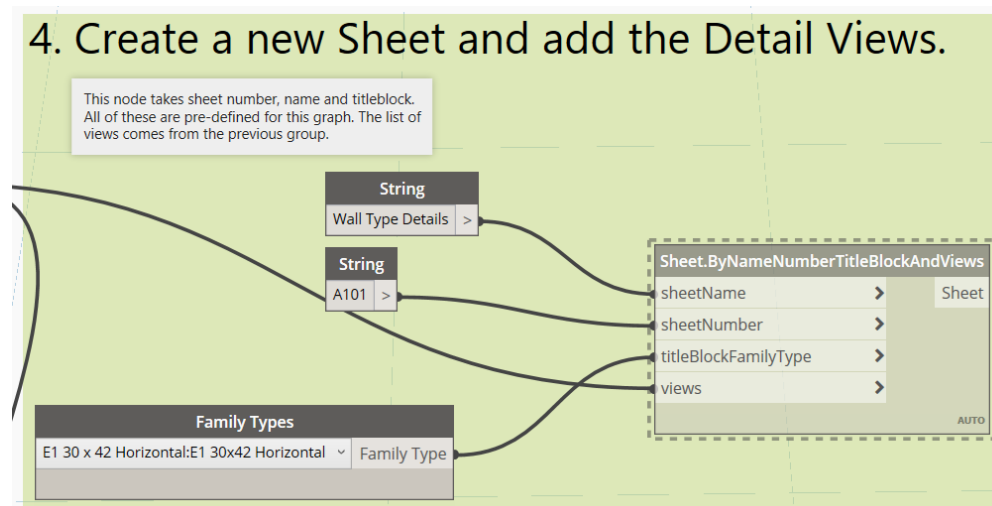


Figure 18 – Create a sheet and populate with the list of required details

When you run the graph, you should see a sheet appear in Revit. If you open that sheet, each of the views from group 3 will have been placed on the sheet. They end up in the lower left corner of the sheet. Simply move them where you wish them to go on the sheet (see Figure 19).



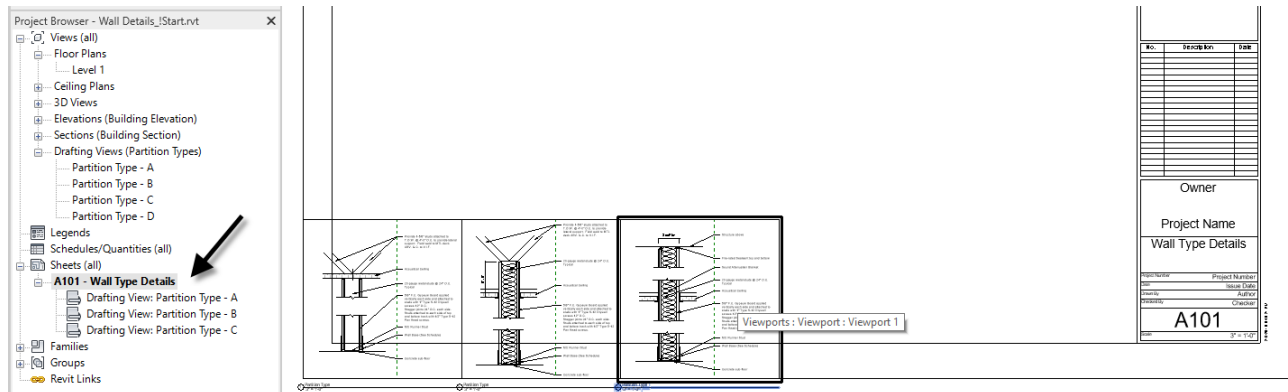


Figure 19 – The resulting sheet with its placed details

This graph achieves the basic requirements. If you wish, you can enhance it to move the viewports to a better location for you instead of doing that manually. You can also create a second graph to update this sheet should the required details change. I will leave those tasks to you as a challenge exercise if you are interested. But keep in mind that you can also just delete the sheet and run the graph again in the future. It is up to you.

## Conclusion

These were two fairly simple examples. But hopefully they help illustrate some of the tasks you should consider using Dynamo for. Anytime you have a well-defined task that is repetitive or laborious, you should consider Dynamo as an alternative to manual processing. Not every repetitive task will be improved with Dynamo. The best candidates are those that you can clearly and quickly define the logic and steps for. This will help you reap the benefits of automation. If you spend hours building a graph that only saves a few minutes, that is probably not the best use of your time. However, if the initial investment in building the graph can be recouped on the current or future projects, then it is likely a good candidate for Dynamo automation.