Revit API Intro Labs

Lab3 – Element Filtering

Created by M. Harada, July 2010
Updated by DevTech AEC WG
Last modified: 10/15/2023

<C#>C# Version**</C#>**

Objective: In this lab, we will learn how to obtain the elements that you are interested in, using filtering mechanism in Revit API. We'll learn how to:

- Retrieve family types
- Retrieve instances of a specific object class
- Find a specific family type
- Find specific instances

Tasks: We'll write a command that accumulatively demonstrates various methods and approaches for filtering elements. Use this lab as an exercise to familiarize yourself with filtering.

1. List family types (e.g., wall types, floor types, and door types)
2. List instances of a specific type of objects (e.g., walls and doors)
3. Find a specific family type (e.g., "Basic Wall: Generic – 200mm" wall type, "M_Single-Flush: 0915 x 2134mm" door type)
4. Find specific Instances (i.e., instances of "Basic Wall: Generic – 200mm" wall type, instances of "M_Single-Flush: 0915 x 2134mm" door type, and walls that are longer than a certain length.)

Figure 1 shows the sample images of output after running the command that you will be defining in this lab:

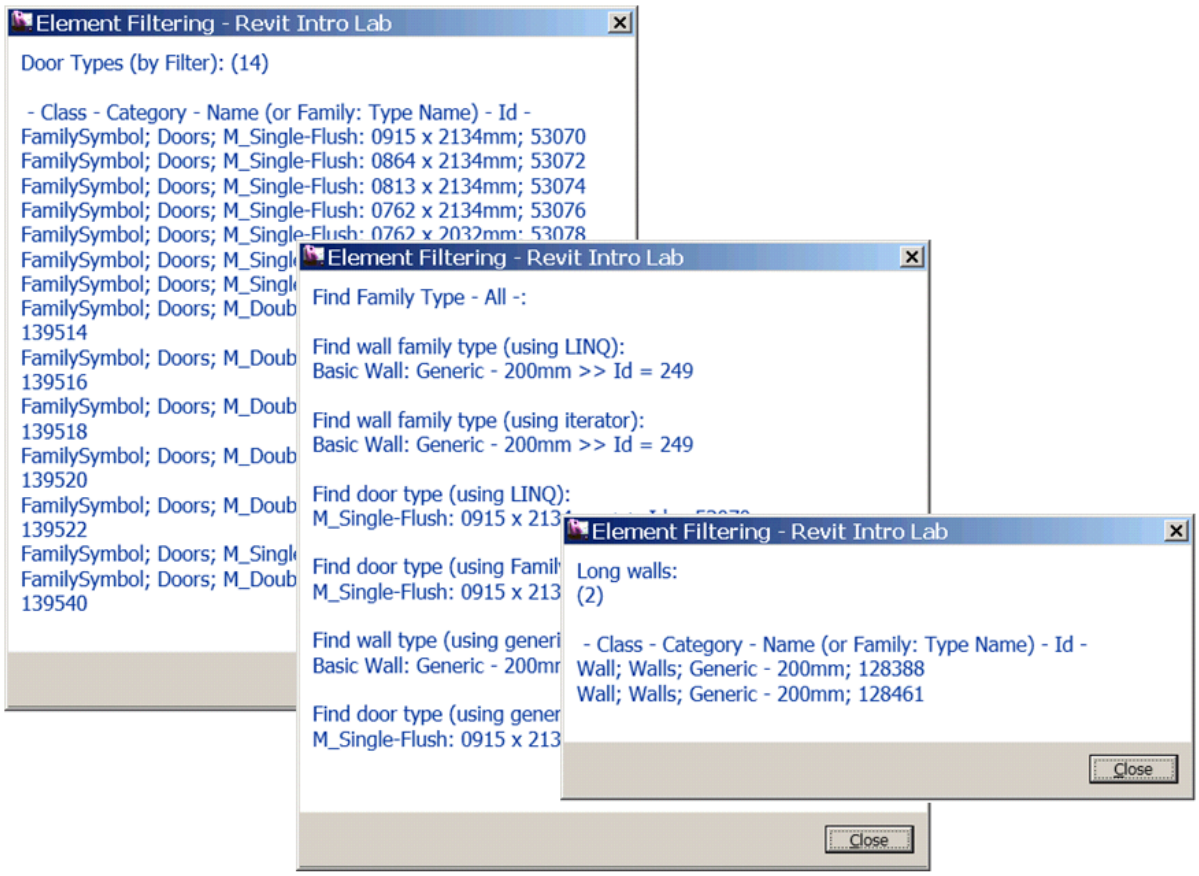

Figure 1. Dialogs showing the result of various filtering

The following is the breakdown of step by step instructions in this lab:

1. Define a New External Command
2. List Family Types
3. List Instances of Specific Object Class
4. Find a Specific Family Type
5. Find Specific Instances
6. Summary

1. Define A New External Command

We'll add another external command to the current project.

1.1 Add a new file and define another external command to your project. Let's name them as follows:

- File name: **3_ElementFiltering.vb (or .cs)**
- Command class name: **ElementFiltering**

(Once again, you may choose to use any names you want here. When you do so, just remember what you are calling your own project, and substitute these names as needed while following the instruction in this document.)

Required Namespaces:
Namespaces needed for this lab are:

- System.Linq
- Autodesk.Revit.DB
- Autodesk.Revit.UI
- Autodesk.Revit.ApplicationServices
- Autodesk.Revit.Attributes

Note (VB.NET only): if you are writing in VB.NET and you import namespaces at the project level, (i.e., in the project properties, there is no need to explicitly import in each file.

1.2 Like we did in Lab2, define member variables, e.g., m_rvtApp and m_rvtDoc, to keep DB level application and document respectively. The following is an example:

<C#>
```
// Element Filtering - learn about Revit element
[Transaction(TransactionMode.Manual)]
public class ElementFiltering : IExternalCommand
{
    //  member variables
    Application m_rvtApp;
    Document m_rvtDoc;

    public Result Execute(
        ExternalCommandData commandData,
        ref string message,
        ElementSet elements)
    {
        //  Get the access to the top most objects.
        UIApplication rvtUIApp = commandData.Application;
        UIDocument rvtUIDoc = rvtUIApp.ActiveUIDocument;
        m_rvtApp = rvtUIApp.Application;
        m_rvtDoc = rvtUIDoc.Document;

        //  ...

        return Result.Succeeded;
    }
}
```
</C#>

2. List Family Types

In the previous lab, we have learned that depending on whether the element is component family-based or system family, we will need to take a different approach to identify an element, using class names and categories. When we are retrieving the list of family types stored in the project database, the similar rules applies.

2.1 System Family Types

You can think of Revit elements as a bundle in a large sack and it is in a database. To access elements in it, you will need to query for it. As an example, the following will collect all the WallType class in the document:

```
<C#>
    var wallTypeCollector1 = new FilteredElementCollector(m_rvtDoc);
    wallTypeCollector1.WherePasses(new ElementClassFilter(typeof(WallType)));
    IList<Element> wallTypes1 = wallTypeCollector1.ToElements();
</C#>
```

FilteredElementCollector is a “container” object to collect elements which we are interested in. We first create it. And it passes through a filter, in this case, a class filter to filter out only to collect elements whose class is WallTypes. The last line converts a filtered element collector into a list of element; this is for convenience for further handling.

In our context, using filtering to retrieve a list of wall types may not be seen as a valuable thing to do. But this may become more convenient if your query becomes more complex.

Revit API offers various alternative forms for filtering for convenience. The following are the same as first two lines in the above code, but using OfClass():

```
<C#>
      FilteredElementCollector wallTypeCollector2 =
          new FilteredElementCollector(m_rvtDoc);
      wallTypeCollector2.OfClass(typeof(WallType));
</C#>
```

You can further simplify it using shortcut:

```
<C#>
    FilteredElementCollector wallTypeCollector3 =
        new FilteredElementCollector(m_rvtDoc).OfClass(typeof(WallType));
</C#>
```

2.3 Component Family Types

For component family, you will need to check element class and categories. Following is an example of getting a list of door family types.

```
<C#>
      var doorTypeCollector = new FilteredElementCollector(m_rvtDoc);
      doorTypeCollector.OfClass(typeof(FamilySymbol));
      doorTypeCollector.OfCategory(BuiltInCategory.OST_Doors);
      IList<Element> doorTypes = doorTypeCollector.ToElements();
</C#>
```

The following code demonstrates the usage:

```
<C#>
    public void ListFamilyTypes()
    {
      //
      // (1) get a list of family types available in the current rvt project.


      // (1.1a) here is an example with wall type.

      var wallTypeCollector1 = new FilteredElementCollector(m_rvtDoc);
      wallTypeCollector1.WherePasses(
          new ElementClassFilter(typeof(WallType)));
      IList<Element> wallTypes1 = wallTypeCollector1.ToElements();

      // using a helper funtion to display the result here. See code below.

      ShowElementList(wallTypes1, "Wall Types (by Filter): ");

      // (1.1b) the following are the same as two lines above.
      // these alternative forms are provided for convenience.
      // using OfClass()
      //
      //FilteredElementCollector wallTypeCollector2 =
      //    new FilteredElementCollector(m_rvtDoc);
      //wallTypeCollector2.OfClass(typeof(WallType));

      // (1.1c) the following are the same as above. For convenience.
      // using short cut this time.
      //
      //FilteredElementCollector wallTypeCollector3 =
      //    new FilteredElementCollector(m_rvtDoc).OfClass(typeof(WallType));

      //
      // (2) Listing for component family types.
      //
      // For component family, it is slightly different.
      // There is no designate property in the document class.
      // You always need to use a filtering. e.g., for doors and windows.
      // Remember for component family, you will need to check
      // element type and category.

      var doorTypeCollector = new FilteredElementCollector(m_rvtDoc);
      doorTypeCollector.OfClass(typeof(FamilySymbol));
```

```
      doorTypeCollector.OfCategory(BuiltInCategory.OST_Doors);
      IList<Element> doorTypes = doorTypeCollector.ToElements();

      ShowElementList(doorTypes, "Door Types (by Filter): ");
    }


   // Helper function to display info from a list of elements passed onto.

    public void ShowElementList(IList<Element> elems, string header)
    {
      string s = " - Class - Category - Name (or Family: Type Name) - Id -
\r\n";
      foreach (Element e in elems)
      {
        s += ElementToString(e);
      }
      TaskDialog.Show(header + "(" + elems.Count.ToString() + "):", s);
    }


    // Helper function: summarize an element information as a line of text,
    // which is composed of: class, category, name and id.
    // name will be "Family: Type" if a given element is ElementType.
    // Intended for quick viewing of list of element, for example.

    public string ElementToString(Element e)
    {
      if (e == null)
      {
        return "none";
      }

      string name = "";

      if (e is ElementType)
      {
        Parameter param = e.get_Parameter(
            BuiltInParameter.SYMBOL_FAMILY_AND_TYPE_NAMES_PARAM);
        if (param != null)
        {
          name = param.AsString();
        }
      }
      else
      {
        name = e.Name;
      }
      return e.GetType().Name + "; "
        + e.Category.Name + "; "
        + name + "; "
        + e.Id.IntegerValue.ToString() + "\r\n";
    }
</C#>
```

To test this, you can call ListFamilyTypes() from the main Execute() method.

Discussion:

- Give examples of family types other than walls, floor and doors.
- Which methods can we use to retrieve them?

Exercise:

- Choose one class of object, write a code to retrieve all its family types.

3. List Instances of a Specific Object Class

To get a list of instances of specific object type, you will need to use filters. The same idea that we learned for family types applies for instances as well.

Here is an example of collecting all the wall instances:

```
<C#>
      var wallCollector = new
          FilteredElementCollector(m_rvtDoc).OfClass(typeof(Wall));
      IList<Element> wallList = wallCollector.ToElements();
</C#>
```

Here is another for collecting all the doors.

```
<C#>
      var doorCollector = new
          FilteredElementCollector(m_rvtDoc).OfClass(typeof(FamilyInstance));
      doorCollector.OfCategory(BuiltInCategory.OST_Doors);
      IList<Element> doorList = doorCollector.ToElements();
</C#>
```

Discussion:

- Give examples of instances other than walls, floor and doors.
- Which methods can we use to retrieve them?

Exercise:

- Write a code to retrieve all the instances of windows (or your choice.)

4. Find a Specific Family Type

In this section, we will look at the way to find a specific family type. Let's say, we would like to retrieve:

- wall type - "Basic Wall: Generic – 200mm"

- door type – “M_Single-Flush: 0915 x 2134mm”

4.1 Find a wall type with a given name.

Let’s start with the wall. There are a few different ways to do this. The first version is to use LINQ query.

```
<C#>
    // Find a specific family type for a wall with a given family and type
    // names. This version uses LINQ query.

    public Element FindFamilyType_Wall_v1(
        string wallFamilyName,
        string wallTypeName)
    {
      // narrow down a collector with class.

      var wallTypeCollector1 = new FilteredElementCollector(m_rvtDoc);
      wallTypeCollector1.OfClass(typeof(WallType));

      // LINQ query

      var wallTypeElems1 =
          from element in wallTypeCollector1
          where element.Name.Equals(wallTypeName)
          select element;

      // get the result.

      Element wallType1 = null; // result will go here.

      // (1) directly accessing from the query result.

      if (wallTypeElems1.Count<Element>() > 0)
      {
        wallType1 = wallTypeElems1.First<Element>();
      }

      // (2) If you want to get the result as a list of element, here is how.

      //IList<Element> wallTypeList1 = wallTypeElems1.ToList();
      //if (wallTypeList1.Count > 0)
      //  wallType1 = wallTypeList1[0]; // Found it.

      return wallType1;
    }
</C#>
```

The second version uses iterations:

```
<C#>
    // Find a specific family type for a wall, which is a system family.
    // This version uses iteration. (cf. Developer guide 89)
    //
```

```
    public Element FindFamilyType_Wall_v2(
        string wallFamilyName,
        string wallTypeName)
    {
      // first, narrow down the collector by Class
      var wallTypeCollector2 =
          new FilteredElementCollector(m_rvtDoc).OfClass(typeof(WallType));

      // use iterator
      FilteredElementIterator wallTypeItr =
          wallTypeCollector2.GetElementIterator();
      wallTypeItr.Reset();

      Element wallType2 = null;

      while (wallTypeItr.MoveNext())
      {
        WallType wType = (WallType)wallTypeItr.Current;
        // we check two names for the match: type name and family name.
        if ((wType.Name == wallTypeName) &
(wType.get_Parameter(BuiltInParameter.SYMBOL_FAMILY_NAME_PARAM).AsString().Eq
uals(wallFamilyName)))
        {
          wallType2 = wType; // we found it.
          break;
        }
      }

      return wallType2;
    }
</C#>
```

4.2 Finding a door type with a given name

Similarly, for door types, you can also approach different ways. The first version uses LINQ query:

```
<C#>
    // Find a specific family type for a door, which is a component family.
    // This version uses LINQ.

    public Element FindFamilyType_Door_v1(
      string doorFamilyName,
      string doorTypeName)
    {
      // narrow down the collection with class and category.

      var doorFamilyCollector1 = new FilteredElementCollector(m_rvtDoc);
      doorFamilyCollector1.OfClass(typeof(FamilySymbol));
      doorFamilyCollector1.OfCategory(BuiltInCategory.OST_Doors);

      // parse the collection for the given name
      // using LINQ query here.
```

```
      var doorTypeElems =
          from element in doorFamilyCollector1
          where element.Name.Equals(doorTypeName) &&
          element.get_Parameter(
              BuiltInParameter.SYMBOL_FAMILY_NAME_PARAM).
              AsString().Equals(doorFamilyName)
          select element;

      // get the result.

      Element doorType1 = null;

      // (1) directly accessing from the query result
      // we should have only one with the given name. minimum error checking.

      //if (doorTypeElems.Count > 0)
      // doorType1 = doorTypeElems(0) // found it.

      // (2) if we want to get the list of element, here is how.

      IList<Element> doorTypeList = doorTypeElems.ToList();
      if (doorTypeList.Count > 0)
      {
        // we should have only one with the given name.
        // minimum error checking
        // found it.

        doorType1 = doorTypeList[0];
      }

      return doorType1;
    }
</C#>
```

Another approach will be to look up a family name from Family, then the type name from Family.Symbols property. Although this is a logical approach, it looks more complex:

```
<C#>

    // Find a specific family type for a door.
    // another approach will be to look up from Family,
    // then from Family.Symbols property.
    // This gets more complicated although it is logical approach.
    //
    public Element FindFamilyType_Door_v2(
        string doorFamilyName,
        string doorTypeName)
    {
      // (1) find the family with the given name.

      var familyCollector = new FilteredElementCollector(m_rvtDoc);
      familyCollector.OfClass(typeof(Family));

      // use the iterator
```

```
      Family doorFamily = null;
      FilteredElementIterator familyItr =
          familyCollector.GetElementIterator();
      //familyItr.Reset()

      while ((familyItr.MoveNext()))
      {
        Family fam = (Family)familyItr.Current;
        // check name and categoty
        if ((fam.Name == doorFamilyName) &
(fam.FamilyCategory.Id.IntegerValue == (int)BuiltInCategory.OST_Doors))
        {
          // we found the family.
          doorFamily = fam;
          break;
        }
      }

      // (2) find the type with the given name.

      Element doorType2 = null;
      // id of door type we are looking for.
      if (doorFamily != null)
      {
        ISet<ElementId> familySymbolIds = doorFamily.GetFamilySymbolIds();

        if (familySymbolIds.Count > 0)
        {
          // Get family symbols which is contained in this family
          foreach (ElementId id in familySymbolIds)
          {
            FamilySymbol dType = doorFamily.Document.GetElement(id)
                as FamilySymbol;
            if ((dType.Name == doorTypeName))
            {
              doorType2 = dType;  // Found it.
              break;
            }
          }
        }
      }

      return doorType2;
    }
```

</C#>

4.3 Defining a more generalized function

So far, we have defined a filter for individual cases. Sometimes having a more generalized form of function to retrieve an element of a given family and type name may get handy. The following function

takes a document, the name of family, the name of type, and optional category information as arguments, and returns the family type found in the document:

<C#>

```
   //  find an element of the given type, name, and category(optional)

    public static Element FindFamilyType(
        Document rvtDoc, Type targetType,
        string targetFamilyName,
        string targetTypeName,
        Nullable<BuiltInCategory> targetCategory)
    {
      // first, narrow down to the elements of the given type and category

      var collector =
          new FilteredElementCollector(rvtDoc).OfClass(targetType);
      if (targetCategory.HasValue)
      {
        collector.OfCategory(targetCategory.Value);
      }

      // parse the collection for the given names
      // using LINQ query here.

      var targetElems =
          from element in collector
          where element.Name.Equals(targetTypeName) &&
          element.get_Parameter(BuiltInParameter.SYMBOL_FAMILY_NAME_PARAM).
          AsString().Equals(targetFamilyName)
          select element;

      // put the result as a list of element fo accessibility.

      IList<Element> elems = targetElems.ToList();

      // return the result.

      if (elems.Count > 0)
      {
        return elems[0];
      }

      return null;
    }
```

</C#>

Using this function, you can find a family type with a given name as follows, e.g.,:

<C#>

```
ElementType wallType3 =
```

```
    (ElementType)FindFamilyType(m_rvtDoc, typeof(WallType),
    "Basic Wall", "Generic - 200mm", null);

ElementType doorType3 =
    (ElementType)FindFamilyType(m_rvtDoc, typeof(FamilySymbol),
    "M_Single-Flush", "0915 x 2134mm", BuiltInCategory.OST_Doors);
</C#>
```

Exercise:

- Implement FindFamilyType() that retrieves a family type of given name and return the family type.
- Using FindFamilyType(), retrieve a wall, door and window type of your choice. (You can hard code the family names.)

5. Find Specific Instances

5.1 Find Instances of a given family type

Another situation might be that we want to retrieve instances of a given family type. The following function takes a class, the element id of a certain family type, and optional category, and returns a list of elements that are instance of the given family type:

```
<C#>

    //  find a list of element with given class, family type and
    //  category (optional).
    public IList<Element> FindInstancesOfType(
        Type targetType,
        ElementId idType,
        Nullable<BuiltInCategory> targetCategory = null)
    {
      // narrow down to the elements of the given type and category

      var collector =
          new FilteredElementCollector(m_rvtDoc).OfClass(targetType);
      if (targetCategory.HasValue)
      {
        collector.OfCategory(targetCategory.Value);
      }

      // parse the collection for the given family type id.
      // using LINQ query here.

      var elems =
          from element in collector
          where element.get_Parameter(BuiltInParameter.SYMBOL_ID_PARAM).
                AsElementId().Equals(idType)
```

```
          select element;

      // put the result as a list of element fo accessibility.

      return elems.ToList();
    }

</C#>
```

For example, using this function, you can find a list of instances of a given family type as follows:

```
<C#>

IList<Element> walls = FindInstancesOfType(typeof(Wall), idWallType, null);

IList<Element> doors = FindInstancesOfType(typeof(FamilyInstance),
    idDoorType, BuiltInCategory.OST_Doors);
</C#>
```

5.2 Find an element with a given class and a name

One other commonly used scenario would be to retrieve some supporting elements in Revit, such as Level and View element. The following functions will be another handy function to retrieve such as each Level element.

```
<C#>

    // find a list of element with given Class, Name and Category (optional).

    public static IList<Element> FindElements(
        Document rvtDoc,
        Type targetType,
        string targetName,
        Nullable<BuiltInCategory> targetCategory)
    {
      // first, narrow down to the elements of the given type and category

      var collector =
          new FilteredElementCollector(rvtDoc).OfClass(targetType);
      if (targetCategory.HasValue)
      {
        collector.OfCategory(targetCategory.Value);
      }

      // parse the collection for the given names
      // using LINQ query here.

      var elems =
          from element in collector
```

```
          where element.Name.Equals(targetName)
          select element;

      // put the result as a list of element for accessibility.

      return elems.ToList();
    }

    // ------------------------------------------------------------------
    // Helper function: searches elements with given Class, Name and
    // Category (optional),
    // and returns the first in the elements found.
    // This gets handy when trying to find, for example, Level.
    // e.g., FindElement(m_rvtDoc, GetType(Level), "Level 1")

    public static Element FindElement(Document rvtDoc, Type targetType,
        string targetName, Nullable<BuiltInCategory> targetCategory)
    {
      // find a list of elements using the overloaded method.
      IList<Element> elems =
          FindElements(rvtDoc, targetType, targetName, targetCategory);

      // return the first one from the result.
      if (elems.Count > 0)
      {
        return elems[0];
      }

      return null;
    }
```

</C#>

For example, using this function, you can find a list of instances of a given family type as follows:

<C#>

```
Level level1 = (Level)FindElement(m_rvtDoc, typeof(Level), "Level 1", null);
```

</C#>

We'll use this in the later labs when we want to create a simple house.

Exercise:

- Implement FindElements() function that takes a document, class, name and optional category as arguments, and returns a list of elements with the given class, name and category.
- Implement FindElement() that calls FindElements(), and returns only the first element in the list.
- Using FindElement(), retrieve a Level element of a given name (You can hard code the Level name)

5.3 Filtering with parameters (Optional)

By now, you must have familiarized yourself with the basics of the element filtering. More specifically, we have learned how to use the following classes:

- FilteredElementCollector
- ElementClassFilter
- ElementCategoryFilter

There are more different kinds of filters, such as:

- BoundingBoxContainsPointFilter
- ElementDesignOptionFilter
- ElementIsCurveDrivenFilter
- ElementIsElementTypeFilter
- ElementParameterFilter
- …

This section of the Revit developer documentation describes filtering. For more detail, please take a look at the documentation.

One thing to note is that a filter can be "Quick" or "Slow". We have not discussed about in this scope of the first day API training labs. But this is something you may be aware of. When the performance becomes of concern, you should definitely take a look at how to filter elements. If needed, you may perform a performance test among possible approaches as well.

Below is one example of parameter filter. The code uses parameter filter to check the wall's parameter value for length, which is equivalent of evaluating:

wall.parameter(length) > 60 feet

<C#>

```
//
// Optional - example of parameter filter.
// find walls whose length is longer than a certain length. e.g., 60 feet
//    wall.parameter(length) > 60 feet
// This could get more complex than looping through in terms of
// writing a code. See page 87 of Developer guide.

public IList<Element> FindLongWalls()
{
  // constant for this function.
  const double kWallLength = 60.0;
  // 60 feet. hard coding for simplicity.

  // first, narrow down to the elements of the given type and category
  var collector =
      new FilteredElementCollector(m_rvtDoc).OfClass(typeof(Wall));
```

```
        // define a filter by parameter
        // 1st arg - value provider
        BuiltInParameter lengthParam = BuiltInParameter.CURVE_ELEM_LENGTH;
        int iLengthParam = (int)lengthParam;
        var paramValueProvider =
            new ParameterValueProvider(new ElementId(iLengthParam));

        // 2nd - evaluator
        FilterNumericGreater evaluator = new FilterNumericGreater();

        // 3rd - rule value
        double ruleVal = kWallLength;

        // 4th - epsilon
        const double eps = 1E-06;

        // define a rule
        var filterRule =
            new FilterDoubleRule(paramValueProvider, evaluator, ruleVal, eps);

        // create a new filter
        var paramFilter = new ElementParameterFilter(filterRule);

        // go through the filter
        IList<Element> elems = collector.WherePasses(paramFilter).ToElements();

        return elems;
    }
```

</C#>

6. Summary

In this lab, we have learned how to filter elements. We have learned how to:

- Retrieve family types
- Retrieve instances of a specific object class
- Find a specific family type
- Find specific instances

In the next lab, we will take a look at how to modify elements in the Revit.