Lab4 – Element Modification

Created by M. Harada, July 2010
Updated by DevTech AEC WG
Last modified: 10/25/2023

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

Objective: In this lab, we will learn how to modify elements. We'll learn how to:

- Modify an element at element level by changing its properties, parameters and location
- Modify an element using transform utility methods, such as Move and Rotate

Tasks: We'll write a command that prompts the user to pick an element and modify its properties. It then prompts to pick an element once again and rotate it using utility methods.

1. Pick an element.
2. Modify its family type
3. Modify its parameters
4. Modify its location (Optional)
5. Pick an element
6. Move it by ElementTransformUtils.MoveElement() method
7. Rotate it by ElementTransformUtils.RotateElement() method

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

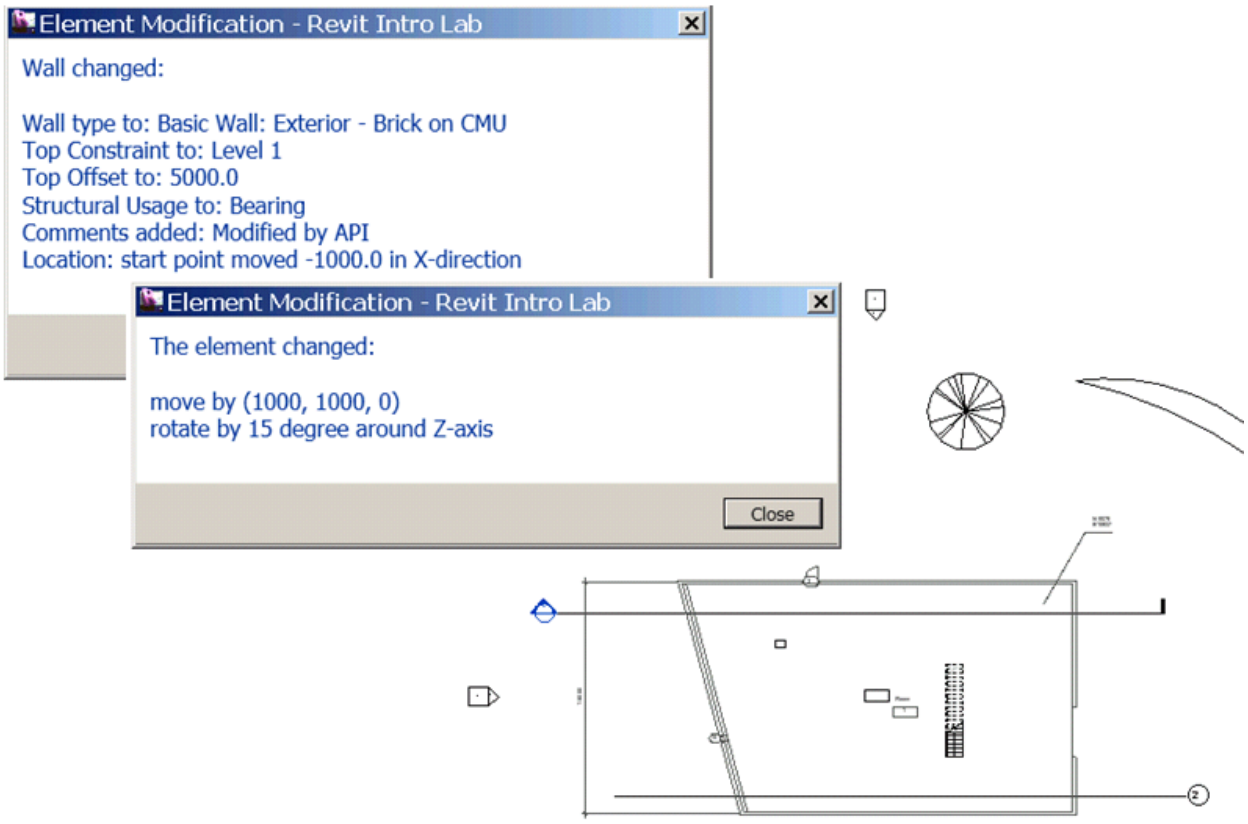

Figure 1. we'll make modification to an element by changing its properties and by using transformation utility methods.

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

1. Define a New External Command
2. Pick an Element
3. Modify properties of an element
4. Move and rotate an element using transformation utility methods
5. 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: **4_ElementModification.vb (or .cs)**

- Command class name: **ElementModification**

Additional Considerations

We'll be using the following methods from the ElementFiltering class that we have written in the previous lab:

- ElementFiltering.FindFamilyType()
- ElementFiltering.FindElement()

1.2 Like we did in the previous labs, define member variables, e.g., m_rvtApp and m_rvtDoc, to keep DB level application and document respectively. e.g., :

```
<C#>
//  Element Modification - learn how to modify elements

[Transaction(TransactionMode.Manual)]
public class ElementModification : 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. Pick an Element

We have already learned how to pick an element in the Lab2. Once again, we can use one of overloaded PickObject() method to pick an object on the screen:

- UIDocument.Selection.PickObject(ObjectType.Element, promptString)

Here is a sample code:

```
<C#>
        // (1) pick an object on a screen.
        Reference refPick = rvtUIDoc.Selection.PickObject(
               ObjectType.Element, "Pick an element");
        Element elem = m_rvtDoc.GetElement(refPick);
</C#>
```

3. Modify Properties of an Element

In the lab 2, we have learned what constitute an element and what kind of information is accessible through the API. In this section, we will look at how to modify some of the information. Given an instance of a model element, we will modify:

- Properties of the given class
- Parameters
- Location Curve

3.1 Modify Family Type of an Instance

For information that is exposed as directly accessible as Class properties, such as Wall.WallType and FamilyInstance.Symbol, you can change it directly. Following shows an example of re-assigning a new family type to a given wall. Here we are using a method FindFamilyType() that we have defined in the previous lab:

```
<C#>
      // e.g., we assume we can only modify a wall
      Wall aWall = (Wall)elem;

      //  find a wall family type with the given name.
      Element newWallType = ElementFiltering.FindFamilyType(m_rvtDoc,
          typeof(WallType), "Basic Wall", "Exterior - Brick on CMU", null);

      //  assign a new family type.
      aWall.WallType = (WallType)newWallType;
</C#>
```

And here is an example with a door:

```
<C#>
    // e.g., an element we are given is a door.
    FamilyInstance aDoor = (FamilyInstance)elem;

    //  find a door family type with the given name.
    Element newDoorType = ElementFiltering.FindFamilyType(m_rvtDoc,
        typeof(FamilySymbol), "M_Single-Flush", "0762 x 2032mm",
        BuiltInCategory.OST_Doors);
```

```
    //  assign a new family type
    aDoor.Symbol = (FamilySymbol)newDoorType;
</C#>
```

A general way is using ChangeTypeId method:

```
<C#>
    // or use a general way: ChangeTypeId:
    aDoor.ChangeTypeId(newDoorType.Id);
</C#>
```

Below is a sample code with some surrounding support information. For simplicity, we assume we have a wall. For other kind of objects, you can apply the similar approach.

```
<C#>
    public void ModifyElementPropertiesWall(Element elem)
    {
      // Constant to this function.
      // this is for wall. e.g., "Basic Wall: Exterior - Brick on CMU"
      // you can modify this to fit your need.
      //
      const string wallFamilyName = "Basic Wall";
      const string wallTypeName = "Exterior - Brick on CMU";
      const string wallFamilyAndTypeName =
                      wallFamilyName + ": " + wallTypeName;

      // for simplicity, we assume we can only modify a wall
      if (!(elem is Wall))
      {
        TaskDialog.Show("Revit Intro Lab",
          "Sorry, I only know how to modify a wall. Please select a wall.");
        return;
      }
      Wall aWall = (Wall)elem;

      // keep the message to the user.
      string msg = "Wall changed: " + "\n" + "\n";

      // (1) change its family type to a different one.
      // (You can enhance this to import symbol if you want.)

      Element newWallType = ElementFiltering.FindFamilyType(m_rvtDoc,
          typeof(WallType), wallFamilyName, wallTypeName, null);

      if (newWallType != null)
      {
        aWall.WallType = (WallType)newWallType;
        msg = msg + "Wall type to: " + wallFamilyAndTypeName + "\n";
      }

      //...
```

```
    }
</C#>
```

3.2 Changing Parameters

To change the value of a parameter, you will first retrieve a parameter of your interest, then use “Set” method to modify the parameter with a new value. There are four overloaded methods. You can change the value of the follow data types:

- Set(ElementId)
- Set(Double)
- Set(Int32)
- Set(String value)

The following is a sample usage to change a wall’s “Top Offset” and “Comments” parameters:

```
<C#>
aWall.get_Parameter(BuiltInParameter.WALL_TOP_OFFSET).Set(14.0);
aWall.get_Parameter(BuiltInParameter.ALL_MODEL_INSTANCE_COMMENTS).Set(
     "Modified by API");
</C#>
```

The code below is an example (with some surrounding support information to show it in the command). It changes the values of the following parameters for a wall instance:

- Top Constraint to “Level 1” (Element Id)
- Top Offset to 5000.0 mm (Double)
- Structural Usage to Bearing(1) (Integer)
- Comments (String)

```
<C#>
      // (2) change its parameters.
      // as a way of exercise, let's constrain the top of the wall
      // to the level1 and set an offset.

      // find the level 1 using the helper function we defined in the lab3.
      Level level1 = (Level)ElementFiltering.FindElement(
          m_rvtDoc, typeof(Level), "Level 1", null);
      if (level1 != null)
      {
        aWall.get_Parameter(BuiltInParameter.WALL_HEIGHT_TYPE).
            Set(level1.Id);
        // Top Constraint
        msg += "Top Constraint to: Level 1" + "\n";
      }

      // Top Offset Double. hard coding for simplisity here.
      double topOffset = mmToFeet(5000.0);
      aWall.get_Parameter(BuiltInParameter.WALL_TOP_OFFSET).Set(topOffset);
```

```
      // Structural Usage = Bearing(1)
      aWall.get_Parameter(BuiltInParameter.WALL_STRUCTURAL_USAGE_PARAM).
          Set(1);

      // Comments - String
      aWall.get_Parameter(BuiltInParameter.ALL_MODEL_INSTANCE_COMMENTS).
          Set("Modified by API");

      msg += "Top Offset to: 5000.0" + "\n";
      msg += "Structural Usage to: Bearing" + "\n";
      msg += "Comments added: Modified by API" + "\n";
</C#>
```

where mmToFeet() is a simple function that converts unit from millimeter to feet.

☝ Note: Internally Revit keeps data in feet.

```
<C#>
    const double _mmToFeet = 0.0032808399;

    public static double mmToFeet(double mmValue)
    {
      return mmValue * _mmToFeet;
    }
</C#>
```

3.2 Changing Location Curve

To change the value of location information, you first retrieve location information from the given instance, and cast it to Location Curve from an instance. This allows you to access to the curve information. You then create a new line bound, and assign the new curve to the wall's location:

```
<C#>
      LocationCurve wallLocation = (LocationCurve)aWall.Location;

      // create a new line bound.
      XYZ newPt1 = new XYZ(0.0, 0.0, 0.0);
      XYZ newPt2 = new XYZ(20.0, 0.0, 0.0);
      Line newWallLine = Line.CreateBound(newPt1, newPt2);

      // finally change the curve.
      wallLocation.Curve = newWallLine;
</C#>
```

The code below shows an example of moving a wall by (-1000.0, 0., 0.) (with some surrounding support information to show it in the command):

```
<C#>
      // (3) Optional: change its location, using location curve
      // LocationCurve also has move and rotation methods.
```

```
        // Note: constaints affect the result.
        // Effect will be more visible with disjoined wall.
        // To test this, you may want to draw a single standing wall,
        // and run this command.

        LocationCurve wallLocation = (LocationCurve)aWall.Location;

        XYZ pt1 = wallLocation.Curve.GetEndPoint(0);
        XYZ pt2 = wallLocation.Curve.GetEndPoint(1);

        // hard coding the displacement value for simility here.
        double dt = mmToFeet(1000.0);
        XYZ newPt1 = new XYZ(pt1.X - dt, pt1.Y - dt, pt1.Z);
        XYZ newPt2 = new XYZ(pt2.X - dt, pt2.Y - dt, pt2.Z);

        // create a new line bound.
        Line newWallLine = Line.CreateBound(newPt1, newPt2);

        // finally change the curve.
        wallLocation.Curve = newWallLine;

        // message to the user.
        msg += "Location: start point moved -1000.0 in X-direction" + "\n";

        TaskDialog.Show("Revit Intro Lab", msg);
</C#>
```

LocationCurve also has Move and Rotation methods.

Exercise:

- Implement a function that takes an instance of element and modify its values such as family type, parameter values, and location information. (For the purpose of this exercise, you may assume a given element is a specific type of object, such as a wall or a door.)
- Call this function from the main Execute method with the element you have picked.

Note: Existing constrains may affect the result of these modifications. For example, if you have other walls joined at the ends, and you tried to modify the wall in a way that you violate the constraints, Revit will not allow you to do so. For the testing purposes, you may want to draw a single self-standing wall, and run the command.

4. Move and Rotate an Element Using Transform Utility Methods

Another way to modify elements is using transform utility methods. The utility class, `ElementTransformUtils`, offers following types of operations:

- Move
- Rotate
- Mirror
- Copy

RevitAPI.chm and Revit Developer Wiki includes sample code that shows the usages of some of these methods. Please refer to them for variation of various methods.

In our training labs, we will take a look at Move and Rotate as an example. The following is an example of move or translation of a given element by (10.0, 10.0, 0):

```
<C#>
      //  move by displacement
      XYZ v = new XYZ(10.0, 10.0, 0.0);
      ElementTransformUtils.MoveElement(m_rvtDoc, elem.Id, v);
</C#>
```

And here is an example of rotating a given element by 15 degree (= π/12) around Z-axis:

```
<C#>
     // try rotate: 15 degree around z-axis.
     XYZ pt1 = XYZ.Zero;
     XYZ pt2 = XYZ.BasisZ;
     Line axis = Line.CreateBound(pt1, pt2);
     ElementTransformUtils.RotateElement(m_rvtDoc, elem.Id, axis, Math.PI / 12.0);
</C#>
```

Following shows an example of a function that move and rotate a given element:

```
<C#>

    // A sampler function that demonstrates how to modify an element
    // through document methods.

    public void ModifyElementByTransformUtilsMethods(Element elem)
    {
      // keep the message to the user.
      string msg = "The element changed: " + "\n\n";

      // try move
      double dt = mmToFeet(1000.0);
      // hard cording for simplicity.
      XYZ v = new XYZ(dt, dt, 0.0);

      ElementTransformUtils.MoveElement(m_rvtDoc, elem.Id, v);

      msg = msg + "move by (1000, 1000, 0)" + "\n";


      // try rotate: 15 degree around z-axis.
      XYZ pt1 = XYZ.Zero;
      XYZ pt2 = XYZ.BasisZ;
      Line axis = Line.CreateBound(pt1, pt2);

      ElementTransformUtils.RotateElement(m_rvtDoc, elem.Id, axis, Math.PI/12.0);
```

```
      msg = msg + "rotate by 15 degree around Z-axis" + "\n";

      // message to the user.
      TaskDialog.Show("Modify element by utils methods", msg);
    }
</C#>
```

Regeneration of Graphics

One last note: when you have modified an element and that changes result in changes model geometry, and you need to access to the updated geometry, the graphics need to be regenerated. You can control this by calling Document.Regenerate() method.

```
        m_rvtDoc.Regenerate()
```

Exercise:

- Implement a function that takes an element, and move and rotate the element by some displacement and rotation value of your choice.

5. Summary

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

- Modify an element at element level by changing its properties, parameters and location
- Modify an element using Document level methods, such as Move and Rotate

In the next lab, we will take a look at how to create elements and build a model in the Revit.