**Revit API Intro Labs**

**Lab4 – Element Modification**

Created by M. Harada, July 2010  
Updated by DevTech AEC WG  
Last modified: 3/9/2015

**<VB.NET>**VB.NET Version**</VB.NET>**

**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:

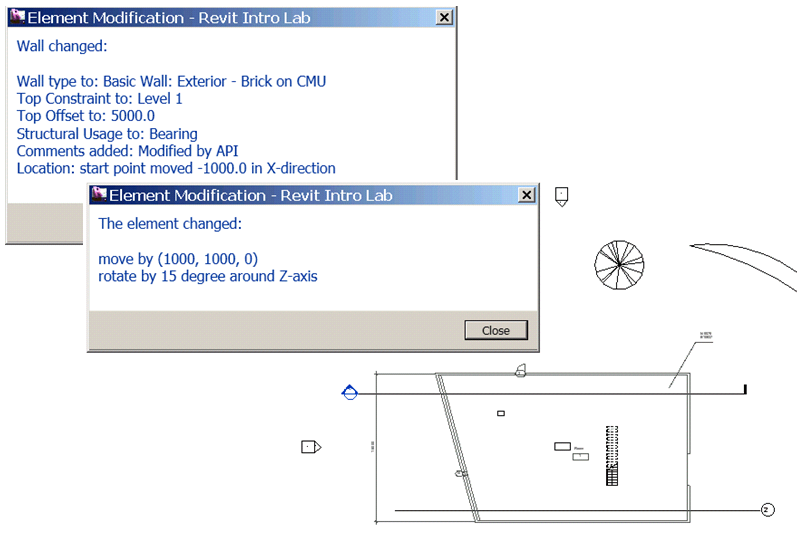


Figure 1. we’ll make modification to an element by changing its properties and   
by using transformation utility methods.

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
6. **Define A New External Command**

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

* 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. 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., :

**<VB.NET>**

'' Element Modification - learn how to modify elements

<Transaction(TransactionMode.Manual)> \_

Public Class ElementModification

Implements IExternalCommand

**'' member variables**

**Dim m\_rvtApp As Application**

**Dim m\_rvtDoc As Document**

Public Function Execute(

ByVal commandData As ExternalCommandData, \_

ByRef message As String, \_

ByVal elements As ElementSet) \_

As Result \_

Implements IExternalCommand.Execute

'' Get the access to the top most objects.

**Dim rvtUIApp As UIApplication = commandData.Application**

**Dim rvtUIDoc As UIDocument = rvtUIApp.ActiveUIDocument**

**m\_rvtApp = rvtUIApp.Application**

**m\_rvtDoc = rvtUIDoc.Document**

'' ...

Return Result.Succeeded

End Function

End Class  
**</VB.NET>**

1. **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:

**<VB.NET>** '' (1) pick an object on a screen.  
 Dim ref As Reference = rvtUIDoc.Selection.PickObject( \_   
 ObjectType.Element, "Pick an element")  
 Dim elem As Element = m\_rvtDoc.GetElement(ref)  
**</VB.NET>**

1. **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:

**<VB.NET>**

'' e.g., an element we are given is a wall.

Dim aWall As Wall = elem

'' find a wall family type with the given name.

Dim newWallType As Element = ElementFiltering.FindFamilyType( m\_rvtDoc, \_

GetType(WallType), "Basic Wall", "Exterior - Brick on CMU")

'' assign a new family type.

aWall.WallType = newWallType

**</VB.NET>**

And here is an example with a door:

**<VB.NET>**

'' e.g., an element we are given is a door.

Dim aDoor As FamilyInstance = elem

'' find a door family type with the given name.

Dim newDoorType As Element = ElementFiltering.FindFamilyType( m\_rvtDoc, \_

GetType(FamilySymbol), "M\_Single-Flush", "0762 x 2032mm", \_   
 BuiltInCategory.OST\_Doors)

'' assign a new family type.

aDoor.Symbol = newDoorType

**</VB.NET>**

A general way is using ChangeTypeId method:

**< VB.NET >**   
 // or use a general way: ChangeTypeId:

aDoor.ChangeTypeId(newDoorType.Id);

**</VB.NET >**

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.

**<VB.NET>**

Sub ModifyElementPropertiesWall(ByVal elem As Element)

'' 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 wallFamilyName As String = "Basic Wall"

Const wallTypeName As String = "Exterior - Brick on CMU"

Const wallFamilyAndTypeName As String =

wallFamilyName + ": " + wallTypeName

'' for simplicity, we assume we can only modify a wall

If Not (TypeOf elem Is Wall) Then

TaskDialog.Show("Revit Intro Lab", \_   
 "Sorry, I only know how to modify a wall. Please select a wall.")

Return

End If

''keep the message to the user.

Dim aWall As Wall = elem

Dim msg As String = "Wall changed: " + vbCr + vbCr

'' (1) change its family type to a different one.

'' To Do: change this to enhance import symbol later.

''

Dim newWallType As Element = \_

ElementFiltering.FindFamilyType(m\_rvtDoc, GetType(WallType), \_   
 wallFamilyName, wallTypeName)

If newWallType IsNot Nothing Then

aWall.WallType = newWallType  
 msg = msg + "Wall type to: " + wallFamilyAndTypeName + vbCr

End If

'' ...

End Sub

**</VB.NET>**

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 following 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:

**<VB.NET>**

aWall.Parameter(BuiltInParameter.WALL\_TOP\_OFFSET).Set(14.0)

aWall.Parameter(BuiltInParameter.ALL\_MODEL\_INSTANCE\_COMMENTS).Set( \_   
 "Modified by API")

**</VB.NET>**

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)

**<VB.NET>**

'' (2) change its parameters.

'' constrain top of the wall to the level1 and set an offset.

'' find the level 1 using the function we defined in lab3.

Dim level1 As Level = \_   
 ElementFiltering.FindElement(m\_rvtDoc, GetType(Level), "Level 1")

If level1 IsNot Nothing Then  
 '' Top Constraint

aWall.Parameter(BuiltInParameter.WALL\_HEIGHT\_TYPE).Set(level1.Id)

msg = msg + "Top Constraint to: Level 1" + vbCr

End If

Dim topOffset As Double = mmToFeet(5000.0) '' hard coding

'' Top Offset Double

aWall.Parameter(BuiltInParameter.WALL\_TOP\_OFFSET).Set(topOffset)

'' Structural Usage = Bearing(1)

aWall.Parameter(BuiltInParameter.WALL\_STRUCTURAL\_USAGE\_PARAM).Set(1)

'' Comments - String   
 aWall.Parameter(BuiltInParameter.ALL\_MODEL\_INSTANCE\_COMMENTS).Set( \_   
 "Modified by API")

msg += "Top Offset to: 5000.0" + vbCr

msg += "Structural Usage to: Bearing" + vbCr

msg += "Comments added: Modified by API" + vbCr

**</VB.NET>**

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

☝ Note: Internally Revit keeps data in feet.

**<VB.NET>**

Const \_mmToFeet As Double = 0.0032808399

Public Shared Function mmToFeet(ByVal mmValue As Double) As Double

Return mmValue \* \_mmToFeet

End Function

**</VB.NET>**

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:

**<VB.NET>**

Dim wallLocation As LocationCurve = aWall.Location

'' create a new line bound.

Dim newPt1 = New XYZ(0.0, 0.0, 0.0)

Dim newPt2 = New XYZ(20.0, 0.0, 0.0)

Dim newWallLine As Line = Line.CreateBound(newPt1, newPt2)

'' change the curve.

wallLocation.Curve = newWallLine

**</VB.NET>**

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):

**<VB.NET>**

'' (3) change its location, using location curve

'' get the location curve from a wall

Dim wallLocation As LocationCurve = aWall.Location

'' define a new location based on the current location

Dim pt1 As XYZ = wallLocation.Curve.GetEndPoint(0)

Dim pt2 As XYZ = wallLocation.Curve.GetEndPoint(1)

'' hard coding the displacement value for simility here.

Dim dt As Double = mmToFeet(1000.0)

Dim newPt1 = New XYZ(pt1.X - dt, pt1.Y - dt, pt1.Z)

Dim newPt2 = New XYZ(pt2.X - dt, pt2.Y - dt, pt2.Z)

'' create a new line bound.

Dim newWallLine As Line = \_   
 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" + vbCr

TaskDialog.Show("Revit Intro Lab", msg)

**</VB.NET>**

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.

1. **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](http://www.autodesk.com/revit-help/?guid=GUID-85E4A43E-88B5-43C6-908C-2D138C9F611D) 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):

**<VB.NET>**

'' move by displacement

Dim v As XYZ = New XYZ(10.0, 10.0, 0.0)

ElementTransformUtils.MoveElement(m\_rvtDoc, elem.Id, v)

**</VB.NET>**

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

**<VB.NET>**

'' rotate by 15 degree around z-axis.

Dim pt1 = XYZ.Zero

Dim pt2 = XYZ.BasisZ

Dim axis As Line = Line.CreateBound(pt1, pt2)

ElementTransformUtils.RotateElement(m\_rvtDoc, elem.Id, axis, Math.PI/12.0);

**</VB.NET>**

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

**<VB.NET>**

'' modify an element through transform utils methods, Move and Rotate

''

Sub ModifyElementByTransformUtilsMethods(ByVal elem As Element)

'' keep the message to the user.

Dim msg As String = "The element changed: " + vbCr + vbCr

'' try move

Dim dt As Double = mmToFeet(1000.0) '' hard cording for simplicity.

Dim v As XYZ = New XYZ(dt, dt, 0.0)

ElementTransformUtils.MoveElement(m\_rvtDoc, elem.Id, v)

msg = msg + "move by (1000, 1000, 0)" + vbCr

'' rotate: 15 degree around z-axis.

Dim pt1 = XYZ.Zero

Dim pt2 = XYZ.BasisZ

Dim axis As Line = Line.CreateBound(pt1, pt2)

ElementTransformUtils.RotateElement(m\_rvtDoc, elem.Id, axis,Math.PI/12.0)

msg = msg + "rotate by 15 degree around Z-axis" + vbCr

'' show the message to the user.

TaskDialog.Show("Revit Intro Lab", msg)

End Sub

**</VB.NET>**

**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.

1. **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.

Autodesk Developer Network