Revit API UI Labs

Lab 4 – Event

May, 2011 by A. Nagy

Last Updated, Date : March 03, 2018

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

Objective: In this lab, we will learn how to subscribe to events. We'll learn how to:

- Subscribe to a specific event
- Use dynamic update mechanism

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

1. Create a new External Application
2. Subscribe to an event
3. Implement a dynamic model updater
4. Summary

1. Create a new External Application

We'll add an external application to the current project.

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

- File name: **4_Event.vb (or .cs)**
- Application class name: **UIEventApp**

(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.)

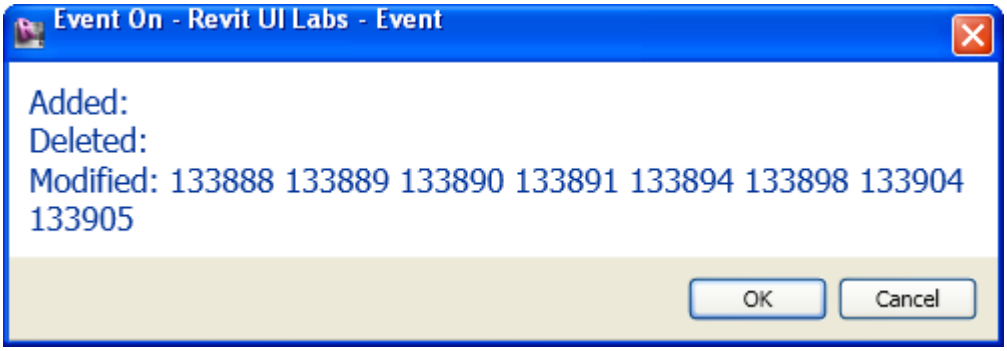
Required Namespaces:
Namespaces needed for this lab are:

- Autodesk.Revit.DB
- Autodesk.Revit.UI
- Autodesk.Revit.ApplicationServices
- Autodesk.Revit.Attributes
- Autodesk.Revit.UI.Selection (this is for selection)

- Autodesk.Revit.DB.Events (for event handling)

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.

2. Subscribe to an event



You can subscribe to various events inside Revit. Just take a look at the list of events in Visual Studio under the various API classes like Application, UIApplication, Document, etc
As an example we'll subscribe to the DocumentChanged event so whenever an object is added, deleted or modified inside Revit, we'll get notified about it
We need to have a function that has the same signature as the event

```
<C#>
    public void UILabs_DocumentChanged(
      object sender, DocumentChangedEventArgs args )
    {
    }
</C#>
```

Then we can assign this to the event using += (cs) or AddHandler (vb) inside the OnStartup() function

```
<C#>
    app.ControlledApplication.DocumentChanged += UILabs_DocumentChanged;
</C#>
```

Inside the DocumentChanged event handler list the information available through args about what changed in the document in a TaskDialog.
Then add a bool variable to the UIEventClass that will control if we are showing a dialog when the Revit document changed or not.

Also, implement 3 different external commands. One will set this bool variable to True, one will set it to false and one will toggle it.

3. Implement a dynamic model updater

The above event is very useful if you want to track changes – e.g. you have an outside database that needs to be updated with new information whenever the document changes.
However, if you want to modify certain parts of the document based on the changes that have just happened, then you'll need a different mechanism called dynamic model updater.
As an example let's create a dynamic model updater that keeps windows and doors always in the center of the hosting wall.
For this first we need to create a class that implements IUpdater and its 5 functions. Inside the class we also need to create an instance of UpdaterId based on the GUID of our Add-In plus another GUID – you can use Visual Studio to create the latter.

<C#>

```
  public class WindowDoorUpdater : IUpdater
  {
    // Unique id for this updater = addin GUID + GUID for this specific
    // updater.

    UpdaterId m_updaterId = null;

    // Flag to indicate if we want to perform an update

    public static bool m_updateActive = false;

    /// <summary>
    /// Constructor
    /// </summary>

    public WindowDoorUpdater(AddInId id)
    {
      m_updaterId = new UpdaterId( id,
        new Guid( "EF43510F-38CB-4980-844C-72174A674D56" ) );
    }

    /// <summary>
    /// This is the main function to do the actual job.
    /// For this exercise, we assume that we want to keep
    /// the door and window always at the center.
    /// </summary>

    public void Execute(UpdaterData data)
    {
      if( !m_updateActive ) return;

    }

    /// <summary>
```

```
    /// This will be shown when the updater is not loaded.
    /// </summary>

    public string GetAdditionalInformation()
    {
      return "Door/Window updater: keeps doors and windows at the center of
walls.";
    }

    /// <summary>
    /// Specify the order of executing updaters.
    /// </summary>

    public ChangePriority GetChangePriority()
    {
      return ChangePriority.DoorsOpeningsWindows;
    }

    /// <summary>
    /// Return updater id.
    /// </summary>

    public UpdaterId GetUpdaterId()
    {
      return m_updaterId;
    }

    /// <summary>
    /// User friendly name of the updater
    /// </summary>

    public string GetUpdaterName()
    {
      return "Window/Door Updater";
    }
  }
</C#>
```

Now create an instance of our class in the OnStartup() function and filter the monitored elements to Walls.

```
</C#>
      WindowDoorUpdater winDoorUpdater =
        new WindowDoorUpdater( app.ActiveAddInId );

      // ActiveAddInId is from addin menifest.
      // Register it

      UpdaterRegistry.RegisterUpdater( winDoorUpdater );

      // Tell which elements we are interested in being notified about.
      // We want to know when wall changes its length.

      ElementClassFilter wallFilter =
```

```
        new ElementClassFilter( typeof( Wall ) );
      UpdaterRegistry.AddTrigger( winDoorUpdater.GetUpdaterId(),
        wallFilter, Element.GetChangeTypeGeometry() );

</C#>
```

Try to implement the rest yourself. Create a function that retrieves the door or window of a given wall. For this you will need to use FilteredElementCollector to retrieve all the doors and windows from the database then check whose owner is the wall in question if any.

Then create another function that centers the door or window based on the location curve of the wall.

4. Summary

In this lab, we learned how to subscribe to events. We've learned how to:

- Subscribe to a specific event
- Use dynamic update mechanism