



Using Revit API Events to Manage Miracles

Don Rudder – CASE Design, Inc.

SD5399

Take a deep gander into various creative uses of the Revit API event functionality. Learn ways to monitor model health, activity, and happiness all while not having to ask your users to do anything to help you achieve it. If you don't write code, don't worry. There will be plenty for everyone in this lecture including C# source code, explanations of scenarios, as well as things to avoid and why you might want to avoid those things.

This class assumes that you already have a basic understanding of the .NET framework as well as the Revit API.

Learning Objectives

At the end of this class, you will be able to:

- Understand what Revit API events are and their limitations
- Know how to subscribe and unsubscribe to events in Revit
- See several examples on creative Revit API event uses
- Bypass some of the Revit API event rule limitations with *IUpdater*

About the Speaker

Don Rudder focuses on the development and management of specialized software and add-ins for various applications. 10+ of his 18 years of experience in the AEC industry has been served as an MEP designer and has served roles varying from CAD/BIM Manager to software developer. Self-taught in 20 programming languages and well versed in .NET, web-based AEC tools and pretty much any kind of Automation. He has also written the API chapters for a few volumes of the Autodesk Official Training Guide Series, Mastering Autodesk Revit Architecture as well as a book entitled 'Instant Autodesk Revit 2013 Customization with .NET How-to'.



@AYBABTM

d.rudder@case-inc.com

What are Revit API Events?

Revit API events provide a means for a custom application to subscribe to and receive notifications when something specific changes in the Revit environment. The custom application can then autonomously react to these notifications without user intervention. As you can imagine, Revit events can provide a quite powerful bit of functionality. With great power comes great responsibility.

Be Kind



Before we go any further, I just want to remind you that just because you can do something doesn't always mean that you should. Events do obviously consume system resources and depending on how intensely used can result in a bad experience for your users. I recommend using events sparingly and only for solving real business problems (LOL).

Revit API Events Defined

Revit API Events are obviously .NET events and .NET events are a form of delegate. In order to properly understand events, it is important to first understand delegates.

- Delegates are smart containers that manage references to methods, as opposed to containers that manage references to objects or properties.
- Delegates can hold reference to any number of methods, but not just any old method. They can only contain methods with signatures matching their own signature. A signature can be explained as the combination of inputs and outputs for a method.
- Think of a delegate as something that holds a collection of methods that will each get fired off when the event is "raised."

Perhaps a simple analogical explanation for a delegate would be similar to a classroom. A professor is explaining or teaching a subject and this information is being listened to by an unknown number of people or devices. This common set of information is then used to react to some kind of assignment. Nobody in the class knows about the assignment until the professor announces it. This announcement could be thought of as the "raising" of the event and then each student would react to it each in their own way. Some will pass, others will fail, but they all are notified in the same way with the same rules.

Just Because

If you enjoy stories about events and delegates as much as a good fairy tell, then you should read the extremely outdated but ever so entertaining: *.NET Delegates: A C# Bedtime Story*: <http://www.sellsbrothers.com/writing/delegates.htm>

How Revit Events Work

When an event fires off, or is “raised”, the delegate goes through the collection of registered custom event handler methods for the specific event and executes them one by one using its values as input parameters for the registered event handler methods. These methods all utilizing the same input and output structures (signature) make it possible to deal with each of these methods in a similar way between each subscription to that specific event.

Lifecycle

Events can persist throughout the life of a Revit session or can be started and stopped on demand. As a general housekeeping rule, it is recommended that you always unsubscribe from events when you are either done using them or at application shut down. For every subscription, there should be an equal unsubscription and should fire off no later than session shut down.

How Many Events Can I Subscribe to?

There is no documented limit to how many different events that you can use in an application or a limit to how many times an event can be subscribed to in the same Revit session. The same event can be subscribed to more than once from the same add-in as well as from numerous add-ins at the same time.



Known Limitations

The most important limitation to note about Revit API events is that it is generally not possible to make model modifications (like committing a transaction) from within an event handler method. The keyword “generally” should give away that there are exceptions to this limitation. The *Idling* event is one that breaks this rule and fully supports model changes. We will implement the *Idling* event in one of the code samples later in the class.

With the exception of the *Idling* event, there are a couple other means of getting around the limitation regarding model changes. We will see another way around the model change limitation when we talk about the *IUpdater* interface later in the class.

Available Revit API Events

You probably guessed by now that you won’t be deciding how events work within the Revit API, and will generally have to work with the events provided by Autodesk. The good news is that there are quite a few to choose from and with a little creativity can probably accomplish just about anything you would ever need.

Where Are They?

There are a few objects within the Revit API that provide events. Detailing out each and every event and how they all work is a bit beyond the scope of this class, but we will definitely get to see a few examples in the provided source examples to get a taste for what is available

and how to use them. We'll focus this class specifically on a few of the events available in the *Application* and *Document* objects.

A more complete list of each event and their accompanying documentation can be found in the main Revit SDK Documentation (chm file). A partial listing for the *Application* events as well as a complete listing for the *Document* events are shown below.

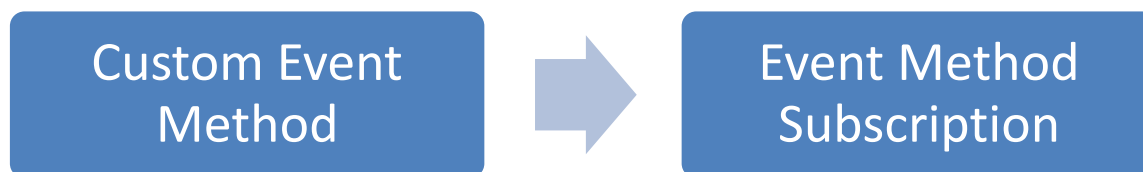
- [-] Application Class
 - [-] Application Members
 - [-] Application Constructor
 - [-] Application Methods
 - [-] Application Properties
 - [-] Application Events
 - [-] ApplicationInitialized Event
 - [-] DocumentChanged Event
 - [-] DocumentClosed Event
 - [-] DocumentClosing Event
 - [-] DocumentCreated Event
 - [-] DocumentCreating Event
 - [-] DocumentOpened Event
 - [-] DocumentOpening Event
 - [-] DocumentPrinted Event
 - [-] DocumentPrinting Event
 - [-] DocumentSaved Event
 - [-] DocumentSavedAs Event
 - [-] DocumentSaving Event

- [-] Document Class
 - [-] Document Members
 - [-] Document Methods
 - [-] Document Properties
 - [-] Document Events
 - [-] DocumentClosing Event
 - [-] DocumentPrinted Event
 - [-] DocumentPrinting Event
 - [-] DocumentSaved Event
 - [-] DocumentSavedAs Event
 - [-] DocumentSaving Event
 - [-] DocumentSavingAs Event
 - [-] ViewPrinted Event
 - [-] ViewPrinting Event

Subscribing and Unsubscribing to

Revit API Events

Subscribing to an event is where you make a request for Revit to add one of your custom methods to the notification queue, or delegate, for a specific event available from within the Revit API. Your custom method would then get called when the event gets fired off, or “raised.” You can handle one event type per custom event method. There are two basic parts to event subscription in Revit.



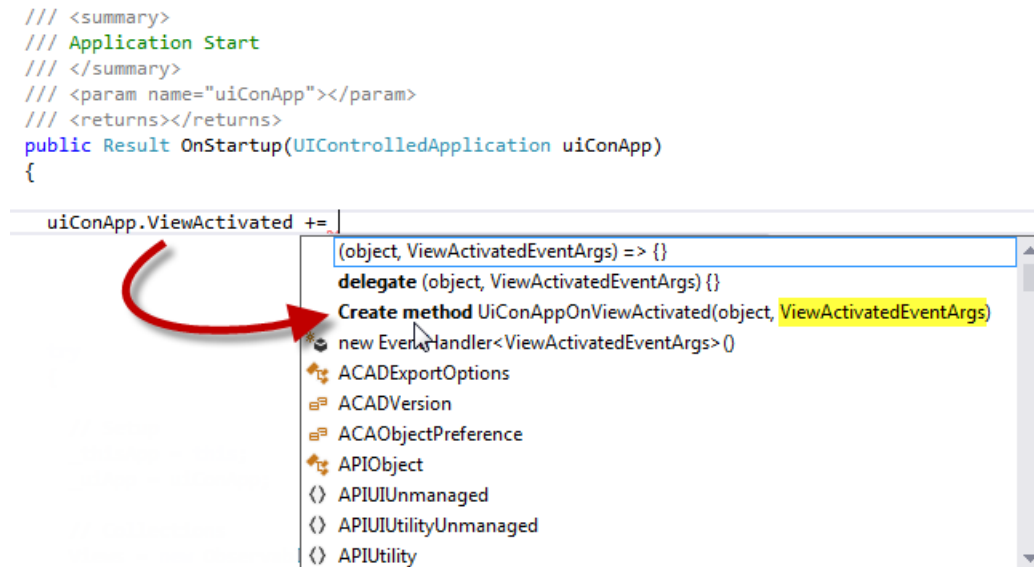
Custom Event Methods

Any custom method that you request to register as an event handler must match the signature for the specific event that you are subscribing to. This basically means that you cannot get creative with the input arguments for your event handler method. An example of this would be if you are subscribing to the *ViewActivated* event, you will be required to build your custom method with the input arguments for handling that exact event.

Once you've chosen an event that you want to implement, you'll need to identify the matching event argument from the Revit API to use in your custom method handler. The slower and more frustrating way to do this would be to research the Revit API SDK and manually type this all into your custom event method. The easier way is to use a Visual Studio extension such as Resharper (<https://www.jetbrains.com/resharper/>) to do this for you. Resharper is an extremely common extension and can be purchased for under \$250.

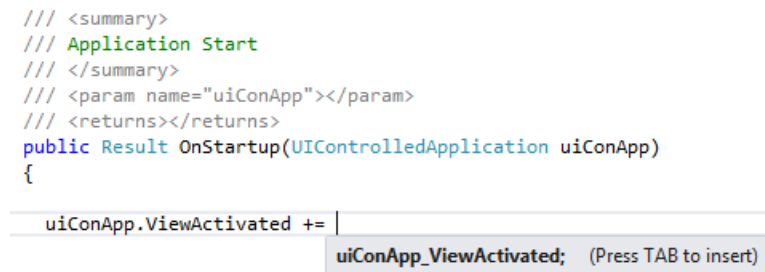
Resharper can really save you time and frustration when it comes to building event based applications. Resharper provides a feature that allows you to automatically build your custom event handler methods implementing all required input parameters with a single click as you are subscribing to an event. The illustrations below demonstrate two ways to subscribe to the *ViewActivated* event.

Subscribing to an Event with Resharper Installed



Subscribing to an Event (no custom Visual Studio extensions)

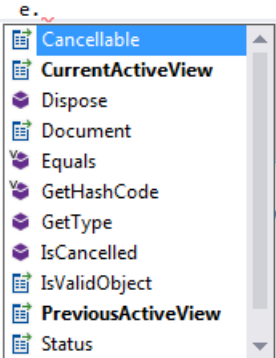
As you can see from the illustration, this way isn't as magical.



Sample Custom Event Handler Method for *ViewActivated*

The method name can be whatever you want, but the input argument types have to be of *object* and then *ViewActivatedEventArgs* as illustrated below. The names of the fields can also be whatever you want them to (sender and e). The available members for the *ViewActivatedEventArgs* object is also shown below via intellisense. This should look pretty familiar if you've ever implemented a button in a windows form click event.

```
/// <summary>
/// View activated
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
internal void OnViewActivated(object sender, ViewActivatedEventArgs e)
{
    e.
```



The image shows an Intellisense dropdown menu for the *e.* property. The menu lists the following members: *Cancellable*, *CurrentActiveView*, *Dispose*, *Document*, *Equals*, *GetHashCode*, *GetType*, *IsCancelled*, *IsValidObject*, *PreviousActiveView*, and *Status*. The *Cancellable* member is currently selected and highlighted in blue.

Enable or Disable an Event

The code snippet below illustrates how you would enable or subscribe to the *ViewActivated* event and your custom event handler method named *OnViewActivated*.

```
uiConApp.ViewActivated += OnViewActivated;
```

You would then unsubscribe or disable the same event with the following line of code.

```
uiConApp.ViewActivated -= OnViewActivated;
```

The only difference between turning an event on or off is the plus (+) sign to turn it on and the minus (-) sign to turn it off.

Examples

I promised I would provide a few creative examples capable in solving real-world problems. Refer to the Visual Studio solution for the full source code.

Log Time Spent in and Export an Image on View Change

Have you ever been curious as to how much time you spent in each view in Revit and wished you had a log to look through when filling out your time sheet? One of the more useful tricks that you can do with events is track exactly what view you were in and for how long.

Name	Kind	Status	
Level 1	FloorPlan	4 sec	
From Yard	ThreeD	2 sec	
North	Elevation	41 sec	
Title Sheet	DrawingSheet	3 sec	

The two sample methods below are event handler methods for the *ViewActivating* and *ViewActivated* events.

```

/// <summary>
/// Fires off each time a view is switched. This is
/// also a good way to know when a new model is open and
/// active in the editor.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
internal void OnViewActivating(object sender, ViewActivatingEventArgs e)
{
    if (e.Document == null) return;

    try
    {
        // End Active View
        if (ActiveViewTimer != null)
        {
            ActiveViewTimer.SetClosed();
            Views.Add(ActiveViewTimer);
            Views.SortBackwards(a => a.Opened);
            PageEventData.UpdatePage();
        }

        // Image Export?
        if (e.CurrentActiveView == null) return;
        if (ExportViewImageOnClose)
            ExportViewImage(false, e.CurrentActiveView.Name);
    }
    catch { }
}

```

```

/// <summary>
/// Fires off after a view has been activated.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
internal void OnViewActivated(object sender, ViewActivatedEventArgs e)
{
    try
    {
        ActiveDoc = e.Document;
        if (ActiveDoc == null) return;

        // New View Timer
        ActiveViewTimer = new clsViewTime(e.CurrentActiveView);

        PageEventData.LabelActiveViewName.Content = e.CurrentActiveView == null
            ? "No Active View"
            : string.Format("{0} {1}",
                e.CurrentActiveView.ViewType,
                e.CurrentActiveView.Name);

        // Image Export?
        if (e.CurrentActiveView == null) return;
        if (ExportViewImageOnOpen)
            ExportViewImage(true, e.CurrentActiveView.Name);
    }
    catch {}
}

```

Log Level, Door, & Wall Deletions (Who, When, etc.)

One of the most common workshared Revit model mysteries has probably got to be the loss of tags and previously input parameter values as a result of host families being deleted and redrawn/modeled rather than edited. It is all too easy for a beginner user to fall into this trap. Something as simple as it can be to delete and then remodel a ceiling or complex wall system (or a level! - OUCH) can result in a lot of lost work. The *OnDocumentChanged* sample will demonstrate how to log each of these elements (walls, doors, and levels) as they get deleted and who deleted them.

	SIP 202mm Wall - conc clad Element Id: 198694	11:40:18 AM d.rudder
	Wall - Timber Clad Element Id: 234869	11:40:18 AM d.rudder
	Level 2 Element Id: 245423	11:40:18 AM d.rudder
	SIP 202mm Wall - conc clad Element Id: 428588	11:40:18 AM d.rudder
	SIP 202mm Wall - conc clad Element Id: 428745	11:40:18 AM d.rudder

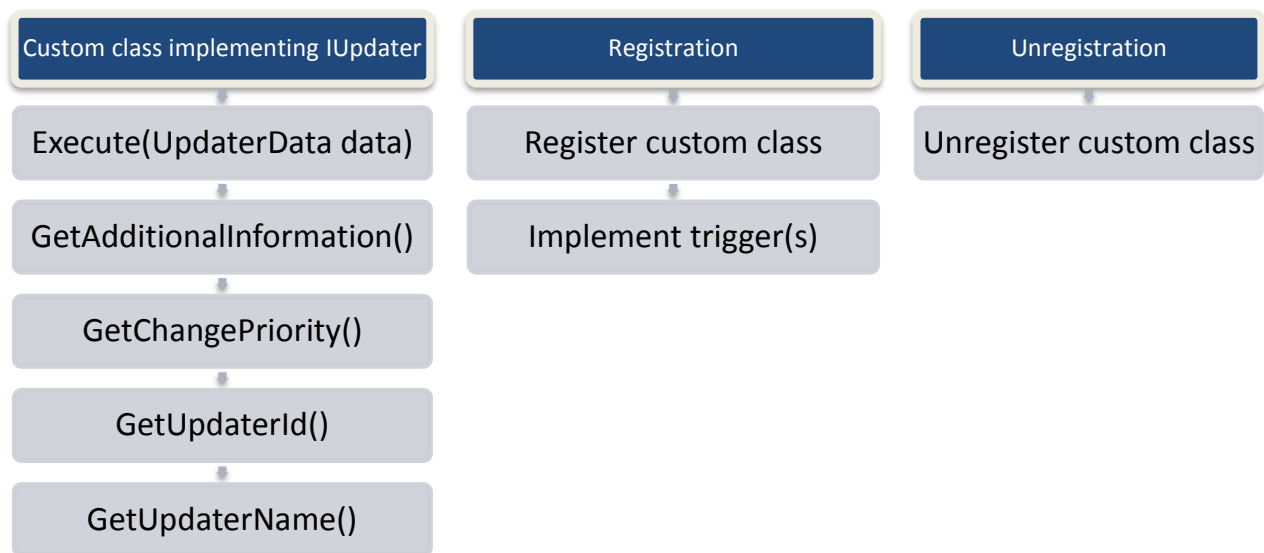
Add Missing Sheets from External File (no User Interaction)

The *Idling* event is definitely one of the most useful. The *OnIdling* event sample demonstrates how to break the model changes rules and should obviously be used with extreme care. Be very mindful when implementing this event if your implementation results in a model change.

Using IUpdater to Get Around Some of the Revit API Event Limitations

Perhaps one of the most powerful and often overlooked features in the Revit API is the *IUpdater*. *IUpdater* is a Revit API interface that provides a means to focus on specific changes that you are interested in and then react to these changes either in read-only mode or in a mode that changes the model.

There are three parts to consider when implementing an *IUpdater* solution. The interface itself must first be implemented. The second part is to register the updater and the last of course being the unregistration to terminate it. The diagram below shows these three concepts as columns with each major feature listed beneath.



Custom Class Implementing the IUpdater Interface

As you can see in the image above, the first column has quite a few items listed beneath it. Each of these must be addressed in your custom updater implementation. Creating a new custom class and implementing the *IUpdater* interface will generate these items for you.

<Record video>

Execute()

This method is the most important and where you provide the functionality that you want to execute for your updater. A method argument type of *UpdaterData* gets applied as an input and can be used to gain access to the changes in the model. The key methods made available from this method argument are:

- GetAddedElementIds() returns a full list of element ids for new elements.

- `GetDeletedElementIds()` returns a list of element ids for what was deleted. It should be noted that an element id that points to an element that no longer exists in the model cannot be used to gain access to any information about the element. You will have to utilize some other reference to those elements other than *`Document.GetElement()`*.
- `GetModifiedElementIds()` provides access to the ids for elements that have changed.

GetAdditionalInformation()

This is used to describe what the updater is intended to do when it is not loaded.

GetChangePriority()

This property allows you to set the priority for your updater so that it provides the functionality that you require. The available options are illustrated below (from the SDK).

Member name	Description
GridsLevelsReferencePlanes	Updater will modify core datums and major reference planes
Masses	Updater will modify masses
FloorsRoofsStructuralWalls	Updater will modify exterior shell of the building or floor slabs
Structure	Updater will modify structural elements like beams, braces, columns and trusses
InteriorWalls	Updater will modify interior/partition walls
DoorsOpeningsWindows	Updater will modify family instances that cut their hosts like doors and windows
MEPFixtures	Updater will modify MEP fixtures such as lights, sinks, equipment, etc.
RoomsSpacesZones	Updater will modify Rooms, MEP spaces or zones
MEPCalculations	Updater will update MEP calculated values
FreeStandingComponents	Updater will modify standalone components
Connections	Updater will modify connections
Rebar	Updater will modify rebar
MEPAccessoriesFittingsSegmentsWires	Updater will modify MEP elements such as segments, fittings and wires
MEPSystems	Updater will modify MEP systems or circuits
Views	Updater will modify views
DetailComponents	Updater will modify detail components and symbolic lines
Annotations	Updater will modify annotations

GetUpdaterId()

Returns the GUID that is used to address the updater.

GetUpdaterName()

Returns the name of your updater.

Registering IUpdater and Setting Triggers

An *IUpdater* (we'll just refer to these as updaters now) must be registered similar to an event, but there are a few differences. Updaters require that you supply a few extra items of information regarding what you are wanting them to notify on. Do you want to react to a parameter change? Maybe you want to react to a geometric change of a wall, or the insertion

of a specific type of door on a specific kind of wall. As you can see, you have a lot more options as to what you can ask Revit to notify you about with updaters than you do with traditional events.

The first thing that you will typically do when registering the updater is first instantiate a reference to one of your custom updater class objects. The code below shows a reference or *m_updX* being set to our *UpdaterXYZ* class.

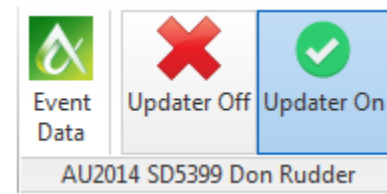
```

/// <summary>
/// Register an IUpdater
/// </summary>
private void RegisterUpdater(UIApplication uiApp)
{
    try
    {
        // Register
        UpdaterXYZ m_updX = new UpdaterXYZ(
            uiApp.ActiveAddInId,
            uiApp.ActiveUIDocument.Document,
            clsGlobals.UpdaterGuidX);
        UpdaterRegistry.RegisterUpdater(m_updX, true);

        // Add an all inclusive (heavy overhead) change trigger
        UpdaterRegistry.AddTrigger(
            m_updX.GetUpdaterId(),
            new ElementClassFilter(typeof(FamilyInstance)),
            Element.GetChangeTypeAny());

        // Add a new element trigger
        UpdaterRegistry.AddTrigger(
            m_updX.GetUpdaterId(),
            new ElementClassFilter(typeof(FamilyInstance)),
            Element.GetChangeTypeElementAddition());
    }
    catch (Exception ex)
    {
        using (TaskDialog td = new TaskDialog("Error"))
        {
            td.TitleAutoPrefix = false;
            td.MainInstruction = "Failed to Register Updater";
            td.MainContent = "Something failed while trying to register the IUpdater: \n\n" + ex.ToString();
            td.Show();
        }
    }
}

```



It should be noted that not all triggers are fired off during an undo or redo and most will not fire off for deleted or new element additions into the model. If you want to react to new or deleted elements, you will need to use specific triggers for those changes. You can still use the same updater class to handle these triggers, but you will just need to add these triggers separately so that your application is notified when elements get added or deleted.

Model Transactions

One of the things that *IUpdater* does not require in order to make changes to the model is a transaction. This might sound contradictory, but the explanation is simple. The changes that your updater makes is piggy-backed on top of the model change that triggered the change to start with. In our XYZ coordinate updater sample, a change to a family instance is necessary in order to fire off our updater reaction. The same model transaction that was used to move the element is used to update the comments parameter.



Unregistering an IUpdater

The process of unregistering an *IUpdater* can be done in a couple of different ways. If you have a reference to the updater, you can just use that and be done with it. There may, however, be situations where you want to disable an *IUpdater* that you did not create but believe is running inside your Revit session. The sample code provided for unregistration can be used in situations where you might not have an existing reference to the updater.

You can get the list of all registered updaters by calling *GetRegisteredUpdaterInfos()* from the *UpdaterRegistry* object. You can then iterate over the results looking for the one that you want to turn off.

```
// Query Registered Updaters
List<UpdaterInfo> m_uInfos =
    UpdaterRegistry.GetRegisteredUpdaterInfos().ToList();

foreach (UpdaterInfo x in m_uInfos)
{
    if (x.UpdaterName.ToLower() !=
        "au2014 sample xyz iupdater") continue;
    try
    {
        UpdaterId m_updater = new UpdaterId(
            uiApp.ActiveAddInId,
            new Guid(clsGlobals.UpdaterGuidX));
        UpdaterRegistry.UnregisterUpdater(m_updater);
    }
    catch {}
}
```

