Autodesk University 2012

# Let's face it: New Autodesk Revit 2013 UI API Functionality

Jeremy Tammik – Autodesk

**CP4107** We take a deeper look at the new user interface and add-in integration functionality provided by the Autodesk Revit 2013 API. We cover 2013 features including Options dialogue custom extensions using WPF components, subscription to Revit progress bar notifications, embedding and controlling a Revit view inside an add-in dialogue for preview purposes, the new drag and drop API, and the UIView. This class is complementary to and expands on **CP3272**, a Snapshot of the Revit UI API. Please note that prior .NET programming and Revit programming experience is required and that this class is not suitable for beginners.

## Learning Objectives

At the end of this class, you will have

- In depth understanding of the add-in integration possibilities offered by the Revit API.
- Add-in interaction with Revit using progress bar notifications, WPF option dialogue extensions, preview control, drag and drop, UIView, etc.
- Improved integration of Revit add-ins with the Revit user interface.
- Understanding and use of relevant AU and Revit SDK API samples.

## About the Speaker

Jeremy is a member of the AEC workgroup of the Autodesk Developer Network ADN team, providing developer support, training, conference presentations, and blogging on the Revit API.

He joined Autodesk in 1988 as the technology evangelist responsible for European developer support to lecture, consult, and support AutoCAD application developers in Europe, the United States, Australia, and Africa. He was a co-founder of ADGE, the AutoCAD Developer Group Europe, and a prolific author on AutoCAD application development. He left Autodesk in 1994 to work as an HVAC application developer, and then rejoined the company in 2005.

Jeremy graduated in mathematics and physics in Germany, worked as a teacher and translator, then as a C++ programmer on early GUI and multitasking projects. He is fluent in six European languages, vegetarian, has four kids, plays the flute, likes reading, travelling, theatre improvisation, carpentry, and loves mountains, oceans, sports, and especially climbing.

jeremy.tammik@eur.autodesk.com

# Table of Contents



## Introduction

The Revit 2013 API enhancements can be divided into the following main areas:

- Integration
- Analysis
- Modelling
- Interoperability
- Miscellaneous

This class focuses on the new add-in integration functionality[1]. Most of this material was also covered at the DevCamp conference in Boston in June 2012[2]. It follows up on the snapshot of the Autodesk Revit User Interface provided by Saikat Bhattacharya's class CP3272 and provides a deeper look at the following topics:

- Document management and View API
- Revit progress bar notifications
- Options dialogue WPF custom extensions
- Embedding and controlling a Revit view
- UIView and Windows coordinates
- Drag and drop

## Sample Add-ins

We will be looking at the following custom AU and standard Revit SDK sample applications to help understand and explore these topics in more depth:

- RevitUiApiNews AU sample implementing the following minimal simplified commands:
  - CmdAddOptionsTab
  - CmdDragDropApi
  - CmdPreviewControlSimple
  - CmdProgressWatcher
  - CmdUiView
- WinTooltip
  - Custom tooltip, UIView, Windows coordinates, Idling event, ReferenceIntersector
- ProgressNotifier SDK sample
  - Subscribe to progress bar events

[1] http://thebuildingcoder.typepad.com/blog/2012/03/revit-2013-and-its-api.html#2
[2] http://thebuildingcoder.typepad.com/blog/2012/06/devcamp-day-two.html#2

- UIAPI SDK sample
  - Add customised tabs to Options dialogue
  - Revit preview control
  - Drag and drop

## Document Management and View API

The new OpenAndActivateDocument method enables opening and activating a project file. The UIDocument.ActiveView property now enables changing the view within an open document.

The UIView class provides access to the user interface view and its Windows location and device coordinates. The GetOpenUIViews method lists all open views for a UI document. We take a closer look at this topic below.

The Revit 2013 API also provides much more complete access to view settings and creation functionality, a topic covered in more depth by Steven Mycynek in his class **CP3133** on using the schedule and view APIs. I'll just mention the bare essentials here:

View properties to read and write detail level, discipline, display style are now provided, e.g.

- ViewDetailLevel: coarse, medium, fine, etc.
- ViewDiscipline: architectural, coordination, electrical, mechanical, plumbing, structural
- DisplayStyle: Wireframe, HLR, Shading, ShadingWithEdges, Rendering, Realistic , FlatColors, RealisticWithEdges, Raytrace

The view creation functionality is provided by static Create methods on all View classes enabling the creation of new schedule, plan[3], section[4], 3D isometric and perspective views. The view family type element id is used to define the detailed view properties.

## Progress Bar Notifications

To receive progress bar notifications providing information about the current Revit progress bar state, an add-in subscribes to the ProgressChanged event. The event handler receives a ProgressChangedEventArgs instance providing the following information:

- Caption - progress bar caption describing operation in progress
- Stage - current stage of the progress bar
- Started, CaptionChanged, RangeChanged, PositionChanged, UserCancelled, Finished
- LowerRange - lower limit of range, always zero
- UpperRange - upper limit of range, any non-zero number
- Position - value in [0, UpperRange] incremented with 'PositionChanged' stage

It is not immediately obvious what kind of information this system actually provides access to. To find out, it may well be worthwhile installing a simple add-in subscribing to this event, logging the data received, and analyzing the output in various situations of interest to your specific application.

The CmdProgressWatcher external command in the RevitUiApiNews AU sample illustrates the whole procedure with a minimum of fuss. Every time the command is launched, it toggles the event subscription on or off:

```
  if( _watching )
  {
    app.ProgressChanged
      -= new EventHandler<ProgressChangedEventArgs>(
```

[3] http://thebuildingcoder.typepad.com/blog/2012/06/create-structural-plan-view.html
[4] http://thebuildingcoder.typepad.com/blog/2012/06/create-section-view-parallel-to-wall.html

```
        OnProgressChanged );
  }
  else
  {
    app.ProgressChanged
      += new EventHandler<ProgressChangedEventArgs>(
        OnProgressChanged );
  }

  Debug.Print( "\r\nProgress Watcher {0}\r\n",
    ( _watching ? "ended" : "begin" ) );

  _watching = !_watching;
```

The event handler simply logs the data it receives to the Visual Studio debug output window:

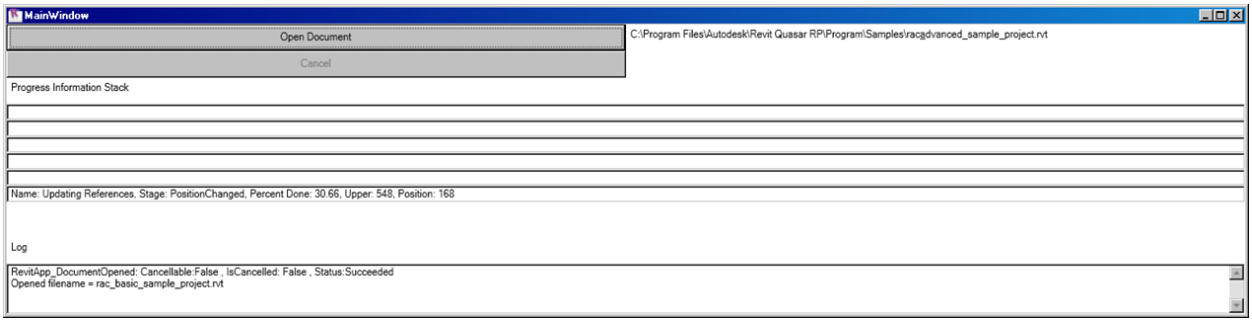
```
void OnProgressChanged(
  object sender,
  ProgressChangedEventArgs e )
{
  double percent = 100.0 * e.Position / e.UpperRange;

  Debug.Print(
    "'{0}' stage {1} position {2} [{3}, {4}] ({5}%)",
    e.Caption, e.Stage, e.Position, e.LowerRange,
    e.UpperRange, percent.ToString( "0.##" ) );
}
```

Here is sample log from opening a project showing that the events may be raised in a somewhat unpredictable sequence and require a certain amount of flexibility in their interpretation:

```
' ' stage Finished position 0 [0, 2147483647] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage Started position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 50 [0, 100] (50%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 50 [0, 100] (50%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 75 [0, 100] (75%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 75 [0, 100] (75%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 50 [0, 100] (50%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 50 [0, 100] (50%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 50 [0, 100] (50%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 75 [0, 100] (75%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 75 [0, 100] (75%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 75 [0, 100] (75%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage Finished position 75 [0, 100] (75%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage Started position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 25 [0, 100] (25%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 25 [0, 100] (25%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 25 [0, 100] (25%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage Finished position 25 [0, 100] (25%)
' ' stage Started position 0 [0, 2147483647] (0%)
' ' stage Finished position 0 [0, 2147483647] (0%)
' ' stage Started position 0 [0, 2147483647] (0%)
' ' stage Finished position 0 [0, 2147483647] (0%)
' ' stage Started position 0 [0, 2147483647] (0%)
' ' stage Finished position 0 [0, 2147483647] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage Started position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 0 [0, 100] (0%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 25 [0, 100] (25%)
```

```
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 25 [0, 100] (25%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage PositionChanged position 25 [0, 100] (25%)
'Drawing: floor_slab.rvt - 3D View: {3D}' stage Finished position 25 [0, 100] (25%)
```

The Revit SDK includes a more complete sample named ProgressNotifier. It displays progress information in a stack data structure, demonstrating that multiple nested levels of progress notification may be simultaneously active due to the nested transactions and sub-transactions employed by the driving processes.




## Options Dialogue Custom WPF Extensions

An add-in can add its own custom tabs to the standard Revit Options dialogue.

This is achieved by defining a WPF control to display, and subscribing to the UIApplication DisplayingOptionsDialog event.

When the dialogue is being displayed, the event handler is called and receives a DisplayingOptionsDialogEventArgs instance. It provides the PagesCount property giving the current number of tabs, including default Revit ones, and the AddTab method. The latter can be called to add a new tab. It takes the tab name and handler information as input. The tab handler information is encapsulated in an instance of the TabbedDialogExtension class.

Its constructor takes two arguments specifying the WPF user control instance and an OK handler. Once instantiated, these cannot be changed. It provides methods to get and set contextual help, properties to read the immutable WPF control and OK handler, and properties to get and set the cancel and 'restore defaults' handlers:

- Control – get the control
- OnOKAction – get the OK handler
- OnCancelAction – get and set the cancel handler
- OnRestoreDefaultsAction – get and set the restore defaults handler

The use is demonstrated with a minimum of fuss by the AU custom tab sample command CmdAddOptionsTab. In its Execute method, it subscribes to the event:

```
  uiapp.DisplayingOptionsDialog
    += new EventHandler<DisplayingOptionsDialogEventArgs>(
      OnDisplayingOptionsDialog );
```

The event handler instantiates a new WPF user control and adds the custom tab:

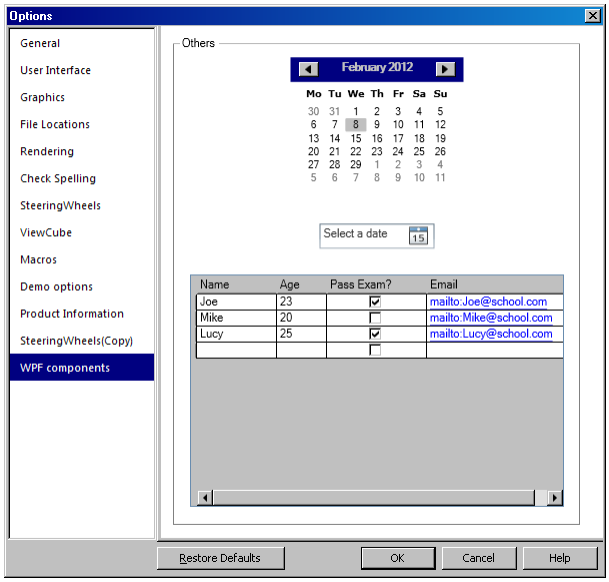
```
void OnDisplayingOptionsDialog(
  object sender,
  DisplayingOptionsDialogEventArgs e )
{
  try
  {
    UserControl1 c = new UserControl1(
      "Jeremy's User Control" );
```

```
    e.AddTab( "Jeremy's Custom Tab",
      new TabbedDialogExtension(
        c, c.OnOK ) );
  }
  catch( Autodesk.Revit.Exceptions
    .InternalException ex )
  {
    Debug.Assert( ex.Message.Equals(
      "Cannot duplicate the existing resource id." ),
      "expected duplicate resource id" );
  }
}
```

The control added is also minimal, presenting one single button. When clicked, a message is displayed.

If a given tab is activated during the user interaction with the Options dialogue, its OK, cancel and 'restore defaults' handlers are called, corresponding to the user actions.

The Revit SDK UIAPI sample demonstrates other examples of adding custom WPF Options dialogue tabs to display a number of controls and handle specific add-in settings.




## Embedding a Revit Preview Control

An add-in can embed and control a Revit view within its own form, dialogue or window.

This is accomplished using the new PreviewControl class. It presents a preview control to browse the Revit model.

Its constructor takes two input arguments, a document and a view id. These are in fact the only way in which the add-in controls what to display. All further interaction happens between the control and the user.

The view can be any graphical view, i.e. must be printable. Perspective views are also supported, and all 3D views can be manipulated using the view cube. The visibility and graphical settings of the view are effective and respected by the control. Standard zoom and pan commands are in place. The hosting form or window must be modal.

To use a Revit preview control, you create a standard .NET form and insert a WPF host in it. A suitable host is the System.Windows.Forms.Integration.ElementHost, which is populated with a newly instantiated Revit preview control instance.

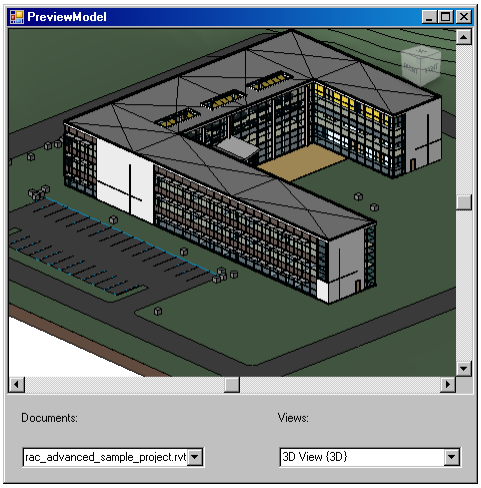
The preview control instance must be disposed of after use. The easiest way to achieve that is to encapsulate it in a 'using' statement.

The RevitUiApiNews AU sample implements the command CmdPreviewControlSimple which demonstrates this in the simplest possible manner. It creates and displays a modal form and populates it with a preview control displaying the currently active Revit view:

```
  void DisplayRevitView(
    Document doc,
    View view,
    IWin32Window owner )
  {

    using( PreviewControl pc
      = new PreviewControl( doc, view.Id ) )
    {
      using( System.Windows.Forms.Form form
        = new System.Windows.Forms.Form() )
      {
        ElementHost elementHost = new ElementHost();

        elementHost.Location
          = new System.Drawing.Point( 0, 0 );

        elementHost.Dock = DockStyle.Fill;
        elementHost.TabIndex = 0;
        elementHost.Parent = form;
        elementHost.Child = pc;

        form.Text = _caption_prefix + view.Name;
        form.Controls.Add( elementHost );
        form.Size = new Size( 400, 400 );
        form.ShowDialog( owner );
      }
    }
  }

  public Result Execute(
    ExternalCommandData commandData,
    ref string message,
    ElementSet elements )
  {
    IWin32Window revit_window
      = new JtWindowHandle(
        ComponentManager.ApplicationWindow );

    UIApplication uiapp = commandData.Application;
    UIDocument uidoc = uiapp.ActiveUIDocument;
    Document doc = uidoc.Document;

    // Determine all printable views that can be
    // used to populate the preview control; not
    // used in this sample, we just display the
    // active view:

    IEnumerable<View> views
```

```
        = new FilteredElementCollector( doc )
          .OfClass( typeof( View ) )
          .Cast<View>()
          .Where<View>( v => v.CanBePrinted );

      View view = doc.ActiveView;

      DisplayRevitView( doc, view, revit_window );

      return Result.Succeeded;
    }
```

The UIAPI SDK sample proves a slightly more powerful and complex demonstration of this. It first allows interactive selection of any of the currently open documents, the option to open a new one, and presents a list of all the selected document views to pick from. Each time any of these two selections is changed, the current preview control is discarded and a new one is instantiated with the updated input arguments.




## Drag and Drop API

Access to the drag and drop API is provided by two overloads of new static UIApplication DoDragDrop method. One of them takes a collection of strings as its single argument and initiates a standard built-in Revit drag and drop operation, treating the strings as file names. The second takes two arguments, an arbitrary .NET object and a drop handler method. It initiates a drag and drop operation with a custom drop implementation:

- DoDragDrop( ICollection<string> )
- DoDragDrop( object, IDropHandler )

The drop handler implements the IDropHandler interface. This just requires one single method, Execute, to handle the drop event for the data passed in. The data is the object provided as a first argument to the DoDragDrop method call.

This method is designed for use in a modeless form, and is thus the one and only Revit API call so far not requiring valid Revit API context. This method can be called from your modeless form at any time, even without Revit giving you explicit permission to interact with the API via some form of call-back or notification, as required for all other Revit API calls.

Drag and drop of a list of files is not new. For the sake of completeness, here is an overview of the standard Revit behaviour on receiving such a list, depending on the type of files being dropped:

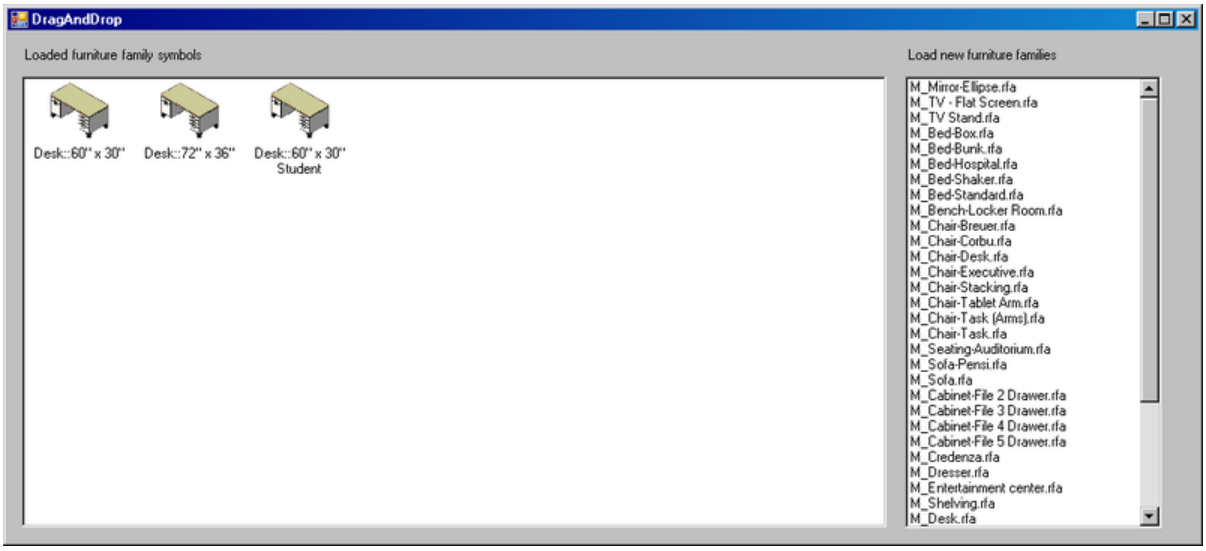
- One AutoCAD format or image file: a new import placement editor is started to import file

- More than one AutoCAD format or image file: a new import placement editor is started for first file
- One family file: the family is loaded, and an editor started to place instance
- More than one family file: all families are loaded
- More than one family file including other format files: Revit tries to open all files
- Valid file or list of files: Revit uses them appropriately
- Any file is not usable: failure is signalled to user, no exception is thrown, so add-in is not notified

Initiating a new drag and drop operation is as simple as calling one of the two DoDragDrop method overloads for either the Revit behaviour or providing a custom handler. In the Windows user interface, a drag can be started by picking some objects and dragging them with the mouse. This situation can be detected and handled by the MouseMove event, which in turn can invoke the DoDragDrop method.

The RevitUiApiNews AU sample implements an external command CmdDragDropApi demonstrating a custom drag and drop handler. It prompts for the selection of a family document, opens it, and displays a list of the symbols it contains. The user can pick a symbol and drag it onto Revit to initiate the standard instance placement via the PromptForFamilyInstancePlacement method.

The UIAPI SDK sample also provides a drag and drop command demonstrating both the standard Revit behaviour as well as a custom drop handler. It displays the following modeless dialogue[5]:




It presents a list of loaded furniture family symbols on the left and all furniture family files found by recursively searching the content library on the right.

Dragging a file name from the list on the right onto Revit will trigger the same built-in default behaviour as dragging the same file from the Windows explorer, i.e. load the family and start an editor to place a family instance.

Dragging a family symbol from the left side of the form is more exciting. In this sample, the MouseMove method determines the selected family symbol element id and passes that to DoDragDrop together with a new handler instance. The drop handler uses the id to open the symbol in the Execute method and invoke the PromptForFamilyInstancePlacement method on it.

[5] http://thebuildingcoder.typepad.com/blog/2012/04/drag-and-drop-api.html

## UIView and Windows Coordinates

The UIView class represents a view window and enables you to pan, zoom and determine its size in Windows coordinates. This is especially exciting, since for the first time ever, it enables an add-in to translate between Revit model and Windows device coordinates.

The UIView class provides three methods of interest:

- GetWindowRectangle: return the rectangle containing the coordinates of the view's drawing area in Windows coordinates.
- GetZoomCorners: return the corners of the view's rectangle in Revit model coordinates.
- ZoomAndCenterRectangle: zoom and centre the view to a specified rectangle.

To access a UIView instance, one can use the UIDocument GetOpenUIViews method, which returns a list of them all. One can use the UIView ViewId property to determine the one corresponding to a given document view.

The RevitUiApiNews AU sample implements an external command CmdUiView demonstrating this and exercising all the methods in a simple fashion[6]:

- Get the active view.
- Get all UIView instances.
- Determine the UIView for the active view.
- Query and report its Windows and Revit coordinates.
- Calculate new Revit coordinates to zoom in by 10%.
- Call ZoomAndCenterRectangle to do so.

WinTooltip provides a more complex sample demonstrating a useful application of the translation possibility between Windows and Revit coordinates to implement a custom tooltip displaying live BIM information relevant to the element the cursor is hovering over[7]. It uses the Idling event to obtain a valid Revit API context in which to query and display the information based on the modeless cursor tracking information. It also provides an example of using the ReferenceIntersector ray casting utility class.

## Learning More

Here are a few places to go to find background material and further reading:

- Revit Developer Center: DevTV and My First Plugin introductions, SDK, Samples, API Help
  http://www.autodesk.com/developrevit
- Developer Guide and Online Help
  http://www.autodesk.com/revitapi-wikihelp
- Revit API Trainings, Webcasts and Archives
  http://www.autodesk.com/apitraining > Courses, Schedule, Webcast Archive > Revit API
- Discussion Group
  http://discussion.autodesk.com > Revit Architecture > Revit API
- ADN AEC DevBlog
  http://adndevblog.typepad.com/AEC
- The Building Coder, Jeremy Tammik's Revit API Blog
  http://thebuildingcoder.typepad.com

[6] http://thebuildingcoder.typepad.com/blog/2012/06/uiview-and-windows-device-coordinates.html
[7] http://thebuildingcoder.typepad.com/blog/2012/10/uiview-windows-coordinates-referenceintersector-and-my-own-tooltip.html

- ADN, The Autodesk Developer Network
  http://www.autodesk.com/joinadn and http://www.autodesk.com/adnopen
- DevHelp Online for ADN members
  http://adn.autodesk.com
- UIView
  http://thebuildingcoder.typepad.com/blog/2012/06/uiview-and-windows-device-coordinates.html
  http://thebuildingcoder.typepad.com/blog/2012/10/uiview-windows-coordinates-referenceintersector-and-my-own-tooltip.html
- Drag and drop
  http://thebuildingcoder.typepad.com/blog/2012/04/drag-and-drop-api.html
- View creation
  http://thebuildingcoder.typepad.com/blog/2012/05/change-section-view-type-and-hide-cut-line.html
  http://thebuildingcoder.typepad.com/blog/2012/06/create-structural-plan-view.html
  http://thebuildingcoder.typepad.com/blog/2012/08/titbits-of-the-week.html
  http://thebuildingcoder.typepad.com/blog/2012/09/parts-assemblies-partutils-and-divideparts.html