Snapshot of the Autodesk® Revit® User Interface API

Saikat Bhattacharya – Autodesk

CP3272 Have you wondered about all the parts of the Revit User Interface that can be customized to reflect you or your company's needs? How can a plug-in be seamlessly integrated with the Revit UI - extending beyond the triggering of a command using the External Tools drop down list? This class aims to take a "snapshot" of the Revit user interface, providing an overview of all the customization possibilities over UI. We shall cover the customization with ribbons, Quick Access Toolbar, task dialogs, replacing Revit commands with custom commands, etc. using the Revit 2013 API. After attending this class, you should have a fair understanding of the possibilities of customizing Revit user interface and be able to better integrate your plug-ins in terms of the look and feel of Revit.

Learning Objectives

At the end of this class, you will

- Have a complete overview of the UI customization with Revit API.
- Be able to work with the Ribbon API, status bars, task dialogs.
- Know about Revit 2013 API including contextual help, replacing Revit commands implementation, discipline controls.
- Have some additional inputs on commonly asked UI customization requirements.

About the Speaker

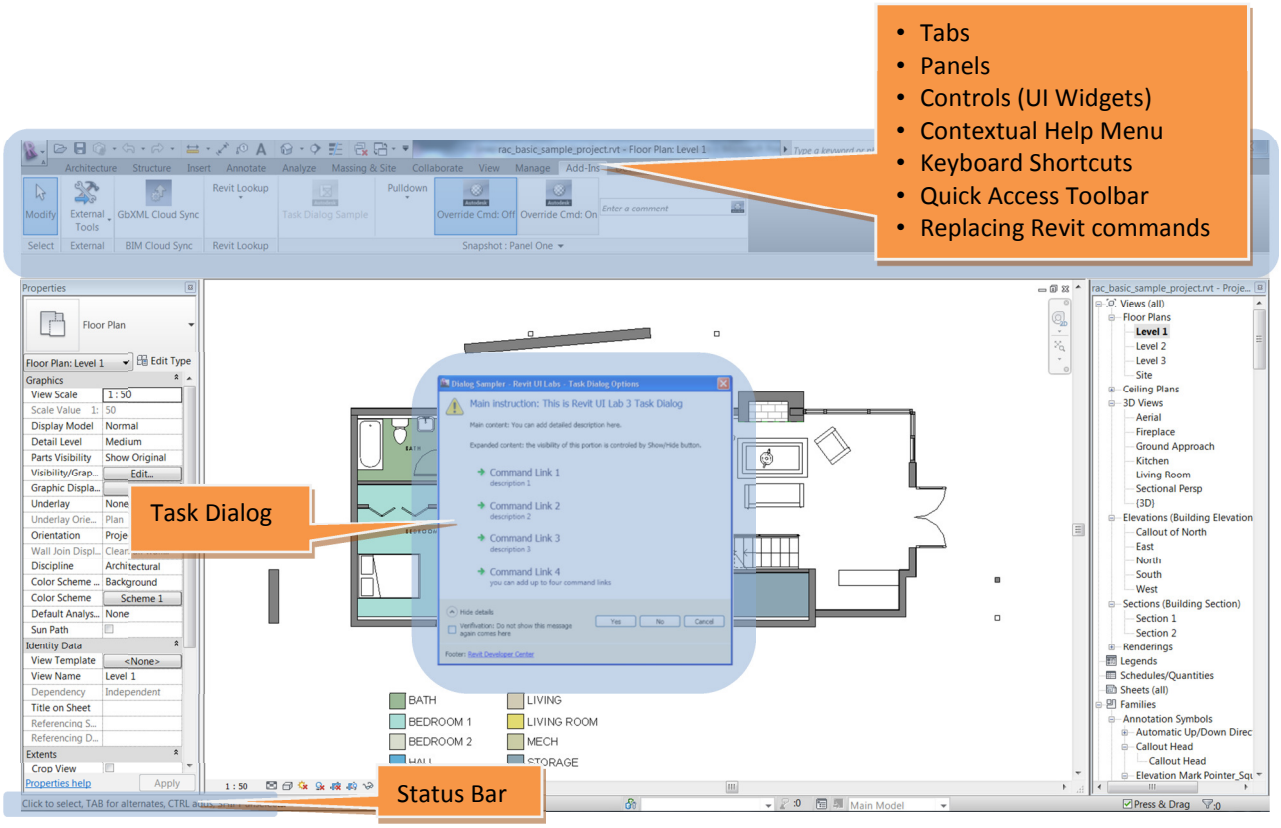
Saikat is a member of the AEC workgroup of the Autodesk Developer Network (ADN) team, providing evangelism, support, training, and delivering technical presentations to third party developers. He joined Autodesk in 2004 as Technical Consultant with the Autodesk Consulting group and then moved to the ADN team. Saikat has prior experience as GIS software developer and as project architect in the construction industry. He holds a Bachelor's degree in Architecture from India and a Master of Science degree from RPI.

saikat.bhattacharya@autodesk.com

Introduction

Revit provides APIs to enable seamless integration of third party add-ins with Revit's user interface (UI) and provide consistent user experience with Revit functionality. With each new release, the existing UI APIs have been enhanced and new UI APIs introduced – all with the goal of making add-ins look, feel and behave just like Revit user interface and functionality therein.

This class aims to take a "snapshot" of the Revit user interface and provide a complete overview of all the customization possibilities over UI. The following snapshot of Revit 2013 lists all the topics that will be covered in this class.



This class along with the following class by Jeremy Tammik should provide a complete in-depth understanding of the Revit user interface customisation possibilities:

CP4107 - Let's Face It: New Autodesk® Revit® 2013 User Interface API Functionality

Jeremy's class will cover the following remaining topics in the Revit UI API:

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

Revit UI API

Access to the Revit API is provided by two .NET assemblies, RevitAPI.dll and RevitAPIUI.dll. RevitAPI.dll contains methods used to access Revit's application, documents, elements, and parameters at the database level. RevitAPIUI.dll, as the name suggests, provides the UI API functionality and contains all API interfaces related to manipulation and customization of the Revit user interface, including Ribbon API, Task Dialogs, User Selection and status bar, besides the External Command and External Application interfaces.

Ribbon API

Revit provides the ability to create custom tabs, ribbons and controls on the Revit user interface using the Ribbon API. This API is the Graphical User Interface (GUI) customization API.

Revit 2010 had the first new look with a ribbon based UI and exposed some of the basic Ribbon APIs. Since then there has been enhancements to the Ribbon UI API in every release.

The current Ribbon API enables API users to create custom tabs, ribbon panels, add various types of UI controls/widgets on the panel, control visibility of the ribbon panels and widgets, add contextual (or F1) help, etc. This API is very easy to use. It does not require learning or knowledge of Windows Presentation Format (WPF) to be able to use this API or to work with Revit's user interface – unlike some of the other Autodesk Product APIs.

In addition to the programming aspect, there are guidelines for designing custom icons. This is to ensure that third party developer applications are nicely integrated with Revit UI. The documentation that is contained in the SDK is the Autodesk Icon Guidelines.pdf. The Ribbon Guidelines topic in the API User Interface Guidelines appendix also contains relevant information on developing a user interface that is compliant with the standards used by Autodesk.

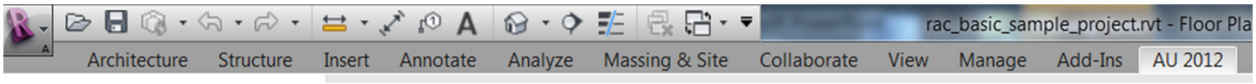
As mentioned in the previous section, the Ribbon API is accessible from the RevitUIAPI dll and is part of the Autodesk.Revit.UI namespace. The Ribbon API is used with the IExternalApplication interface.

Ribbon Tab

The current API supports creating new custom ribbon tabs. This comes in handy when an API user has an extensive list of user interface controls for a specific app, in which case all the widgets for the single app can be placed on different ribbon panels in a separate tab. Or if there are a couple of apps which are offered by a specific company, it can be organized better by having a separate company specific tab with the company name. The ribbon tab can be created using UIApplication.CreateRibbonTab(*tabname*) method.

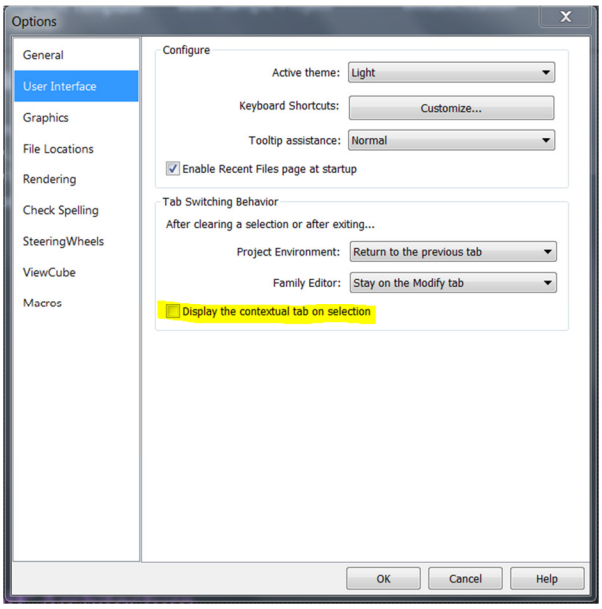
Since Revit's UI real estate is limited, the maximum number of tabs that can be created is 20 - after which it will throw exception of *InvalidOperationException* type.

By default, all user interface controls created by External Applications, are placed on the Add-Ins Tab. These UI items can be placed in the *Analyze* tab and custom tabs as well.
Note: If there are no external applications or commands registered via the .addin manifest file, the Add-Ins tab will not show up on the Revit user interface.



Persisting Ribbon Tab/Locking Ribbon Tab

When Revit users select an element from the user interface, by default, Revit switches the tab to make the selected element's specific contextual tab active. To keep the Add-in panel active always during this process of working with add-in ribbons, end users can either drag the add-in panel off the ribbon add-ins tab. The other alternative is to uncheck 'Display the contextual tab on Selection' in the Big R > Options dialog, under the User Interface tab.



Ribbon Panel

As you might have known already, external commands are listed under the 'External Tools' drop-down list button in the *Add-In* tab. External applications which use the Ribbon API are displayed on the custom ribbon panels. These custom panels act as container and can contain a number of ribbon items or widgets, including:

- buttons, both large and small, which can be either simple push buttons,
- pull-down buttons containing multiple commands,
- split buttons which are pull-down buttons with a default push button attached,
- radio buttons,
- combo boxes
- text boxes
- Slide out controls which can be accessed by clicking the bottom of the panel.

Panels can also include vertical separators to help separate commands into logical groups.

The RibbonPanel class represents the ribbon panel that can be added to the Add-Ins, Analyze or any custom tab. The ribbon panels are created using UIControlledApplication.CreateRibbonPanel() method. There are three overloads for the same method:

1) Adding a panel to the default Add-in tab which can be done using the following method overload

```
  RibbonPanel firstPanel = application.CreateRibbonPanel("Snapshot : Panel
One");
```

2) Adding a panel to an existing custom tab by specifying the name of the tab

```
  RibbonPanel secondPanel = application.CreateRibbonPanel("AU 2012",
panelName);
```

3) Adding a panel to either the *Add-in* or *Analyze* tab using the Tab enumeration

```
  RibbonPanel thirdPanel = application.CreateRibbonPanel(Tab.Analyze,
"Snapshot : Panel Three");
```

Panel Name vs. Panel Title

RibbonPanel.Name represents the name of a panel whereas RibbonPanel.Title represents the text that appears on the ribbon panel. Two ribbon panels can have the same title, as long as they have different names. So to avoid any conflicts with panel names, it might be a good idea to check if a panel name is already being used in other add-ins that the end user might already have. The UIControlledApplication class provides the ability to extract all the ribbon panels from the Revit Add-in, Analyse or any custom tabs using one of the overloads of the *GetRibbonPanels()*. The code included below shows how this method can be used to get the list of panels and a validation can be performed to ensure that there is no conflict with proposed the ribbon panel name with those that already exist.

```
// If panels have different names, same Titles can exist
List<RibbonPanel> loadedPanels = controlledApp.GetRibbonPanels();

foreach (RibbonPanel p in loadedPanels)
{
  if (p.Name.Equals(panelName))
  {
    return true;
  }
  }
return false;
```

Ribbon Control Classes

The Ribbon panel contains a number of ribbon buttons or other ribbon items. Thus, the RibbonItem object represents an item on RibbonPanel, and this can be push-button, pull-down button, combo box, textbox, radio button etc. This class contains the information for creating one RibbonItem including properties like Visible, Enabled, etc.

Each ribbon control has two classes associated with it – one derived from RibbonItemData that is used to create the control (i.e. SplitButtonData) and add it to a ribbon panel; and one derived from RibbonItem (i.e. SplitButton) which represents the item after it is added to a panel. The properties available from RibbonItemData (and the derived classes) are also available from RibbonItem (and the corresponding derived classes). These properties can be set prior to adding the control to the panel or can be set using the RibbonItem class after it has been added to the panel.

Thus, each of the UI ribbon item like push buttons, split buttons, combobox, etc have two classes associated with them –

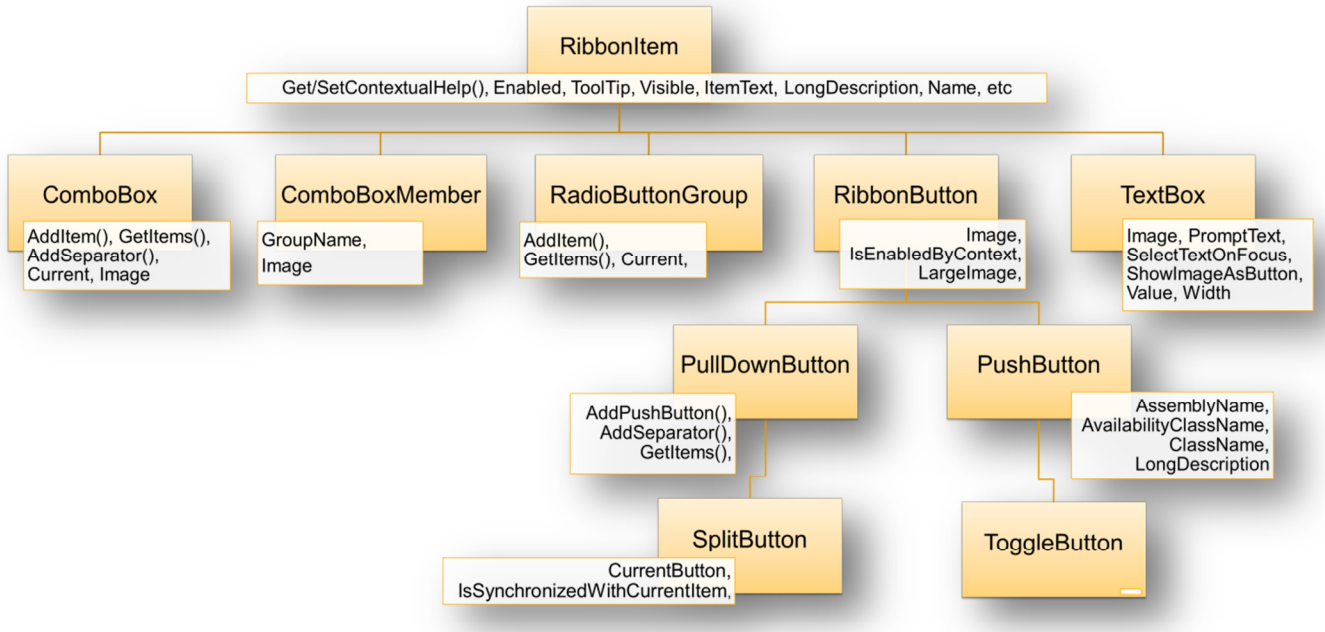
- PushButton, PushButtonData
- PullButton, PullButtonData
- SplitButton, SplitButtonData
- ComboBox, ComboBoxData

- …and so on.

Each of the ribbon items (split buttons, pull down buttons, push down buttons, etc) provide some common set of functionality, for example, ability to set the tool tip property, associate images to the ribbon items, etc.

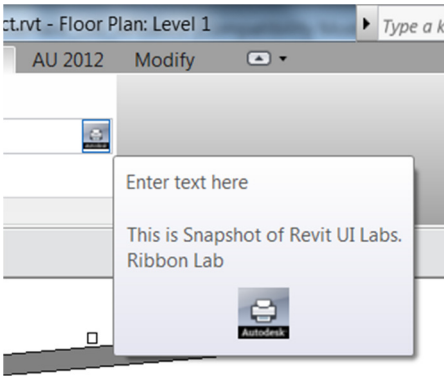
Ribbon Control Class Hierarchy

The following image shows the Ribbon API Class hierarchy with the additional class-specific method and properties exposed:



Tooltips

The RibbonItem class, from which all the ribbon items are derived out of, provides a ToolTip property. This helps provide a description that appears as a tool tip for each of the ribbon items. This text is displayed when the mouse pointer moves over the item. If there is a need to provide extended tooltip, it can be displayed by setting the LongDescription property on a ribbon item. This tooltip is shown when the mouse hovers over the command for a longer duration. You can split the text of this option into multiple paragraphs by placing <p> tags around each paragraph. The *ToolTipImage* property helps set the image that can be shown as a part of the button extended tooltip, too.



If no tooltip has been set on a ribbon item via the RibbonItem or RibbonItemData class, RibbonItem.ItemText is displayed, by default.
The screenshot above of the extended tooltip has been created using the following code snippet:

```
// fill the text gox information
TextBoxData txtBoxData = new TextBoxData("TextBox");
```

```
txtBoxData.Image = new BitmapImage(
  new Uri(imageFolder + "printer_16.png"));
txtBoxData.Name = "Text Box";

txtBoxData.ToolTip = "Enter text here";
txtBoxData.LongDescription =
  "<p>This is Snapshot of Revit UI Labs.</p><p>Ribbon Lab</p>";
txtBoxData.ToolTipImage = new BitmapImage(
  new Uri(imageFolder + "printer_32.png"));
```

Associating images

The Ribbon API also provides the ability to associate images to each of the ribbon items. This can be done using the *RibbonItem.LargeItem* property which is the image that is shown on the item if it not part of a stacked set of items and can also be set for items that are included in a set of pull down buttons. This image should be 32 x 32 pixels.

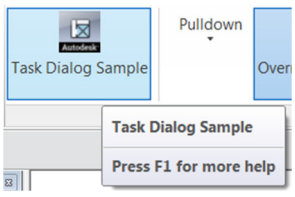
The *RibbonItem.Image* property helps set the image that will be shown on the ribbon item if it is a part of a stacked set, in text boxes and in combo boxes. It is also used if the ribbon item is promoted to the Quick Access Toolbar (QAT). This image should be 16 x 16 pixels.

```
// create bitmap image for button
Uri uriImage = new Uri(imageFolder + "rooms_32.png");
BitmapImage largeImage = new BitmapImage(uriImage);

// assign bitmap to button
pushButton.LargeImage = largeImage;

// assign a small bitmap to button which is used if
// command is moved to Quick Access Toolbar
Uri uriSmallImage = new Uri(imageFolder + "rooms_16.png");
BitmapImage smallImage = new BitmapImage(uriSmallImage);

// assign small image to button
pushButton.Image = smallImage;
```

To have the ribbon display icons appear as expected, please ensure the icon images are set to 96 dpi resolution.

Ribbon Controls

Push Buttons



This image shows a push button. When a push button is pressed, the corresponding command is triggered.

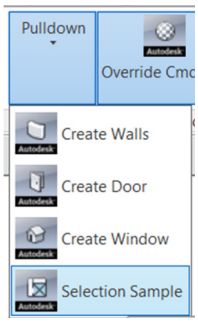
The following first two lines of code show how we can use the PushButtonData class to create a push button on a ribbon and then the PushButton class represents the ribbon item after it has been added to the panel.

```
PushButtonData pushButtonData = new PushButtonData(
  "TaskDialogSample",
  "Task Dialog Sample",
```

```
    assemblyPath + "\\" + assemblyName,
    "SnapshotRevitUI_CS.UITaskDialog");

  PushButton pushButton =
    panel.AddItem(pushButtonData) as PushButton;
  pushButton.AvailabilityClassName =
    "SnapshotRevitUI_CS.SampleAccessibilityChecker";

  ContextualHelp contextHelp = new ContextualHelp(
    ContextualHelpType.Url,
    "http://www.autodesk.com/developrevit");

  pushButton.SetContextualHelp(contextHelp);

  // create bitmap image for button
  Uri uriImage = new Uri(imageFolder + "rooms_32.png");
  BitmapImage largeImage = new BitmapImage(uriImage);

  // assign bitmap to button
  pushButton.LargeImage = largeImage;

  // assign a small bitmap to button which is used if
  // command is moved to Quick Access Toolbar
  Uri uriSmallImage = new Uri(imageFolder + "rooms_16.png");
  BitmapImage smallImage = new BitmapImage(uriSmallImage);

  // assign small image to button
  pushButton.Image = smallImage;
```

Following this, we can use the PushButton object to set its various properties like Images, tool tip, contextual help, etc.

Drop-down Buttons



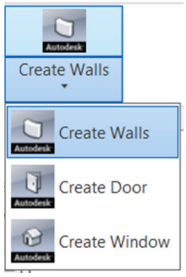
Drop-down buttons, as the name suggests, refers to the UI widget which on click expands to list more commands. On the API side, drop-down buttons are referred to as PulldownButtons. Horizontal separators can be added between items in the drop-down menu.

The code snippet included below shows how to create a drop-down button:

```
  PushButtonData pushButtonData1 = new PushButtonData(
    "CreateWallsSample",
    "Create Walls",
    assemblyPath + "\\" + assemblyName,
    "SnapshotRevitUI_CS.CreateWalls");

  PushButtonData pushButtonData2 = new PushButtonData(
    "CreateDoorSample",
```

```
    "Create Door",
    assemblyPath + "\\" + assemblyName,
    "SnapshotRevitUI_CS.CreateDoor");
      …
      …

  // make a pulldown button now
  PulldownButtonData pulldownBtnData =
    new PulldownButtonData("PulldownButton", "Pulldown");
  PulldownButton pulldownBtn =
    panel.AddItem(pulldownBtnData) as PulldownButton;
  pulldownBtn.AddPushButton(pushButtonData1);
  pulldownBtn.AddPushButton(pushButtonData2);
```

Split buttons

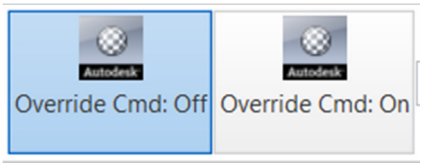


Split buttons are combination of push button which is the top half of the button and bottom half which is a drop-down button. Note that the ToolTip, ToolTipImage and LongDescription properties for SplitButton are ignored – only the tooltip for the current push button is shown instead.

```
  PushButtonData pushButtonData1 = new PushButtonData(
    "SplitCreateWall",
    "Create Walls",
    assemblyPath + "\\" + assemblyName,
    "SnapshotRevitUI_CS.CreateWalls");

  PushButtonData pushButtonData2 = new PushButtonData(
    "SplitCreateDoor",
    "Create Door",
    assemblyPath + "\\" + assemblyName,
    "SnapshotRevitUI_CS.CreateDoor");
  pushButtonData2.LargeImage = new BitmapImage(
    new Uri(imageFolder + "doors_32.png"));
      …

  // make a split button now
  SplitButtonData splitBtnData = new SplitButtonData(
    "SplitButton", "Split Button");
  SplitButton splitBtn =
    panel.AddItem(splitBtnData) as SplitButton;
  splitBtn.AddPushButton(pushButtonData1);
  splitBtn.AddPushButton(pushButtonData2);
```

Radio Buttons

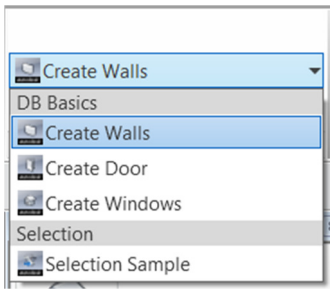
A radio button group helps toggle between options by letting users selected only one ribbon item at a time. After adding a RadioButtonGroup to a panel, use the AddItem() or AddItems() methods to add



toggle buttons which are nothing but PushButtons, to the group. The RadioButtonGroup.Current property can be used to access the currently selected button. Just like SplitButton, tooltips do not apply to radio button group but instead the tooltip for each toggle button is displayed.

```
RadioButtonGroupData radioData =
  new RadioButtonGroupData("radioGroup");
RadioButtonGroup radioButtonGroup =
  panel.AddItem(radioData) as RadioButtonGroup;

ToggleButtonData tb1 = new ToggleButtonData(
  "OverrideCommand1",
  "Override Cmd: Off",
  assemblyPath + "\\" + assemblyName,
  "SnapshotRevitUI_CS.OverrideOff");

ToggleButtonData tb2 = new ToggleButtonData(
  "OverrideCommand2",
  "Override Cmd: On",
  assemblyPath + "\\" + assemblyName,
  "SnapshotRevitUI_CS.OverrideOn");
tb2.ToolTip = "Override the Wall Creation command";
tb2.LargeImage = new BitmapImage(
  new Uri(imageFolder + "globe_32.png"));
radioButtonGroup.AddItem(tb1);
radioButtonGroup.AddItem(tb2);
```

ComboBox



A combo box is a drop-down list with a set of selectable items that can be shown or hidden by clicking the arrow

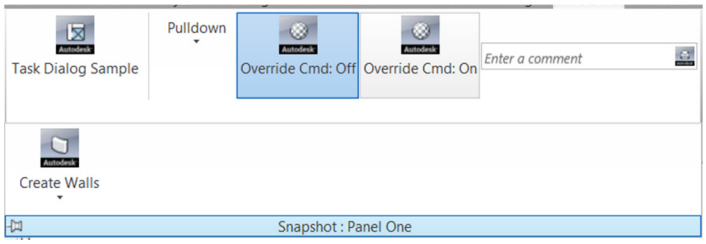
Separators can also be added to separate items in the list or members can be optionally grouped using the ComboBoxMember.GroupName property. All members with the same GroupName will be grouped together with a header that shows the group name. Any items not assigned a GroupName will be placed at the top of the list.

```
ComboBoxMemberData comboBoxMemberData1 =
  new ComboBoxMemberData("ComboCreateWalls", "Create Walls");
comboBoxMemberData1.Image =
  LoadPNGImageFromResource(
  "SnapshotRevitUI_CS.Resources.wall_16.png");
comboBoxMemberData1.GroupName = "DB Basics";

// #2
ComboBoxMemberData comboBoxMemberData2 =
  new ComboBoxMemberData("ComboCreateDoor", "Create Door");
comboBoxMemberData2.Image = LoadPNGImageFromResource(
  "SnapshotRevitUI_CS.Resources.doors_16.png");
```

```
comboBoxMemberData2.GroupName = "DB Basics";
…

// make a radio button group now
ComboBoxData comboBxData = new ComboBoxData("ComboBox");
ComboBox comboBx = panel.AddItem(comboBxData) as ComboBox;
comboBx.ToolTip = "Select an Option";
comboBx.LongDescription = "select a command you want to run";
comboBx.AddItem(comboBoxMemberData1);
comboBx.AddItem(comboBoxMemberData2);
…
```

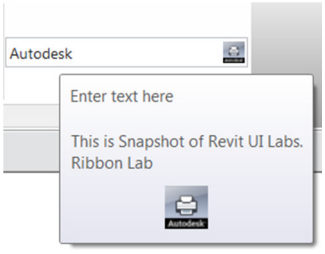
ComboBox additionally expose events to help be notified when current item is changed, or when the drop-down of the ComboBox is opened or closed.

SlideOut Panel



The API enables creation of a slide out at the bottom of the ribbon panel. This can be done using the RibbonPanel.AddSlideOut() method. When a slide-out is added, an arrow is shown on the bottom of the panel, indicating the existence of the slide-out panel. After calling AddSlideOut(), subsequent calls to add new items to the panel will be added to the slide-out, so the slide-out must be added after all other controls have been added to the ribbon panel.

TextBox



A text box is an input control for users to enter text. The image for a text box can be used as a clickable button by setting the ShowImageAsButton property to true.

The text entered in the text box is only accepted if the user hits the Enter key or if they click the associated image when the image is shown as a button.

In addition to providing a tooltip for a text box, the PromptText property can be used to indicate to the user what type of information to enter in the text box. Prompt text is displayed when the text box is empty and does not have keyboard focus in italics.

The width of the text box can be set using the Width property. The default is 200 device-independent units.

```
// fill the text gox information
TextBoxData txtBoxData = new TextBoxData("TextBox");
txtBoxData.Image = new BitmapImage(
  new Uri(imageFolder + "printer_16.png"));
txtBoxData.Name = "Text Box";
txtBoxData.ToolTip = "Enter text here";
txtBoxData.LongDescription =
  "<p>This is Snapshot of Revit UI Labs.</p><p>Ribbon Lab</p>";
txtBoxData.ToolTipImage = new BitmapImage(
```

```
    new Uri(imageFolder + "printer_32.png"));

  // create the text box item on the panel
  TextBox txtBox = panel.AddItem(txtBoxData) as TextBox;
  txtBox.PromptText = "Enter a comment";
  txtBox.ShowImageAsButton = true;

  txtBox.EnterPressed +=
    new EventHandler
      <Autodesk.Revit.UI.Events.TextBoxEnterPressedEventArgs>
      (txtBox_EnterPressed);
  txtBox.Width = 180;
```

The TextBox class exposes an event to be notified when Enter button is pressed in the text box or when they click on the associated image for the text box when ShowImageAsButton is set to true.

Stacked Items

Ribbon items can be stacked one on top of another to save panel space - as shown in the image. Each item in the stack can be a push button, a drop-down button, a combo box or a text box. Radio button groups and split buttons cannot be stacked.

Stacked buttons should have an image associated through their (16x16) *Image* property, and not the *LargeImage* property.

```
  PushButtonData buttonWalls = new PushButtonData(
    "StackCreateWall",
    "Create Walls",
    assemblyPath + "\\" + assemblyName,
    "SnapshotRevitUI_CS.CreateWalls");
  buttonWalls.Image = LoadPNGImageFromResource(
    "SnapshotRevitUI_CS.Resources.wall_16.png");

  PushButtonData buttonDoor = new PushButtonData(
    "StackedCreateDoor",
    "Create Door",
    assemblyPath + "\\" + assemblyName,
    "SnapshotRevitUI_CS.CreateDoor");
  buttonDoor.Image = LoadPNGImageFromResource(
    "SnapshotRevitUI_CS.Resources.doors_16.png");
  …

  // Add the stacked buttons to the panel
  panel.AddStackedItems(buttonWalls, buttonDoor, buttonWindows);
```

Combining two Ribbon Controls

If there is a need to create a labelled TextBox, API users can use a combination of ribbon controls to meet this requirement – which in this case would be to create a stacked control with the first item as a disabled pushbutton above the second item, which would be the TextBox itself.

Availability of buttons and external commands

Availability of Push or Toggle button

The class that represents a push button or a toggle button (a button that is added to a RadioButtonGroup), also makes it possible to control whether the button may be pressed or not. This is made possible via the *AvailabilityClassName* property which accepts the full class name of a class which controls the availability of this button - controls whether or not an external command button may be pressed. The class which provides this entry point has to implement the *IExternalCommandAvailability* interface and this passes the application and set of categories matching the categories of the selected items. Using these two parameters, API users can check for certain specific criteria like if the active view is a 3D view or if the selected elements belong to a specific category, and accordingly control the visibility of the button.

```
  pushButton.AvailabilityClassName =
    "SnapshotRevitUI_CS.SampleAccessibilityChecker";

   class SampleAccessibilityChecker : IExternalCommandAvailability
  {
    public bool IsCommandAvailable(Autodesk.Revit.UI.UIApplication
applicationData,
      CategorySet selectedCategories)
    {
      // check if there is any active document
      if (null != applicationData.ActiveUIDocument)
      {
        // Get access to the active view
        View activeView =
applicationData.ActiveUIDocument.Document.ActiveView;
        //If the view is 3D view
        if (ViewType.ThreeD == activeView.ViewType)
        {
          return true;
        }

      }
      return false;
    }
  }
```

Additionally, the Revit API chm file mentions that this *IExternalCommandAvailability* interface should share the same assembly with add-in External Command.

Availability of External Commands

The availability of external commands from External Tools drop-down list can be controlled by the settings <VisibilityMode> in the .addin manifest file. The setting in this node can control the availability of the command based on specific flavors of Revit, control if the command should be visible in project mode or family editor mode, etc. External commands can also use <AvailabilityClassName> to control greyed out depending on context. This AvailabilityClassName is the class name that implements the *IExternalCommandAvailability* as discussed in the previous section.

The availability of external applications (other than in Autodesk Revit 2013) can be controlled by accessing the *UIControlledApp.ControlledApplication.Product* in the OnStartup method of Revit. This Product property returns a *ProductType* enumeration specifying the specific flavor of Revit that the external application has been loaded in. And accordingly, API users can choose to display their ribbon items and have greater control over their visibility/availability.

Availability of Ribbon Items in Zero Document State

By default, ribbon buttons defined by external applications are disabled in zero document state – a state when there are no open documents in Revit. If API users wish to have their addin buttons enabled, they need to set the *AvailabilityClassName* property for the, say, PushButtonData to be set to a class which implements the *IExternalCommandAvailability* interface and have this class return a *True* value. The code snippet included above shows how to create a class which implements this *IExternalCommandAvailability* interface. The only caveat in this case is that the AvailabilityClassName should be assigned to the PushButtonData before adding this ribbon item to the panel – setting it after it has been added to the panel does not work. Alternatively, we can also set this property on the PushButton instance after it has been added to the ribbon panel.

New in Revit 2013

Discipline Control

Leveraging the new Revit 2013 API, your application can read the properties of Application class to determine when to enable or disable aspects of the UI. This is possible using the new Application.IsArchitectureEnabled, IsStructureEnabled, and other similar properties. These new properties not only provide read but also modify access to the available disciplines.

Enabling and disabling disciplines is available only in Autodesk Revit 2013 (and not in any other Revit product). When a discipline's status is toggled, Revit's UI will be adjusted and certain operations and features will be enabled or disabled as appropriate.

The code snippet included below shows how to use the discipline controls in Autodesk Revit 2013 to check the flavor and appropriately control the availability of a button.

```
pushButtonData1.AvailabilityClassName =
  "SnapshotRevitUI_CS.ApplicationAvailabilityChecker";

class ApplicationAvailabilityChecker : IExternalCommandAvailability
{
  public bool IsCommandAvailable(UIApplication applicationData,
    CategorySet selectedCategories)
  {
    Application revitApplication = applicationData.Application;

    //If the flavor of Revit is Architecture in Autodesk Revit
    if (revitApplication.IsArchitectureEnabled)
      return true;
    else
      return false;
  }
}
```

Contextual (F1) help

With Revit 2013 API, contextual (or F1) help can be assigned to each ribbon item. This enables users to simply hit F1 on an add-in command button (after putting the mouse pointer on the button) and this will bring up an Internet URL or a local help file. This feature thus provides the same user experience when working with Revit's UI buttons and is part of the broader Add-in integration project.
These actions that can be assigned via the contextual help are - linking to an external URL, launching a locally installed help file, linking to a topic on Autodesk Wiki help. This can be done using *RibbonItem.SetContextualHelp()* method, as shown below.

1) Linking to a specific URL:

```
ContextualHelp contextHelp = new ContextualHelp(
  ContextualHelpType.Url,
  "http://www.autodesk.com/developrevit");
pushButton.SetContextualHelp(contextHelp);
```

2) Linking to a specific page on Revit Wiki help:

```
ContextualHelp contextHelp1 = new ContextualHelp(
  ContextualHelpType.ContextId, "HID_OBJECTS_WALL");
pushButtonData1.SetContextualHelp(contextHelp1);
```

3) Linking to a specific page in the Revit API chm file – this could be any locally installed help file too:

```
ContextualHelp ch2 = new ContextualHelp(
  ContextualHelpType.ChmFile,
  @"C:\Saikat\Revit SDKs\2013\RTM\Software Development Kit\RevitAPI.chm");
ch2.HelpTopicUrl = @"html/0c0d640b-7810-55e4-3c5e-cd295dede87b.htm";
pushButtonData2.SetContextualHelp(ch2);
```

These actions can also be invoked at any point of time in your add-in by creating a *ContextualHelp* object and using *ContextualHelp.Launch()* method.

Keyboard Shortcuts and Quick Access Toolbar

Revit 2013 API also includes support for API commands to be assigned Keyboard Shortcuts and also moved to Quick Access Toolbar. These assignments are preserved if add-ins are added, removed or changed.

Replace Implementation of Commands

Revit 2013 API includes the ability for an add-in to replace an existing Revit command with its own implementation. The existing Revit commands can be located in any tab, application menu or right-click menu. We still cannot call an existing command but we can replace it completely.

A built-in command can be replaced using a new class called *RevitCommandId*, which provides the ability to look up and retrieve an object representing the Revit Command Id given an ID string using *RevitCommandId.LookupCommandId().* One approach to obtain the built-in command's ID string that the *RevitCommandId* class can use is to call the built-in command from the Revit UI and then examine the journal file and search for the command Id string.

Once we have the RevitCommandId, we can use the *CreateAddInCommandBinding()* on the Application object to create an instance of *AddInCommandBinding.* This object provides the ability to override the Revit command implementation with Executed and CanExecute events.

```
RevitCommandId wallCreate =
  RevitCommandId.LookupCommandId("ID_OBJECTS_WALL");

AddInCommandBinding binding =
  commandData.Application.CreateAddInCommandBinding(wallCreate);

binding.Executed +=
  new EventHandler<ExecutedEventArgs>(binding_Executed);
binding.CanExecute +=
  new EventHandler<CanExecuteEventArgs>(binding_CanExecute);
```

The alternate implementation (the implementation which will replace the command) is provided by the *AddInCommandBinding's* Executed event handler. The code in this event handler is called whenever the command Id that is being replaced, is triggered.

The CanExecute event gets triggered when the command associated with this *AddInCommandBinding* initiates a check to determine whether the command can be executed on the command target.

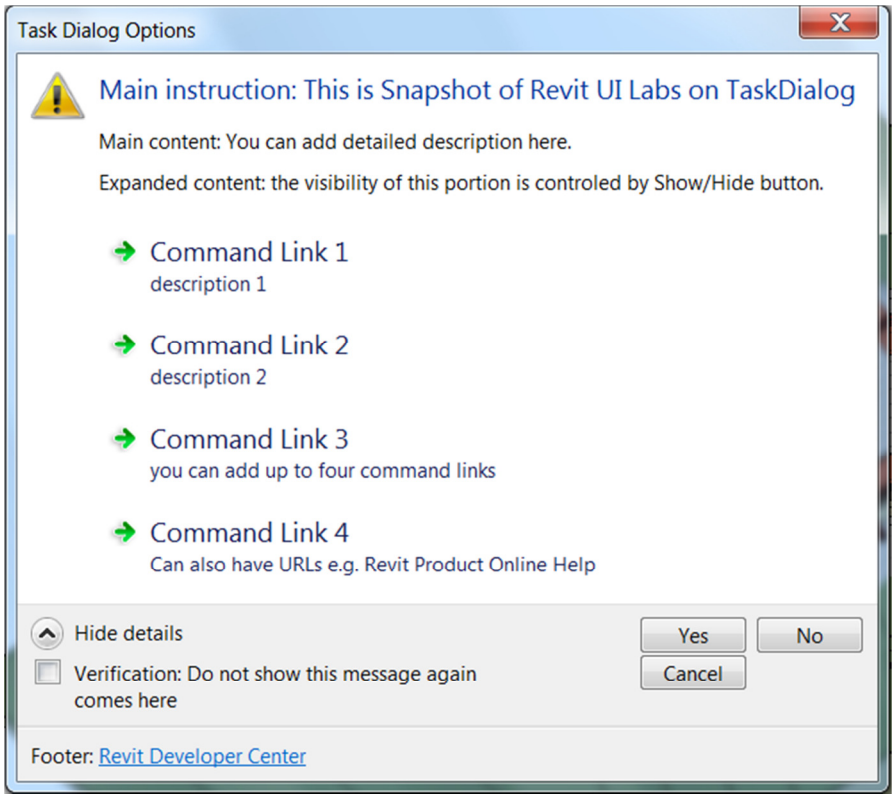
To remove this override implementation of the Revit command, we can simply use the *RemoveAddInCommandBinding* and pass on the Revit Command Id whose implementation we want to restore to default.

```
RevitCommandId wallCreate =
  RevitCommandId.LookupCommandId("ID_OBJECTS_WALL");

commandData.Application.RemoveAddInCommandBinding(wallCreate);
```

Task Dialogs

A TaskDialog is a modal dialog with a set of controls and is the Revit-styled alternative to a simple Windows MessageBox. It can be used to display/provide information, ask a question and receive simple input from the user based on a set of options. It has a common set of controls that are arranged in a standard order to assure consistent look and feel with the rest of Revit user interface.

Task dialogs cannot display other controls such as, text inputs, list boxes, combo boxes, check boxes, etc. They also only accommodate single step, single action operations; meaning a user may make a single choice and complete the task dialog operation. As a result any dialog that requires such additional controls or multiple steps operations (as with a wizard) would need to be implemented as custom dialogs using .NET controls to have a similar look & feel to Task Dialogs.



There are two ways to create and show a task dialog to the user –

The first option is to create a new instance of the TaskDialog, set its properties individually, and use the instance method Show() to show it to the user.

```
TaskDialog myDialog = new TaskDialog("Task Dialog Options");
if (stepByStep) myDialog.Show();

myDialog.MainIcon = TaskDialogIcon.TaskDialogIconWarning;
// or TaskDialogIcon.TaskDialogIconNone.
if (stepByStep) myDialog.Show();

myDialog.MainInstruction =
  "Main instruction: This is Snapshot of Revit UI on TaskDialog";
if (stepByStep) myDialog.Show();

myDialog.MainContent =
  "Main content: You can add detailed description here.";
if (stepByStep) myDialog.Show();
```

The second is to use one of the static Show() methods to construct and show the dialog in one step. When you use the static methods, only a subset of the options can be specified.

```
TaskDialogResult result = TaskDialog.Show(
  "Static Task Dialog",
  "Main message",
```

```
    (TaskDialogCommonButtons.Yes
    ^ TaskDialogCommonButtons.No
    ^ TaskDialogCommonButtons.Cancel),
    TaskDialogResult.No);
  TaskDialog.Show("Snapshot of Revit UI Labs",
    "You pressed: " + result.ToString());
```

And task dialogs can be used to set instructions, detailed text, icons, buttons, command links, verification texts, etc.

User Selection and Status Bar

The Selection class has methods that allow users to select elements - single or multiple selections using pick selection or using rectangular box selection, a point on screen, edge or a face. This selection mechanism uses the cursor and then returns control back to the add-in. These functions do not automatically add the new selection to the collection containing the active selection.

The PickObject() method prompts the user to select an object in the Revit model.

The PickObjects() method prompts the user to select multiple objects in the Revit model.

The PickBox() method prompts the user to pick two points that lets users specify a rectangular area on the screen. The style of the box can be Crossing, Enclosing or Diagonal and the method returns a PickedBox element with access to minimum and maximum coordinates.

The PickElementsByRectangle() method prompts the user to select multiple elements using a rectangle, what is often also known as ‘box select’.

The PickPoint() method prompts the user to pick a point in the active sketch plane and this returns the XYZ value corresponding to the picked point.

```
  IList<Element> elemsRec =
    uiDoc.Selection.PickElementsByRectangle(
    "Select by rectangle");
  // show it.
  ShowElementList(elemsRec, "Pick By Rectangle: ");
```

The type of object to be selected is specified when calling PickObject() or PickObjects(). Types of objects that can be specified are: Element, PointOnElement, Edge or Face.

We can also add custom status messages to the pick functions prompting users on what needs to be selected or picked. Each of the Pick functions has an overload that has a String parameter in which a custom status message can be provided.

With this API, we also have the ability to define the snap settings (or types) during selection. PickPoint() method also has 2 overloads with ObjectSnapTypes parameter which is used to specify the snap types used for selection. The snap types available for selection include Endpoints, Midpoints, Nearest, Workplane Grid, Intersections, Centers, Perpendicular, Tangents, Quadrants, Points and None (for no snap).

```
  ObjectSnapTypes snapTypes =
    ObjectSnapTypes.Endpoints | ObjectSnapTypes.Intersections;
  msg = String.Empty;
  msg = "Select an end point or intersection";
  XYZ point = uiDoc.Selection.PickPoint(snapTypes, msg);
```

```
msg = String.Empty;
msg += PointToString(point);
TaskDialog.Show("Snapshot of Revit UI Labs", msg);
```

We can also set the active work-plane with the View.SketchPlane() so that we have control on the point that we are having the user select, thus, enhancing the UI experience using the API.

Highlighting Elements

If an add-in is required to highlight certain elements in the user interface programmatically, we can use the Selection API to meet this requirement. All we need to do is to add the element(s) to be highlighted, to a selection set using the API, as shown below.

```
uidoc.Selection.Elements.Add(element);
```

This can help highlight the required elements programmatically.

3D Point Selection

As you might have seen already, PickPoint() only works for 2D point selection. If there is a requirement for 3D point selection somewhere that is not on the active work plane, we need to identify the plane on which the point has to be picked. To do this, we need to prompt the users to select the face of an element, create a new plane using this face. With this new plane, we need to create a sketch plane, set this to be to be the active work plane and show this sketch plane. Now we can prompt the users to select the point using PickPoint().

Selection Filter

The user selection API allows API users to filter objects during the selection. The element or object selection – single and multiple selection methods, all have overloads that take an ISelectionFilter as a parameter. These pick methods, in particular, include the PickElementsByRectangle, PickObject and PickObjects methods. And it is this ISelectionFilter interface that enables the filtration of objects during a selection operation. It has two methods that can be overridden: AllowElement() which is used to specify if an element is allowed to be selected, and AllowReference() which is used to specify if a reference to a piece of geometry is allowed to be selected.

The code snippet included below demonstrates the use of ISelectionFilter to allow selection of planar faces only.

```
SelectionFilterWall selFilterWall =
  new SelectionFilterWall();
Reference refWall = uiDoc.Selection.PickObject(
  ObjectType.Element,
  selFilterWall,
  "Select a wall");

  class SelectionFilterWall : ISelectionFilter
{
  public bool AllowElement(Element elem)
  {
      if (elem.Category.Id.IntegerValue.Equals(
        (int)BuiltInCategory.OST_Walls))
      {
```

```
            return true;
        }
        return false;
    }

    public bool AllowReference(Reference reference, XYZ position)
    {
        return true;
    }
}
```

SDK Samples

Revit SDK also contains couple of samples which cover the UI API topics discussed in this class. They are:

- The **Ribbon** sample demonstrates how we can create customized ribbon with various ribbon items (like PullDown buttons, Stackable buttons, Split buttons, etc).
- The **External Command** sample illustrates how to enable/disable each external command based on users selection and application information.
- The **Selections** sample demonstrates how to perform various selection operations including PickObject, PickObjects, PickPoints and even implenting the ISelectionFilter.

Conclusion

In this class, we looked at a 'snapshot' of the Revit UI and discussed its customization possibilities with the Revit API. We also covered some new APIs related to this topic in Revit 2013. During this process, we also covered some of commonly asked for customization requirements by the Revit API community.

Besides the UI API topics that were covered in this class, following is the list of the remaining Revit UI API topics:

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

These, as mentioned before, will be covered in ***CP4107*** *- Let's Face It: New Autodesk® Revit® 2013 User Interface API Functionality.* Attending both these classes, should provide the complete understanding of the UI customization with Revit API.

Learn More

Resources on your system

- Software Development Kit
  - Samples
  - Documentation
  - Tools - Autodesk Revit® LookUp and Add-In Manager

Resources Online

- Revit API page on ADN Open: http://www.autodesk.com/developrevit
  - Latest SDK
  - DevTVs
  - API Training Material

- Revit Product help and API Developer guide.
  - http://www.autodesk.com/revitapi-wikihelp

- Blogs
  - ADN AEC DevBlog : http://adndevblog.typepad.com/AEC
  - The Building Coder blog by Jeremy Tammik: http://thebuildingcoder.typepad.com

- Discussion Forums
  - http://discussion.autodesk.com > Revit Architecture > Revit API

- The Autodesk Developer Network, and DevHelp Online for ADN members
  - http://www.autodesk.com/joinadn
  - http://adn.autodesk.com

Thanks for attending this class!