AUTODESK UNIVERSITY 2014

Advanced Revit® Remote Batch Command Processing

David Echols – Hankins & Anderson, Inc. – Senior Programmer

SD5980 This class will explain a process to run external commands in batch mode from a central server to remote Revit® application workstations. We will cover how to use client and server applications that communicate with each other to manage Revit® software on remote workstations with WCF (Windows Communication Foundation) services. We will examine how to pass XML command data to the Revit® application to open a Revit® model and initiate batch commands. We will also show a specific use case for batch export of DWG files for sheets. We will examine a flexible system for handling Revit® dialog boxes on the fly with usage examples and code snippets, and we will discuss the failure processing API in the context of bypassing warning and error messages while custom commands are running. Finally, we will show you how to gracefully close both the open Revit® model and the Revit® application.

Learning Objectives

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

- Learn how to start and stop the Revit® application on remote workstations and open Revit® models for batch processing.
- Learn how to load an assembly based on XML data and then execute the code on the open Revit® model.
- Learn how to suppress Revit® dialog boxes and error/warning messages.
- Learn how to close the open Revit® model and exit the Revit® application.

About the Speaker

Dave is a Senior Programmer for Hankins & Anderson, Inc., an architecture and engineering firm based in Richmond, VA. Hankins & Anderson is an Autodesk Developer Network member. Living in Virginia Beach, VA, he provides custom programming and IT support for the firm's engineering design software, including Autodesk® Revit®, AutoCAD® MEP, AutoCAD® Architecture and more. He has 28 years of experience in building custom solutions and has worked for DOD Contractors, design firms, the Autodesk VAR channel and a small startup designing and developing a digital pen forms application. His Autodesk development experience begins with version 2.6 of AutoCAD® and Autodesk® Revit® 2009. Dave is proficient in the .NET Framework, using C# and the various Autodesk .NET APIs to develop and maintain his company's internal applications. He founded the Hampton Roads Autodesk Users Group and served various roles during the life of the club.

D.Echols@ha-inc.com

http://www.ha-inc.com

H&A hankins & anderson

AUTODESK Authorized Developer

Contents

Introduction

Over the years, AutoCAD automation has evolved through the incorporation and continuous improvement of numerous APIs; Script files, Autolisp/Visual Lisp, ADS, VBA, .NET and JavaScript. AutoCAD was designed from its early origins with automation in mind. The same cannot be said for Revit®. It was designed as a 3D modeling tool entirely driven by the GUI. This design difference makes Revit® more difficult to automate than AutoCAD. The only API available to Revit® developers is the .NET API. Over the years, the Revit® API development team has added new functionality making automation easier. This class will explore some of these new features and describe a process for automating Revit® in a Client/Server environment.

Why Automate?

Most of my company's submittals require several items; the Revit® models, an accurate drawing index, PDF files of each drawing sheet in each model and AutoCAD DWG files of each drawing sheet in each model. We have some projects with weekly progress submittals. Some of our larger projects have over 50 Revit® models, so compiling these items for each submittal becomes time consuming when using the Revit® GUI. We had to find a way to automate Revit®.

Version 1.0

Starting with Revit® 2012, we put in place a process to automate Revit® for three specific items:

1. Drawing Sheet data extraction into a SQL Server database
2. PDF printing if valid Drawing Sheets
3. DWG export of valid Drawing Sheets

This first implementation was based on a presentation by Rod Howarth given as an AU Virtual Lecture for Autodesk University 2011. See Rod's blog article for details; http://blog.rodhowarth.com/2011/09/Revit®-autodesk-university-2011-virtual.html. Rod's example was centered on using Revit® Server and sending commands to the server to open a model and run a command. My company did not want to use Revit® Server so I adopted Rod's idea for use on several remote workstations not being used during off hours. This system has evolved slightly over the last three years and has worked for Revit® 2012, 2013 and 2014. During a peak in our project work, this system extracted drawing sheet data and exported PDF files for approximately 2,800 drawing sheets.

Version 2.0

Version 1.0 requires some manual setup for new projects and is not flexible when schedule changes are needed. Version 2.0 was started to fix these issues. Goals for the new version area;

1. Simplified process for creating the data that defines a job.
2. Flexible scheduling of overnight jobs on the remote workstations.

3. On demand scheduling of jobs during the day for our CADD Support personnel and Project Managers.

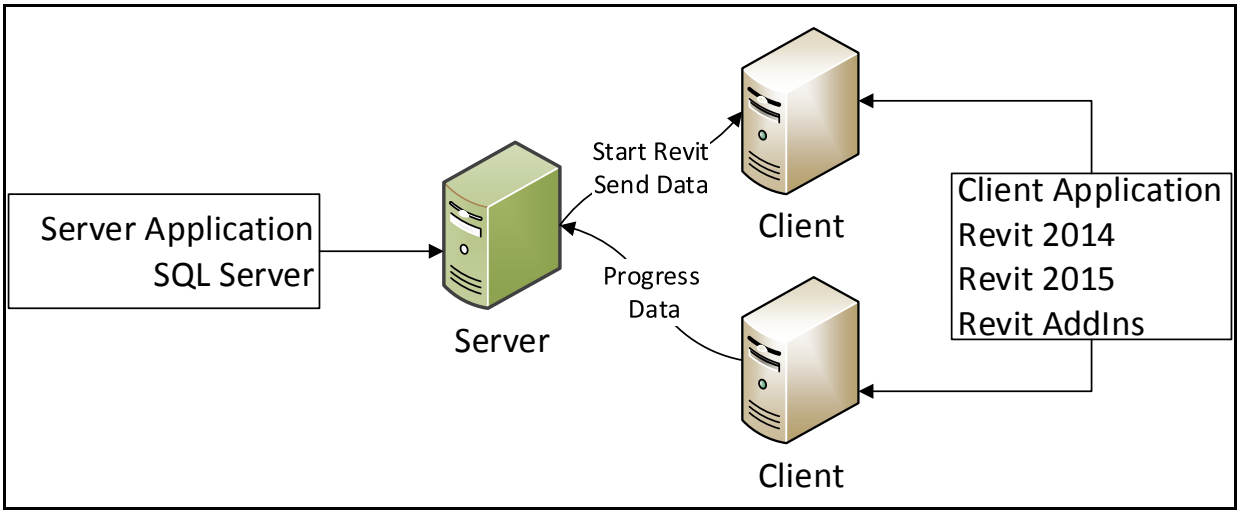
The following sections describe the foundation of this new version and detailed information needed to automate Revit® and handle GUI interactions that occur while running Revit®.

Start and stop Revit® on remote workstations and open Revit® models

The most basic requirement for automating Revit® is starting and stopping the Revit® application on a remote workstation. After the software is started it needed to open one or more Revit® models for processing. A proof in concept application was developed using a client server approach and Microsoft Windows Communication Foundation (WCF) services. This application needed to prove three things:

1. A command could be sent from a server to a client to start a specific version of Revit®.
2. Data could be sent to Revit® to tell it what to do.
3. A Revit® model could be opened after starting Revit®.

The following is a basic network diagram showing data flow:



1 - Network data flow with client/server application

Communication Decisions

In order to communicate between two workstation, some form of remote messaging needs to be used. The two computers will be on the company's internal LAN and need to be a two way connection. Commands need to be sent from the server to the client and the client needs to send progress information back to the server. NetTCP remoting was chosen for this task. The following code shows the NetTCP code used in the server application.

```
string ipaddress = RemoteJobClientsListView.SelectedItems[0].SubItems[1].Text;
EndPointAddr = "net.tcp://" + ipaddress + ":8000/ClientService";
NetTcpBinding tcpBinding = new NetTcpBinding();
tcpBinding.TransactionFlow = false;
tcpBinding.Security.Transport.ProtectionLevel = ProtectionLevel.EncryptAndSign;
tcpBinding.Security.Transport.ClientCredentialType =
TcpClientCredentialType.Windows;
tcpBinding.Security.Mode = SecurityMode.None;
EndpointAddress endpointAddress = new EndpointAddress(EndPointAddr);

Append("Attempt to connect to: " + EndPointAddr);

IClientServiceContract proxy =
  ChannelFactory<IClientServiceContract>.CreateChannel(tcpBinding,
endpointAddress);
```

2 - Implementing NETTCPBinding in RemoteServerForm.cs

In the code above, a known client IP Address is used to build the URL for a client workstation. A NetTcpBinding object is created and configured and an EndPointAddress object is created using the URL. Both objects are used to open the TCP channel to the client using the WCF IClientServiceContract interface discussed below.

WCF Service

A WCF service is used to handle the data and commands passed between the server and client. A separate assembly named ServiceLibrary holds all the WCF specific interfaces and implementations. Part of a ServiceContract Interface is shown below with OperationContract methods defined for transmitting the data to tell Revit® what to do "bool TransmitJob(string version, string jobData)" and starting Revit® "bool StartRevit®(string version)". The term "Job" is used to define the set of data Revit® needs to process a command. A sample XML Job file is shown in Appendix B and is included with the class files.

```
[ServiceContract()]
public interface IClientServiceContract
{
  [OperationContract]
  bool StartRevit(string version);

  [OperationContract]
  bool TransmitJob(string jobData);
}
```

3 - The IClientServiceContract interface in IClientServiceContract.cs

The interface is implemented in the ClientServiceContract class. The TramsmitJob method receives a string containing the contents of the XML job file and saves it to a preconfigured folder as a file named Job.xml. The job file could be read from a file in disk or generated from a SQL Server database.

```
[DataContract]
public class ClientServiceContract : IClientServiceContract
{
  public bool TransmitJob(string jobData)
  {
    try
    {
      string jobFileName = @"C:\ProgramData\RemoteJobs\Job.xml";
      Debug.Print(jobFileName);
      XmlDocument xmlDoc = new XmlDocument();
      xmlDoc.LoadXml(jobData);
      xmlDoc.Save(jobFileName);
      Debug.Print(jobFileName + " was saved!");
      return true;
    }
    catch (Exception ex)
    {
      DebugPrint(ex);
    }
    return false;
  }
}
```

4 - The TransmitJob method in ClientServiceContract.cs

The StartRevit® method shown below calls a private method that uses the System.Diagnostics.Process class to start the appropriate version of Revit®.

```
[DataContract]
public class ClientServiceContract : IClientServiceContract
{
  public bool StartRevit(string version)
  {
    switch (version)
    {
      case "2014":
        return StartRevitProcess("2014");
        break;

      case "2015":
        return StartRevitProcess("2015");
        break;

      default:
        return false;
        break;
  }
}
```

5 - The StartRevit® method in ClientServiceContract.cs

```
private bool StartRevitProcess(string version)
{
  string revitExe = string.Format(
                        @"{0}C:\Program Files\Autodesk\Revit {1}\Revit.exe{0}",
                        DoubleQuote,
                        version);
  ProcessStartInfo startInfo = new ProcessStartInfo(revitExe, " /nosplash");
  startInfo.UseShellExecute = false;
  startInfo.RedirectStandardError = false;
  if (RevitClientData == null)
  {
    RevitClientData = new RevitClientDataContract(Process.Start(startInfo));
    RevitClientData.ApplicationProcess.Exited += RevitApplicationProcess_Exited;
  }
  Debug.Print("Revit " + version + " has started!");
  return true;
}
```

6 - The StartRevit®Process method in ClientServiceContract.cs

The Job Data

Job data was briefly mentioned above. It is transmitted from the server to the client. The file is a formatted XML document that has the basic schema shown below.

```
<Job>
  <!—Job elements go here -->
  <JobFiles>
    <JobFile>
      <!—JobFile elements go here -->
    </JobFile>
    <JobFile>
      <!—JobFile elements go here -->
    </JobFile>
  </JobFiles>
</Job>
```

7 - See Full Example in Appendix B

The Job section contains XML elements that hold data about the job. In our company's case, we have the project number, description, database name, dll file location and full class name of a class holding the command being run on each Revit® file in the job. The JobFiles Element holds one or more JobFile elements describing each model being processed.

Supporting class are used to hold this data; Job.cs and JobFile.cs are included in the class files. The job object is initialized with the XML data for use in the code.

```
Job job = new Job(GlobalSettings.RemoteJobXmlData);
```

8 - Instantiating the Job object in IdlingHandler.cs

Opening a Model

Once Revit® is started, a model needs to be opened for processing. Version 1.0 uses journal scripts to start Revit® and open a model. That approach would not work in Version 2. A new way had to be found. Fortunately, the Revit® API has some features that will help us with this. The first is the UIApplication.OpenAndActivateDocument method. This method can be called to open a model and make that model the active document in Revit®. It provides the same functionality as a user manually opening a model through the user interface. The Revit® API also provides the Application.OpenDocumentFile method to open documents, but this will not work for the type of processing we need to do. In my experience the document needs to be active for the graphics engine to regenerate each sheet when printing to PDF or exporting DWG files.

The second feature is the UIApplication.Idling event. This event is fired when Revit® is not processing any user initiated commands. As soon as the Revit® application opens without an active document it fires the Idling event handler. In the Revit® AddIns an idling event is created in the OnStartup method of a class that implements the IExternalApplication interface. The event is added as follows:

```
RevitUiControlledApp.Idling += IdlingHandler.RemoteJobHandler;
```

9 - Initializing the Idling Event

Code in the IdlingHandler.RemoteJobHandler method is run when the idling event is raised. The code below sets up various options for opening a model based on the data from the Job file. It then calls the OpenAndActrivateDocument method to open the model defined in the XML Job file.

```
WorksetConfiguration workSetConfiguration = new
WorksetConfiguration(WorksetConfigurationOption.CloseAllWorksets);
try
{
  if (job.OpenAllWorksets)
  {
    workSetConfiguration = new
WorksetConfiguration(WorksetConfigurationOption.OpenAllWorksets);
  }

  OpenOptions openOptions = new OpenOptions();
  openOptions.SetOpenWorksetsConfiguration(workSetConfiguration);
  openOptions.DetachFromCentralOption = DetachFromCentralOption.DoNotDetach;
  if (job.DetachFromCentral)
  {
    openOptions.DetachFromCentralOption =
DetachFromCentralOption.DetachAndDiscardWorksets;
    if (job.PreserveWorksets)
    {
      openOptions.DetachFromCentralOption =
DetachFromCentralOption.DetachAndPreserveWorksets;
    }
  }

  ModelPath modelPath =
ModelPathUtils.ConvertUserVisiblePathToModelPath(item.RevitFileName);
  UIDocument uiDocument =
GlobalSettings.RevitUiApp.OpenAndActivateDocument(modelPath, openOptions, false);
```

10 - Code to open a document in IdlingHandler.cs

Load an assembly based on XML data and execute the code on the open Revit® model

Once the model is open, code needs to be run based on the information in the XML job file. This code will come from an assembly created for this purpose. The XML job file contains two specific pieces of data used to implement this feature.

Loading the Assembly

The first piece of data is the DllFileName. This is the .NET assembly that contains code used to process the Revit® model. The second piece of data is the ClassName. It is the full namespace path of the class in the assembly that contains a remote command. In the code shown below, the LoadDllWithReflection method takes two arguments; the DllFileName and the ClassName. It uses the DllFileName to load the assembly with the Assembly.LoadFrom method. The ClassName type is then loaded using the Assembly.GetType method and returned.

```
public static Type LoadDllWithReflection(string dllLocation, string className)
{
  try
  {
    Assembly assembly = Assembly.LoadFrom(dllLocation);
    Type type = assembly.GetType(className);
    return type;
  }
  catch (ReflectionTypeLoadException ex)
  {
    var exceptions = ex.LoaderExceptions;
    foreach (var failedType in ex.Types)
    {
      if (failedType != null)
        Debug.WriteLine(failedType.FullName);
    }
    foreach (Exception loadException in exceptions)
    {
      Debug.WriteLine(loadException.ToString());
    }
  }
  return null;
}
```

11 - The LoadDllWithReflection method in IdlingHandler.cs

Get and run the Command

After the type is returned the command defined in the type needs to be executed. A remote command implements the IRemoteCommand interface. It contains one method definition; RunRemotely. The GetCommandFromType method is used to find the RunRemotely implementation in the type. The Activator.CreateInstance is called to create an instance of the type with the implemented RunRemotely method.

```
public static IRemoteCommand GetCommandFromType(Type type)
{
  IRemoteCommand command = Activator.CreateInstance(type) as IRemoteCommand;
  if (command == null)
    throw new Exception("Could not get Remote Command");
   return command;
}
```

12 - The GetCommandFromType method in IdlingHandler.cs

The returned type is a valid instance of the class defined by ClassName. The code below shows how the RunRemotely method is run after a valid instance is created.

```
Type type = LoadDllWithReflection(dllLocation, className);
IRemoteCommand command = GetCommandFromType(type);
command.RunRemotely(uiDocument.Document, item);
```

13 - LoadDllWithReflection and GetCommandFromType usage in IdlingHandler.cs

Suppress Revit® dialog boxes and error/warning messages

One of the main hurdles when automating Revit® is the user interface. As mentioned above, Revit® was not initially designed for automation. Over the years, the Revit® API development team has added functionality to help developers react to GUI events. There are two methods for handling the Revit® user interface; the ControlledApplication.FailuresProcessing event and the UIApplication.DialogBoxShowing event. Both of these events were added as part of the Revit® 2010 API; there was an OnDialogBox event in earlier versions I am unfamiliar with.

Handling Warnings and Errors

The Revit® API provides the FailuresProcessing API for handling warnings and errors. This is a global failure event and will process all warnings and errors during a Revit® session. A handler for the event is added in the OnStartup event handler for the Revit® Application and removed in the OnShutdown event handler.

```
public Result OnStartup(UIControlledApplication application)
{
  GlobalSettings.RevitControlledApp.FailuresProcessing +=
FailuresProcessingHandler.Handler;
}

public Result OnShutdown(UIControlledApplication application)
{
  GlobalSettings.RevitControlledApp.FailuresProcessing -=
FailuresProcessingHandler.Handler;
}
```

14 - Adding and removing the FailuresProcessing event handler

Failures are categorized using the four levels in the FailureSeverity enumeration. The categories are listed in order from least severe to most severe.

- FailureSeverity.None – No failures in the document
- FailureSeverity.Warning – Warnings can be ignored
- FailureSeverity.Error – Failure that cannot be ignored
- FailureSeverity.DocumentCorruption – Failure that forces transaction rollback

For version 1.0 and 2.0 of our application we are not making changes to documents and committing those changes. We are only concerned with the middle two categories; FailureSeverity.Warning and FailureSeverity.Error.

Initializing FailuresProcessing

Before we can begin to handle warning and error failure messages we need to initialize the FailuresProcessing system. The FailuresAccessor class is the gateway to all failure information.

```
public static void Handler(object sender, FailuresProcessingEventArgs e)
{
  FailuresAccessor failuresAccessor = e.GetFailuresAccessor();
  if (failuresAccessor == null)
  {
    return;
  }

  IList<FailureMessageAccessor> failureMessages =
failuresAccessor.GetFailureMessages();
  if ((failureMessages == null) || (failureMessages.Count == 0))
  {
    e.SetProcessingResult(FailureProcessingResult.Continue);
    return;
  }
  // more code
}
```

15 - Initializing the FailuresProcessing system

In the code above a FailuresAccessor object is instantiated using the FailuresProcessingEventArgs object and checked to make sure it is valid. A list of all failure messages needs to be loaded using the FailureMessageAccessor object. This information is returned as a IList<FailureMessageAccessor> list of FailureMessageAccessor objects from the FailuresAccessor.GetFailureMessages() method. If the list is null or has zero objects no more processing is necessary. The failure processing result is set to FailureProcessingResult.Continue and the handler is exited.

Warnings

Warnings are failures that can be ignored by the end user. This category of messages is very easy to handle.

```
public static void Handler(object sender, FailuresProcessingEventArgs e)
{
  // initialization code
  failuresAccessor.DeleteAllWarnings();
  // more code
}
```

16 - Bypass all warning message by deleting them

The FailureAccessor.DeleteAllWarnings method prevents any warning message from being displayed in the user interface.

Errors

Errors are more complicated than disabling warnings. Two steps can be taken to handle any remaining errors. The first step is to call the FailuresAccessor.ResolveFailures method.

```
public static void Handler(object sender, FailuresProcessingEventArgs e)
{
  // initialization code
  failuresAccessor.DeleteAllWarnings();
  // more code
}
```

17 - Resolve failures with the FailuresAccessor.ResolveFailures method

The ResolveFailures method will try to resolve any unresolved failure by checking to see if a FailureResolution has been set for each failure. If a FailureResolution has been set, the method will resolve the failure with the preset resolution. If no FailureResolution has been set, the method will resolve the failure with the default FailureResolution. After calling this method, the FailureProcessingResult must be set to FailureProcessingResult.ProceedWithCommit in order to prevent errors from being shown in the user interface.

Remaining failure messages that can be handled are errors that require deleting elements. Each failure message needs to be checked to see if is in the FailureSeverity.Error category. The following code shows how each failure message is checked for elements that can be deleted.

```
foreach (FailureMessageAccessor failureMessageAccessor in failureMessages)
{
  try
  {
    if (failureMessageAccessor == null)
    {
      continue;
    }
    FailureSeverity failureSeverity = FailureSeverity.None;

    try
    {
      failureSeverity = failureMessageAccessor.GetSeverity();
    }
    catch (Exception ex)
    {
      failureSeverity = FailureSeverity.None;
    }

    if (failureSeverity == FailureSeverity.Error)
    {
      FailureResolutionType resolutionType =
failureMessageAccessor.GetCurrentResolutionType();
      if (failuresAccessor.IsElementsDeletionPermitted())
      {
        ICollection<ElementId> failingElementIds =
failureMessageAccessor.GetFailingElementIds();
        if ((failingElementIds != null) & (failingElementIds.Count > 0))
        {
          foreach (ElementId failingElementId in
failureMessageAccessor.GetFailingElementIds())
          {
            failedElementIds.Add(failingElementId);
          }
        }
      }
    }
  }
  catch (Exception ex)
  {
  // log error
  }
}
if (failedElementIds.Count > 0)
{
  failuresAccessor.DeleteElements(failedElementIds);
}
e.SetProcessingResult(FailureProcessingResult.ProceedWithCommit);
```

18 - Delete elements to handle remaining failures

If the FailureMessageAccessor is an error, the FailureAccessor is checked to see if elements can be deleted by calling the FailuresAccessor.IsElementsDeletionPermitted. If elements can be deleted, the ElementId for each failed element is collected from each FailureMessageAccessor by calling the FailureMessageAccessor.GetFailingElementIds method. Each collected ElementId is added to a collection of ElementIds. If there are ElementIds in the collection, it is passed to the FailuresAccessor.DeleteElements method. The FailureProcessingResult must be set to FailureProcessingResult.ProceedWithCommit which is done by calling the FailuresProcessingEventArgs.SetProcessingResult.

Handling Dialog Box Responses

As noted earlier, Revit® is driven by its GUI. While working with Revit® models dialog boxes are constantly being displayed to the user. These dialog boxes need to be handled in order to batch process models in unattended mode. The same information needs to be supplied to the dialog box that a user would enter in an interactive Revit® session. The Revit® API provides the UIApplication.DialogBoxShowing event to help us with this. The event handler is added and removed in the same manner as other API Events.

```
public Result OnStartup(UIControlledApplication application)
{
  GlobalSettings.RevitUiControlledApp.DialogBoxShowing +=
DismissDialogsHandler.Handler;
}

public Result OnShutdown(UIControlledApplication application)
{
  GlobalSettings.RevitUiControlledApp.DialogBoxShowing -=
DismissDialogsHandler.Handler;
}
```

19 - Adding and removing the DialogBoxShowing event handler

What Goes in the Handler Method

The Revit® API documentation states "*This event is raised when Revit® is just about to show a dialog box or a message box.*". This information is further defined by the documentation by telling us the DialogBoxShowingEventArgs object will be one of the following objects; a TaskDialogShowingEventArgs object to handle TaskDialogs and a MessageBoxShowingEventArgs object to handle MessageBoxes. The following code is used to capture both objects.

```
public static void Handler(object sender, DialogBoxShowingEventArgs
dialogBoxShowingEventArguments)
{
  TaskDialogShowingEventArgs taskDialogEventArguments =
dialogBoxShowingEventArguments as TaskDialogShowingEventArgs;
  TaskDialogId dialogId = TaskDialogId.Unknown;
  if (taskDialogEventArguments != null)
  {
    // code here
    HandleTaskDialog(dialogBoxShowingEventArguments, dialogId);
  }
  else
  {
    MessageBoxShowingEventArgs messageBoxEventArguments =
dialogBoxShowingEventArguments as MessageBoxShowingEventArgs;
    if (messageBoxEventArguments != null)
    {
      // Nothing here right now
    }
    else
    {
      // code here
      HandleTaskDialog(dialogBoxShowingEventArguments, dialogId);
    }
  }
}
```

20 - The Handler method in DismissDialogHandler.cs

TaskDialogs and MessageBoxes are dismissed by setting the DialogBoxShowingEvebtArgs.OverrideResult property with the appropriate values. Internally, the API invokes the button on the TaskDialog or MessageBox correcponding to the supplied override result. This simulates the user pressing the button in the GUI.

```
private static void HandleTaskDialog(
                              DialogBoxShowingEventArgs dialogBoxEventArguments,
                              TaskDialogId id)
{
  if (TaskDialogManager.Instance.ActiveDialogGroup == TaskDialogGroup.Unknown)
  {
    LogManager.LogMessage(
          string.Format("*** Opened normally by user: [{0}]",
                        Enum<TaskDialogId>.GetName((int)id)),
                          LogSeverityType.Debug);
    return;
  }
  int overrideResult = TaskDialogManager.Instance.GetOverrideResult(id);
  if (overrideResult < 1)
  {
    LogManager.LogMessage(
          string.Format("*** Opened normally by user: [{0}]",
                        Enum<TaskDialogId>.GetName((int)id)),
                          LogSeverityType.Debug);
    return;
  }
  LogManager.LogMessage(string.Format("*** Handling Task Dialog ID [{0}]",
                                      Enum<TaskDialogId>.GetName((int)id)),
                        LogSeverityType.Debug);
  dialogBoxEventArguments.OverrideResult(overrideResult);
}
```

21 - Handle DialogBoxes and MessageBoxes

Valid values are one of the values in the TaskDialogResult enumeration which is shown below using the JetBrains dotPeek decompiler. See Appendix D for a list of tool I use for information on the dotPeek application.

```
// Decompiled with JetBrains decompiler
// Type: Autodesk.Revit.UI.TaskDialogResult
// Assembly: RevitAPIUI, Version=2015.0.0.0, Culture=neutral, PublicKeyToken=null
// MVID: D7063A84-A5B4-4FF1-8579-AB6D1C6A6268
// Assembly location: C:\Program Files\Autodesk\Revit 2015\RevitAPIUI.dll

namespace Autodesk.Revit.UI
{
  /// <summary>
  /// Enum to specify the task dialog result.
  /// </summary>
  /// <since>2011</since>
  public enum TaskDialogResult
  {
    None = 0,
    Ok = 1,
    Cancel = 2,
    Retry = 4,
    Yes = 6,
    No = 7,
    Close = 8,
    CommandLink1 = 1001,
    CommandLink2 = 1002,
    CommandLink3 = 1003,
    CommandLink4 = 1004,
  }
}
```

22 - Autodesk.Revit®.UI.TaskDialogResult Enumeration

DismissDialog Helper Classes

On the surface, the process for dismissing TaskDialogs is straightforward. Autodesk documentation and samples provide a fairly good high level explanation of the process. Unfortunately the low level information needed about specific TaskDialogs is not provided. The following questions can be raised when using the DialogBoxShowing event in your applications.

1. Where are all the TaskDialogIds defined?
2. How many TaskDialogs does the Revit® application contain?
3. Does a specific TaskDialog have CommandLinks, CommandButtons or a mixture of both?
4. How can TaskDialogs be managed for different workflows in Revit®?

The following helper classes and enumerations will help answer these questions and provide a framework for managing TaskDialogs.

TaskDialogId Enumeration

The TaskDialogID enumeration contains values for TaskDialogIds based on the TaskDialogId returned by the Revit® API. Collecting these values was trial and error at first. Debug output was monitored and journal files were searched but new ids were always being found.

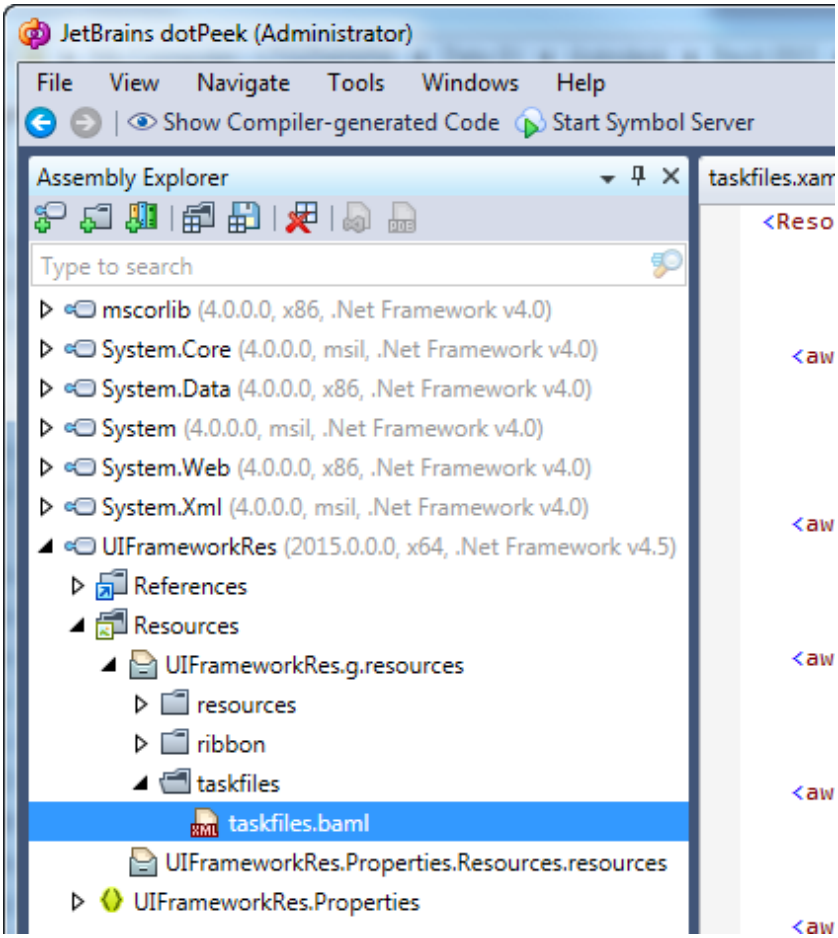
```
    /// <summary>
    /// The TaskDialog TaskDialog_Changes_Not_Saved.
    /// </summary>
    /// <remarks>
    /// <![CDATA[
    /// Jrn.Command "SystemMenu" ,
    /// "Quit the application; prompts to save projects , ID_APP_EXIT"
    /// ' 0:< TaskDialog "You have made changes to this file
    ///                   that have not been saved.  What do you want to do?"
    /// 'Id : TaskDialog_Changes_Not_Saved
    /// 'CommonButtons : Cancel
    /// 'Command Links:
    /// '1001 : Synchronize with central
    /// '1002 : Save locally
    /// '1003 : Do not save the project
    /// 'DefaultButton : 1001
    /// 'H 10-Oct-2014 10:37:39.918;   0:<
    /// Jrn.Data "TaskDialogResult"  _
    ///         , "You have made changes to this file
    ///           that have not been saved.  What do you want to do?",  _
    ///          "Do not save the project", "1003"
    /// ]]>
    /// </remarks>
    [Message("This TaskDialog is displayed when closing the local document that
has not been saved or synchronized with central.")]
    [Description("Changes Not Saved")]
    TaskDialog_Changes_Not_Saved,
```

23 - TaskDialog_Changes_Not_Saved TaskDialogId

The code above is the TaskDialogId.TaskDialog_Changes_Not_Saved enumeration value with remarks that include journal file output. The journal output tells us that CommandLink1 (value 1001) is the default value and that CommandLink3 was selected to close the TaskDialog.

TaskDialog Classes

After extensive research online I found that TaskDialogID values and other valuable information were contained in the UIFrameworkRes.dll resource assembly in the Revit® application folder. Exploring the assembly with a decompiler yielded the following information.



24 - The Decompiled UIFrameworkRes.dll

The section titled “taskfiles.baml” contained a list of TaskDialogIds along with XML formatted data about the TaskDialogs. This information is included in the class material as the files “TaskFiles.xaml.xml” and “TaslDialog Id List.txt”. The XML file contains the decompiled baml file and the text file contains a stripped down list of TaskDialogIds defined in the file. There is a 2014 and a 2015 version of these files.

This information is captured in the TaskDialogs.cs as a set of simple classes. The code below is the ChangesNotSaved class that pairs up with the TaskDialog_Changes_Not_Saved TaskDialogId.

```
/// <summary>
/// Class ChangesNotSaved.
/// </summary>
/// <remarks>
/// XML is from taskfiles.baml resouces in the UIFrameworkRes.dll
/// in the "C:\Program Files\Autodesk\Revit 2015" folder
/// Captured using:
/// JetBrains dotPeek 1.2
/// Build 1.2.1.226 on 2014-07-15T03:19:08
/// <![CDATA[
/// <aw:TaskDialog x:Key="TaskDialog_Changes_Not_Saved"
///                WindowTitle="Changes Not Saved"
///                MainInstruction="You have made changes to this file
///                                 that have not been saved.  What do you want to do?"
///                FooterText="&lt;a href=&quot;#&quot;&gt;
///                            Click here to learn more&lt;/a&gt;"
///                CommonButtons="Cancel"
///                EnableHyperlinks="true"
///                AllowDialogCancellation="true"
///                UseCommandLinks="true"
///                DefaultButton="1001"
///                Width="0"
///                WidthInXP="0"
///                x:Uid="TaskDialog_Changes_Not_Saved">
///   <aw:TaskDialog.Buttons>
///     <aw:TaskDialogButton ButtonId="1001"
///                          ButtonText="Synchronize with central: Synchronizes your
///                                      changes with the central model and allows
///                                      other users to view them."
///                          x:Uid="TaskDialog_Changes_Not_Saved_1001"/>
///     <aw:TaskDialogButton ButtonId="1002"
///                          ButtonText="Save locally: Saves your changes to your local
///                                      file but does not synchronize them with
///                                      the central model."
///                          x:Uid="TaskDialog_Changes_Not_Saved_1002"/>
///     <aw:TaskDialogButton ButtonId="1003"
///                          ButtonText="Do not save the project: Discards any unsaved
///                                      changes you made to the project."
///                          x:Uid="TaskDialog_Changes_Not_Saved_1003"/>
///   </aw:TaskDialog.Buttons>
/// </aw:TaskDialog>
/// ]]>
/// </remarks>
public class ChangesNotSaved : TaskDialogBase
{
  // OverrideResults
  public const int Cancel = (int)TaskDialogResult.Cancel;
  public const int SynchronizeWithCentral = (int)TaskDialogResult.CommandLink1;
  public const int SaveLocally = (int)TaskDialogResult.CommandLink2;
  public const int DoNotSaveTheProject = (int)TaskDialogResult.CommandLink3;

  public ChangesNotSaved(int overrideResult)
    : base(TaskDialogId.TaskDialog_Changes_Not_Saved)
  {
    OverrideResult = overrideResult;
  }
}
```

25 - The ChangesNotSaved Class with Comments

Structuring these classes in this manner allows easy identification of the allowed override results for the TaskDialog and multiple instances instantiated with different override results for different workflows.

TaskDialogGroup and TaskDialogGroups

A TaskDialog group is a set of TaskDialog classes placed together for use in specific workflows. The TaskDialogGroup holds enumerations for various workflows. The code below is a section of the TaskDialogGroup enumeration found in the “TaskDialogGroup.cs” file.

```
/// <summary>
/// This TaskDialog Group is used during the DocumentOpened event.
/// </summary>
[Message("This TaskDialog Group Type is used during the DocumentOpened event.")]
[Description("Document Opened")]
DocumentOpened,

/// <summary>
/// This TaskDialog Group is used during the DocumentOpening event.
/// </summary>
[Message("This TaskDialog Group Type is used during the DocumentOpening event.")]
[Description("Document Opening")]
DocumentOpening,

/// <summary>
/// This TaskDialog Group is used while processing a Remote job.
/// </summary>
[Message("This TaskDialog Group Type is used while processing a Remote job.")]
[Description("Process remote job without TaskDialogs displaying")]
RemoteJobProcessing
```

26 - A Section of the TaskDialogGroup Enumeration

The RemoteJobProcessing enumeration is used when running remote jobs. A TaskDialogGroup is designed as a Dictionary obkect that hold multiple other Dictionary objects with instantiated TaskDialog classes. The code below shows part of the RemoteJobProcessing group being built in the TaskDialogGroups class.

```
private Dictionary<TaskDialogId, TaskDialogBase> BuildRemoteJobProcessingGroup()
{
  Dictionary<TaskDialogId, TaskDialogBase> dialogGroup = new
Dictionary<TaskDialogId, TaskDialogBase>();
  // Defined TaskDialogBoxes (they have TaskDialogBoxId values)
  dialogGroup.Add(TaskDialogId.TaskDialog_Cannot_Create_Local_File, new
CannotCreateLocalFile(CannotCreateLocalFile.Close));
  dialogGroup.Add(TaskDialogId.TaskDialog_Cannot_Find_Central_Model, new
CannotFindCentralModel(CannotFindCentralModel.Close));
  dialogGroup.Add(TaskDialogId.TaskDialog_Cannot_Save_While_Message_Displayed, new
CannotSaveWhileMessageDisplayed(CannotSaveWhileMessageDisplayed.Close));
  dialogGroup.Add(TaskDialogId.TaskDialog_Changes_Not_Saved, new
ChangesNotSaved(ChangesNotSaved.DoNotSaveTheProject));
```

27 - A portion of the BuildRemoteJobProcessingGroup Method

The code for rhe TaskDialogGroups is building a dictionary based lookup system. Approximately a dozen TaskDialogGroups are defined for our use. Below is the TaskDialogGroups constructor.

```
public TaskDialogGroups()
{
  _dialogGroups = new Dictionary<TaskDialogGroup, Dictionary<TaskDialogId,
TaskDialogBase>>();
  _dialogGroups.Add(TaskDialogGroup.Unknown, BuildUnknownGroup());
  _dialogGroups.Add(TaskDialogGroup.DocumentCreated, BuildDocumentCreatedGroup());
  _dialogGroups.Add(TaskDialogGroup.DocumentOpened, BuildDocumentOpenedGroup());
  _dialogGroups.Add(TaskDialogGroup.DocumentOpening, BuildDocumentOpeningGroup());
  _dialogGroups.Add(TaskDialogGroup.DwgToDraftingView,
BuildDwgToDraftingViewGroup());
  _dialogGroups.Add(TaskDialogGroup.LoadFamilyCancel,
BuildLoadFamilyCancelGroup());
  _dialogGroups.Add(TaskDialogGroup.LoadFamilyDragDrop,
BuildLoadFamilyDragDropGroup());
  _dialogGroups.Add(TaskDialogGroup.LoadFamilyOverwrite,
BuildLoadFamilyOverwriteGroup());
  _dialogGroups.Add(TaskDialogGroup.LoadFamilyOverwriteParameters,
BuildLoadFamilyOverwriteParametersGroup());
  _dialogGroups.Add(TaskDialogGroup.RemoteJobProcessing,
BuildRemoteJobProcessingGroup());
  _dialogGroups.Add(TaskDialogGroup.UpgradeContent, BuildUpgradeContentGroup());
  _dialogGroups.Add(TaskDialogGroup.ViewPrinting, BuildViewPrintingGroup());
}
```

28 - The TaskDialogGroups Constructor in TaskDialogGroups.cs

This constructor is called once at the beginning of each Revit® session and initializes the TaskDialogGroups for use during the rest of the session.

TaskDialogManager Ties It All Together

The TaskDialogManager class is a singleton class providing an easy to use interface for the functionality described above. It has three main functions; initialize all the TaskDialogGroups, activate or deactivate a TaskDialogGroup at runtime and get the override result for a specific TaskDialogId in the active TaskDialogGroup.

```
TaskDialogManager()
{
  DialogGroups = new TaskDialogGroups();
  ActiveDialogGroup = TaskDialogGroup.Unknown;
  ActiveTaskDialogGroup = DialogGroups.GetGroup(TaskDialogGroup.Unknown);
  PreviousDialogGroup = TaskDialogGroup.Unknown;
  PreviousTaskDialogGroup = DialogGroups.GetGroup(TaskDialogGroup.Unknown);
}
```

29 - The TaskDialogManager constructor in TaskDialogManager.cs

The TaskDialogGroups are initialized by calling the TaskDialogGroups constructor. Default values of Unknown are supplied for the ActiveDialogGroup property.

```
public void ActivateDialogGroup(TaskDialogGroup group)
{
  if (ActiveDialogGroup == TaskDialogGroup.RemoteJobProcessing)
  {
    PreviousDialogGroup = ActiveDialogGroup;
    PreviousTaskDialogGroup = ActiveTaskDialogGroup;
    return;
  }
  LogManager.LogMessage(
          string.Format("TaskDialogManager.ActivateDialogGroup - group = {0}",
                        group));
  LogManager.LogMessage(
          string.Format("PreviousDialogGroup = {0}", PreviousDialogGroup));
  PreviousDialogGroup = ActiveDialogGroup;
  PreviousTaskDialogGroup = ActiveTaskDialogGroup;
  ActiveDialogGroup = group;
  ActiveTaskDialogGroup = DialogGroups.GetGroup(group);
}
```

30 - The ActivateDialogGroup method in TaskDialogManager.cs

The ActivateDialogGroup method sets the ActiveDialogGroup property to the TaskDialogGroup enumeration value passed into the method. It then looks up the TaskDialogGroup discionary from the set of TaskdialogGroups and sets the ActiveTaskDialogGroup property.

```
if (GlobalSettings.DocumentProcessingMode == ProcessingMode.RemoteJob)
{
  TaskDialogManager.Instance.ActivateDialogGroup(
                                    TaskDialogGroup.RemoteJobProcessing);
}
```

31 - ActivateDialogGroup usage

In the code above the ActivateDialogGroup method activates the RemoteJobProcessing TaskDialogGroup if a remote job is being run. When handling a TaskDialog the HandleTaskDialog method discussed earlier calls the GetOverrideResult method using the TaskDialogManager.

```
public int GetOverrideResult(TaskDialogId id)
{
  TaskDialogBase taskDialogInstance = null;
  if (ActiveTaskDialogGroup.TryGetValue(id, out taskDialogInstance))
  {
    return taskDialogInstance.OverrideResult;
  }
  return (int)TaskDialogId.Unknown;
}
```

32 - The GetOverrideResult method in TaskDialogManager.cs

The method uses the TaskDialogId as the key to lookup the TaskDialog class instantiated in the dictionary.

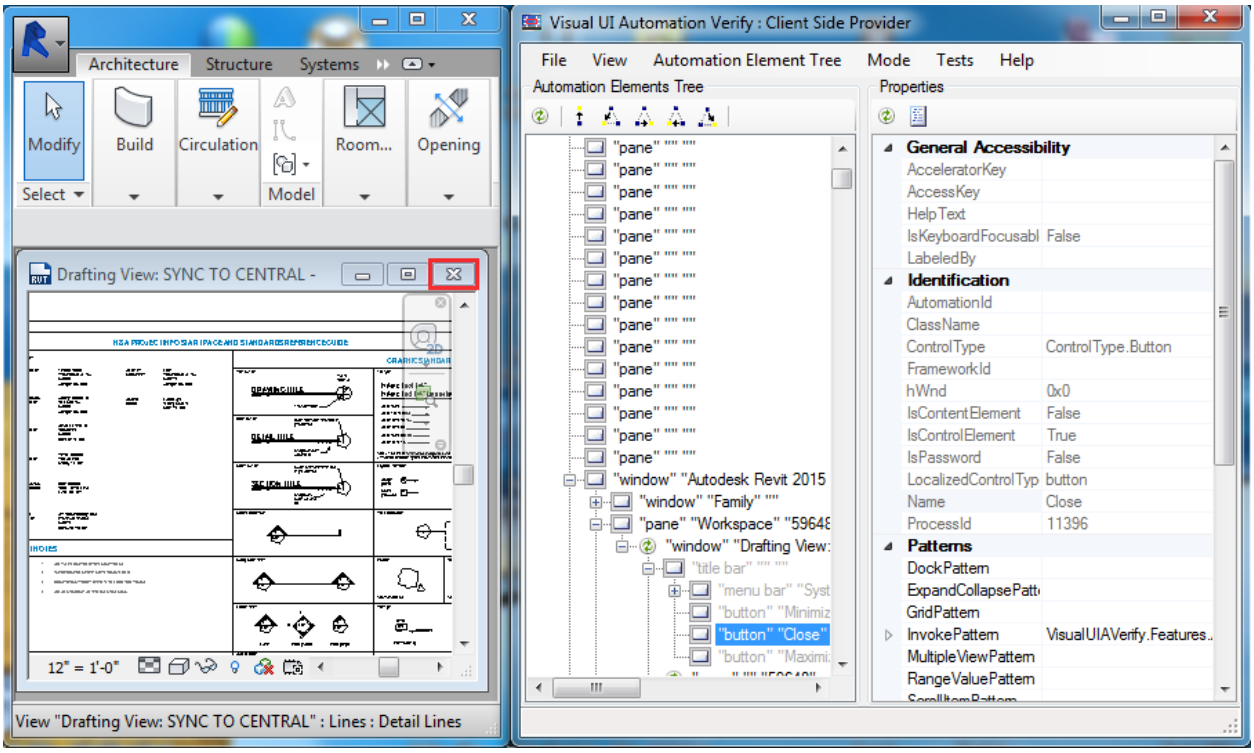
Close the open Revit® model and exit Revit®

In order to process more than one model, we need to be able to open a series of models. This can be accomplished in two ways. The first is to start the Revit® application, open a model, process the model and then close the Revit® application. Version 1.0 of our solution uses this approach by writing out a small journal file, starting Revit® with the journal file, running the command in the model and closing the Revit® application. Version 2.0 of our solution uses the UIApplication.OpenAndActivateDocument method. The Revit® API currently does not support switching between two documents so only one document can be open and active.

Closing the active document

This is one area we need to move outside the boundary of the Revit® API and use undocumented functionality to close a document. The solution we chose was to use the UIAutomation library provided by Microsoft with the .NET Framework. This library is provided as a means of creating automated testing scripts but we use it to simulate user interaction with the Revit® user interface.

Using the Inspect tool and/or the Visual UAI Verify tool (see Appendix D) from the Windows SDKs, it is possible to locate and interact with controls in the Revit® GUI. The image below shows the Revit® 2015 application side by side with the Visual UIA Verify application.



33 - The Visual UIA Verify hierarchy for closing the active document

The Visual UIA Verify application is highlighting the Close button in the Revit® user interface for the active document and showing the full tree of AutomationElements from the main Revit® application window down to the Close button in the open drafting view. For clarity, the tree looks like this:

- “window” “Autodesk Revit 2015” (the main application window)
    - “pane” “workspace” (the panel where open/active views are displayed)
        - “window” “Drafting View” (the window of the open, active drafting view)
            - “title bar” “” (the title bar of the open window, grayed out)
                - “button” “Close” (the close button, grayed out)

The drafting view is the only view open in the Revit® session. When a user presses the close button in the only open view the active document is closed. The following code is executed to close the active document.

```
private void CloseDocument()
{
  RevitUI.ApplicationWindowElement.SetFocus();
  if (RevitUI.Instance.WorkspaceElement == null)
  {
    return;
  }

  TreeWalker walker = new
TreeWalker(System.Windows.Automation.Automation.RawViewCondition);
  AutomationElement windowElement =
walker.GetFirstChild(RevitUI.Instance.WorkspaceElement);
  if (windowElement == null)
  {
    return;
  }

  AutomationElement titleBarElement = walker.GetFirstChild(windowElement);
  while (titleBarElement != null)
  {
    if (titleBarElement.Current.LocalizedControlType.ToUpper().Equals("TITLE
BAR"))
    {
      break;
    }
    titleBarElement = walker.GetNextSibling(titleBarElement);
  }
  if (titleBarElement == null)
  {
    return;
  }

  AutomationElement closeButtonElement = walker.GetFirstChild(titleBarElement);
  while (closeButtonElement != null)
  {
    if (closeButtonElement.Current.AutomationId.ToUpper().Equals("CLOSE"))
    {
      break;
    }
    closeButtonElement = walker.GetNextSibling(closeButtonElement);
  }
  if (closeButtonElement == null)
  {
    return;
  }

  RevitUI.InvokeButton(closeButtonElement);
}
```

34 - The CloseDocument method in CloseDocumentCommand.cs

The CloseDocument method uses two references to AutomationElement object handled in the RevitUI class (RevitUI.cs). The ApplicationWindowElement is the main Revit® window (the first item in the list above) and the WorkspaceElement is the pane containing any open view windows (the second item in the list above. There are several ways to locate child elements in a tree of elements. Three options are available for filtering children AutomationElements. Each provides a different view of its child elements.

- Automation.ContentViewCondition – All child elements can contain content
- Automation.ControlViewCondition – All child elements are controls
- Automation.RawViewCondition – All child elements are visible (no filter)

The code above initializes the TreeWalker object with the Automation.RawViewCondition because the “title bar” element in the tree is grayed out and is not available using the other two filters. The TreeWalker object finds each successive child beneath it until it reached the close button. The RevitUI.InvokeButton helper method is used to invoke the button.

```
public static void InvokeButton(AutomationElement buttonElement)
{
  if (buttonElement == null)
  {
    return;
  }
  try
  {
    Object currentInvokePattern;
    if (buttonElement.TryGetCurrentPattern(InvokePattern.Pattern, out
currentInvokePattern))
    {
      InvokePattern invokePattern = currentInvokePattern as InvokePattern;
      if (invokePattern == null)
      {
        return;
      }
      invokePattern.Invoke();
    }
  }
  catch (Exception ex)
  {
    // Log the exception
  }
}
```

35 - The RevUI.InvokeButton method in RevitUI.cs

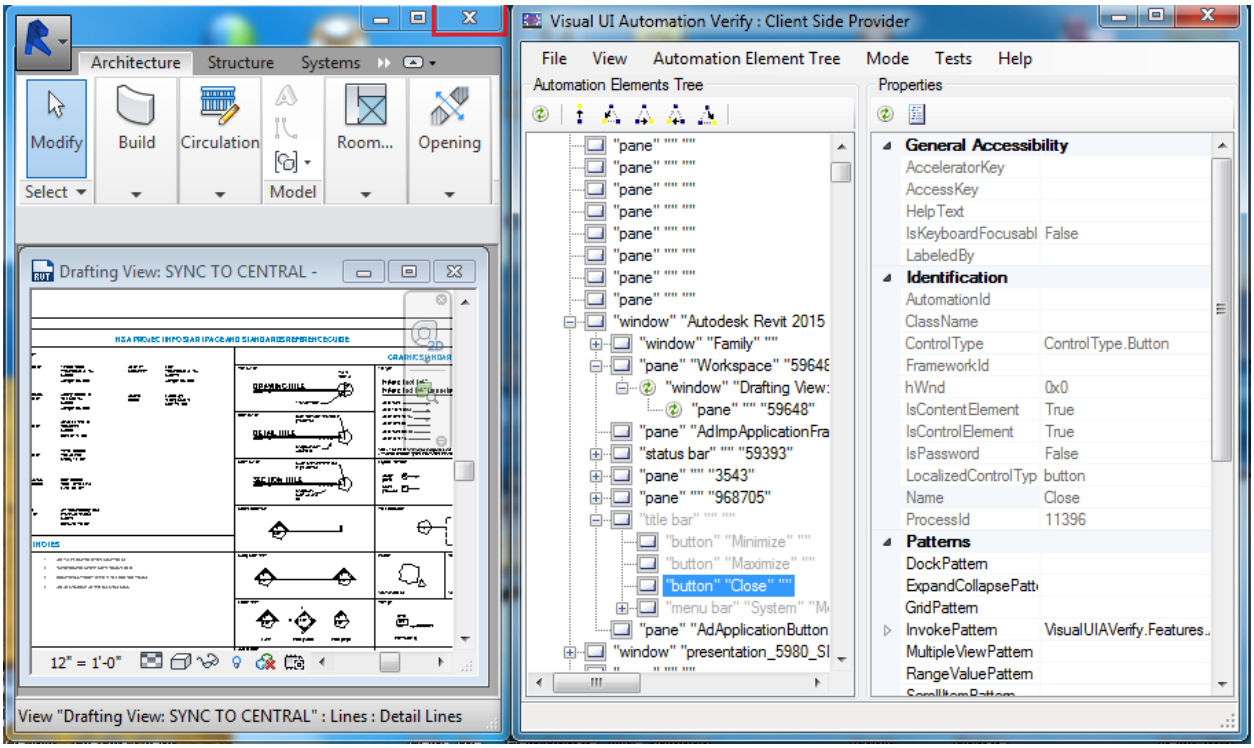
The InvokeButton method makes sure the button element is valid. If the TryGetCurrentPattern method returns true and the object it returns can be cast to a valid InvokePattern object the button is invoked using the Invoke method of the instantiated InvokePattern. The Invoke method mimics a user clicking the close button in the interactive user interface.

Closing the Revit® application

In order to process multiple jobs in series, we need to be able to gracefully close the Revit® application after a job is completed. There are multiple options available for closing the Revit® application.

- Use the WIN32 API to close the main application window
- Use the Process class to kill the application
- Use UIAutomation

UIAutomation is the most graceful option since it closes the application in the same manner as an interactive user. Closing the application using UIAutomation is similar to closing the active document but less complex. The image below is from the Visual UIA Verify tool we used above.



36 - The Visual UIA Verify hierarchy for closing the Revit® application

The Visual UIA Verify application is highlighting the Close button in the Revit® user interface for the main application window and showing the full tree of AutomationElements from the main application window down to the Close button in the window's title bar. For clarity, the tree looks like this:

- "window" "Autodesk Revit 2015" (the main application window)
  - "title bar" "" (the title bar of the open window, grayed out)
    - "button" "Close" (the close button, grayed out)

```
private void CloseApplication()
{
  RevitUI.ApplicationWindowElement.SetFocus();

  TreeWalker walker = new
TreeWalker(System.Windows.Automation.Automation.RawViewCondition);

  AutomationElement titleBarElement =
walker.GetFirstChild(RevitUI.ApplicationWindowElement);
  while (titleBarElement != null)
  {
    if (titleBarElement.Current.LocalizedControlType.ToUpper().Equals("TITLE
BAR"))
    {
      break;
    }
    titleBarElement = walker.GetNextSibling(titleBarElement);
  }
  if (titleBarElement == null)
  {
    return;
  }

  AutomationElement closeButtonElement = walker.GetFirstChild(titleBarElement);
  while (closeButtonElement != null)
  {
    if (closeButtonElement.Current.AutomationId.ToUpper().Equals("CLOSE"))
    {
      break;
    }
    closeButtonElement = walker.GetNextSibling(closeButtonElement);
  }
  if (closeButtonElement == null)
  {
    return;
  }

   RevitUI.InvokeButton(closeButtonElement);
}
```

37 - The CloseApplication method in CloseApplicationCommand.cs

The CloseApplication method uses the same logic to find the Close button and run the Invoke method on the control. The code is simpler because there are two less controls in the path to the title bar element.

Conclusion

Automating Revit® in the manner presented in this handout is not a simple undertaking. Trial and error was used in early development to learn how to handle the Revit® user interface and error processing. The commands to process the model were written, tested, tweaked and

rewritten before they functioned well. Each release of Revit® introduced new functionality and deprecated code that needed to be addressed. Version 1.0 of our system required some manual intervention at times but still saved the company time and money. It has been a worthwhile endeavor to get version 2.0 working. It has reduced some of the manual work and helped improve our company's work processes.

Reviewing the learning objectives we can now start the Revit® application from a remote server and open a Revit® model. The client/server portion is fairly routine and the Revit® API provides us with robust options for opening a model. Using standard .NET Framework functionality, we can load an assembly with our external commands and run them in the open Revit® model. We abide by the rules the Revit® API enforces by running the commands from within the Idling event handler. We have learned how to handle warning and errors by using the FailuresProcessing API. A robust system of helper classes make handling TaskDialogs relatively easy. We have learned where TaskDialog data is stored and how we can access it in future versions of Revit®. We learned how to use a couple of tools to explore the inner workings of the Revit® user interface. Although we have to use this undocumented functionality, we learned how to close the active Revit® document in order to process more than one model. We learned how to use the same functionality to close the Revit® application.

Using the provided class files, you now have the ability to experiment with this entire process and automate Revit® for your own needs. Happy automating!!

Appendices

Appendix A – List of Figures

Appendix B – A complete job file for a smaller project

```
<?xml version="1.0" encoding="utf-8"?>
<Job xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ProjectNumber>973400</ProjectNumber>
  <Description>DWG Export - 973400 Description</Description>
  <Version>2015</Version>
  <DllFileName>C:\Program Files\Hankins &amp;amp; Anderson, Inc\H&amp;amp;A Revit 2015
AddIns\HA.Commands.dll</DllFileName>
  <CommandName>HA.Commands.DwgExport.Command</CommandName>
  <OutputFolder>P:\Projects\973400\CADD\05 Batch Processing</OutputFolder>
  <EstimatedRunTimeInMinutes>15</EstimatedRunTimeInMinutes>
  <DetachFromCentral>true</DetachFromCentral>
  <PreserveWorksets>false</PreserveWorksets>
  <OpenAllWorksets>true</OpenAllWorksets>
  <LogFailureWarnings>true</LogFailureWarnings>
  <LogFailureErrors>true</LogFailureErrors>
  <MasterDatabaseName>SheetData_973400</MasterDatabaseName>
  <JobFiles>
    <JobFile>
      <Name>973400-CADLINKS15-Central</Name>
      <Description>DWG Export - 973400-CADLINKS15-Central</Description>
      <FileName>P:\Projects\973400\CADD\Revit 2015\973400-CADLINKS15-Central.rvt</FileName>
      <Identifier>b6e8255a-8bca-485b-adb0-2fe6b197ee87</Identifier>
      <ProcessingOrder>1</ProcessingOrder>
    </JobFile>
    <JobFile>
      <Name>973400-M15-Central</Name>
      <Description>DWG Export - 973400-M15-Central</Description>
      <FileName>P:\Projects\973400\CADD\Revit 2015\973400-M15-Central.rvt</FileName>
      <Identifier>a31e55ff-47c5-4303-a469-47ff15fe1517</Identifier>
      <ProcessingOrder>2</ProcessingOrder>
    </JobFile>
    …
  </JobFiles>
</Job>
```

38 - See Job.xml in the Class Materials

Appendix C – References

Rod Howarth's original blog reference to his Autodesk University 2011 virtual presentation. His virtual presentation does not appear to be available on the Autodesk University website any longer.

http://blog.rodhowarth.com/2011/09/Revit-autodesk-university-2011-virtual.html

Jeremy Tammik's blog, the Building Coder has provided many useful tidbits over the years. Specific to this presentation are articles on the FailuresProcessing API and the DialogBoxShowing event. The SDK examples and RevitLookup tool Jeremy maintains are invaluable.
http://thebuildingcoder.typepad.com/

Harry Mattison's Boost Your Bim blog contains many useful code snippets covering a wide range of Revit® API topics.
https://boostyourbim.wordpress.com/

Appendix D – Tools Used

Appendix D is a listing of the various tools I use in my development

JetBrains dotPeek assembly explorer and decompiler is very useful for exploring any .NET assembly. A great way to learn about the Revit® API is by exploring the RevitAPI and the RevitAPIUI assemblies.

http://www.jetbrains.com/decompiler/

Red Gate Software's .NET Reflector is a tool similar to dotPeek. It is more mature and includes integration with Visual Studio.

http://www.red-gate.com/products/dotnet-development/reflector/

The Visual UIA Verify tool is shipped as part of the Microsoft Windows SDK. The latest version is the Windows 8.1 SDK. On my system the Visual UIA Verify application is in this folder: C:\Program Files (x86)\Windows Kits\8.1\bin\x64\UIAVerify

http://msdn.microsoft.com/en-us/windows/desktop/bg162891.aspx

The Inspect tool is shipped as part of the Microsoft Windows SDK. The latest version is the Windows 8.1 SDK. On my system the Inspect application is in this folder: C:\Program Files (x86)\Windows Kits\8.1\bin\x64

http://msdn.microsoft.com/en-us/windows/desktop/bg162891.aspx

The Sysinternals tools provided by Microsoft have been part of my toolbox for many years. I use DebugView and Process Monitor almost daily.

http://technet.microsoft.com/en-us/sysinternals/bb545021.aspx

Appendix E – Class Files

Appendix E is a description of class files supporting this handout and class presentation. It will be updated prior to the beginning if AU 2014.