

AS319568-L

Do You Want to Build an Add-In?

John Pierson

Parallax Team, Inc.

NOTE: For the latest version of this handout. Please visit, [this link](#).

For other materials, please visit, <https://www.sixtysecondrevit.com/au2019/>

Magnificent Lab Assistants:

Jason Boehning, Marcello Sgambelluri, Carl Storms

Learning Objectives

- **Learn** about the Revit API.
- **Learn** how to set up Visual Studio for Revit add-in development.
- **Gain** access to valuable resources to build upon.
- **Learn** how to add your own functionality to Revit.

Description

This hands-on lab aims to provide an introduction to creating your own Revit add-ins using C#. This lab will not be an overly complex look at add-in creation from a computer programmer's point of view. Instead, it will introduce Visual Studio and C# from a Revit user's perspective. We'll learn how to set up Visual Studio templates to get started quickly, how to observe the Revit database to find out what we can add, and how to build simple functionality to make Revit do what we want. In addition to these, you'll gain insight into a ton of valuable resources to use as you continue your journey into the Revit API.

Speaker



John Pierson is a Design Technology Specialist at Parallax Team, a full-service Implementation firm. Specializing in the creation of custom tools and workflows for the AEC industry, John has become well known for his Dynamo-based solutions. Currently, John manages several Dynamo packages that are available to the Dynamo community. This includes Rhythm, Bang! and Monocle totaling over 105,000 downloads.

In addition to the creation of automated workflows and custom tools, John is a Revit certified professional for all disciplines. As a Revit certified professional, he provides custom content creation, project-based modelling, and Revit training.

John is an avid contributor to the Dynamo community, being a moderator on the Dynamo forum and a contributor to the Revit Forum. John is also a top speaker at many events including, Autodesk University, BiLT Conferences and various user groups everywhere. Find him at [sixtysecondrevit.com](https://www.sixtysecondrevit.com), [@60secondrevit](#) or on [parallaxteam.com](https://www.parallaxteam.com)

Introduction

This class is meant to be an introduction to Revit Add-Ins to get the average Revit user up and running as quickly as possible. I structured this class based on my own experiences hacking away at Revit Add-Ins until it worked for me. I wanted to give you some useful tips and comments I have picked up over the years.

There's a **slight** chance that...

- This class will get some terms incorrect. It happens to all of us. Revit Add-In Development is a pretty big topic to cover when you are a Revit user first.
- We will reference more refined resources for the next steps or for answers to our questions.
- We will venture into some “questionable” territory from the point-of-view of “real programmers”. Don't worry though, it's going to be okay. 😊

There's a **100%** chance that you...

- Will have fun learning about C# and gaining some insight to get started.
- Will make awesome stuff that can be used in Revit.
- Will learn at least one thing that you did not know before opening this handout.
- Will find out about great resources to venture down your Revit API Journey and grow.

Resources (from people way smarter than me)

- [Pushing Revit to the Next Level: An Intro to Revit Plugins with C# - Jeremy Graham](#)

Jeremy goes into real depth regarding how to work within C# and the Revit API. He defines tons of key terms and really breaks down some of the nuances when getting started from scratch. This class is a great next step when you want to learn more about the semantics of C#.

- [Revit API Resources – Danny Bentley](#)

Danny's YouTube is an amazing resource for next steps in your Revit Add-In journey. He has everything on there from sample workflows and great explanations of how visual studio works.

[Danny's YouTube](#) & [Danny's Twitter](#)

- [Untangling Python: A Crash Course on Dynamo's Python Node - Gui Talarico](#)

Gui's class is an amazing resource on using Python for Revit to its fullest. He covers deep Python concepts and provides info on how to utilize outside development environments. He is also an active contributor to the AEC community when it comes to Python and the API. His API resources are something I use on the daily.

<http://www.revitapidocs.com/> & <https://apidocs.co/>

Key Terms

Visual Studio

Visual studio is an IDE (integrated development environment) that allows us to author code to execute with Revit. Basically, a cool text editor that allows us to compile code. Visual studio is to code as Revit is to RVT files. For these exercises we are using Visual studio community which is free.

C#

C# is a general-purpose, modern and object-oriented programming language pronounced as “C sharp”.

Object Oriented Programming

C# is an object-oriented programming language. This means that all the items in C# are objects. Within objects we have properties and methods. Properties give us info about the object, while methods allow us to make the object perform a function. This can be compared to Revit in the sense that all items are objects. Looking at a wall element, we know that this has certain parameters (properties) and modifiers(methods).

Build

Building our code in visual studio means we are compiling it for use in Revit. Within this build process we can have other things happen like copying files around or cleaning up files.

IntelliSense

IntelliSense is a code-completion aid that includes several features: List Members, Parameter Info, Quick Info, and Complete Word. These features help you to learn more about the code you're using, keep track of the parameters you're typing, and add calls to properties and methods with only a few keystrokes. **Basically, this is autocompleting for our code. Which is awesome.**

What is an API?

Technically API stands for Application Programming Interface. This is a protocol to communicate with something, in our case the Revit API.

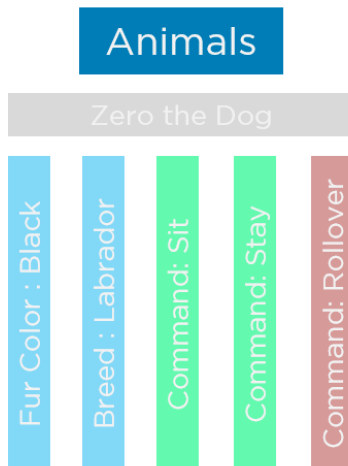
Simple Definition

A common language for us to communicate with the receiving party.

API of a Dog



Zero (my wife and I's dog)



Namespace: Animals is a category with classes.

Zero : A class in the animals namespace with it's own methods and properties.

Properties : Queries on the object

Method : An action on the object.

Invalid Method: An action on the object.

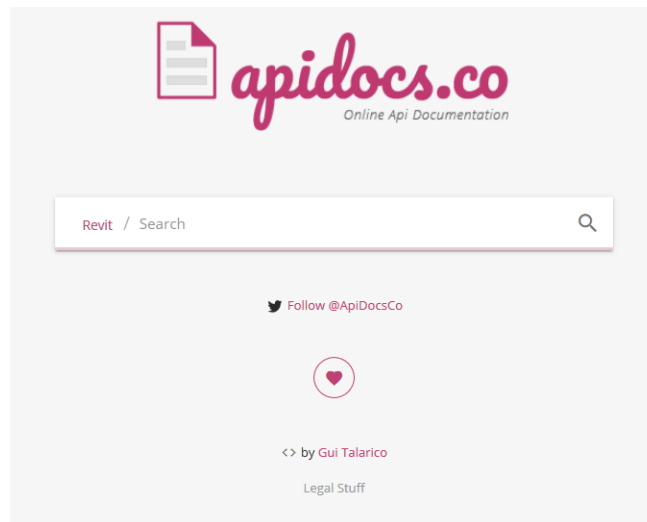
In this case, this API does not support that method, so it returns null.

Exploring the Revit API

Whenever a Revit user finds themselves wanting to add functionality to Revit, they often have an idea or goal in mind. Typically, you want to access or change data on a Revit element and simply could not find a node for it in other custom packages.

Find Properties and Methods of Revit Elements

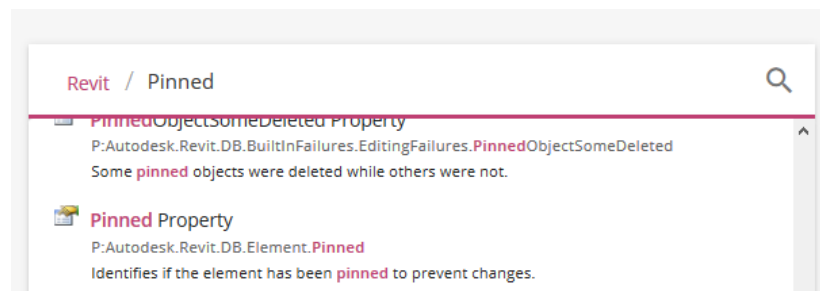
The absolute easiest method of finding out what you can do to a Revit element is to use the website, [APIDocs](https://apidocs.co). APIDocs is built by [Gui Talarico](https://github.com/GuiTalarico) and provides instant access to various API documentation for Revit, Rhino and Grasshopper. (API is Application Programming Interface and is a set of definitions for a program)



[Apidocs.co](https://apidocs.co) landing page.

Finding Element Properties

On APIDocs, you can search for properties based on what you want to achieve. In this example we want to access the **Pinned** property of elements with Python. Since this is possible using the UI, it should be possible through the API as well. Searching on API docs yields the following result.



We can see that Element has a property called pinned.

Setting up Visual Studio

There are a ton of valuable resources out there to get you started making Revit add-ins from scratch, but this class aims to demonstrate a simpler method.

Using Revit Add-In Templates

How do you normally start a Revit project? With a template? (I hope so). With visual studio it is no different. We can start from a pre-baked template to hit the ground running!

TheBuildingCoder's Revit Add-In Wizards

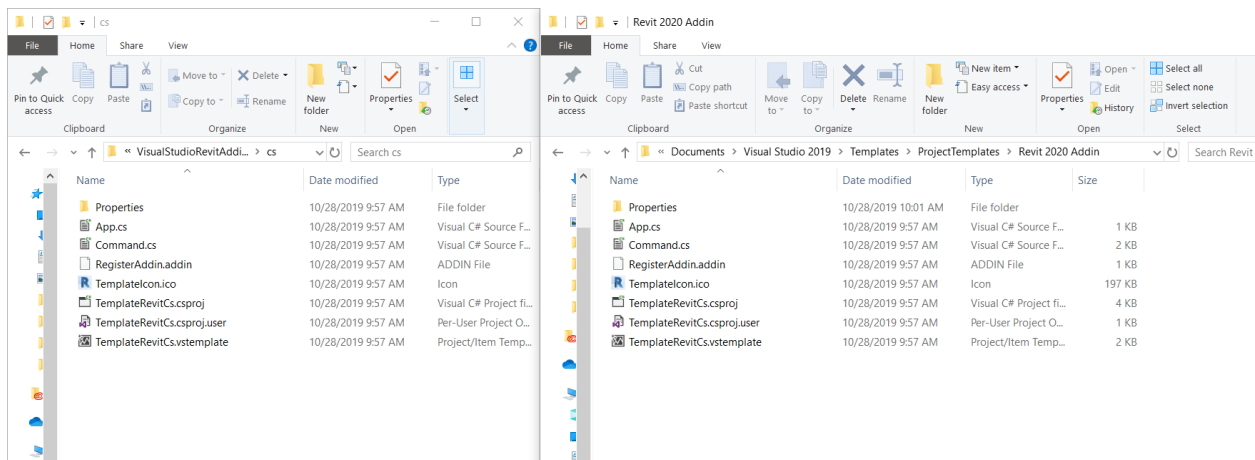
With each version of Revit comes a new template. Luckily, Jeremy Tammik (aka TheBuildingCoder) provides a new one each year!

This is hosted on GitHub at this address:

<https://github.com/jeremytammik/VisualStudioRevitAddinWizard>

Configuring the Template in Visual Studio

Simply download the latest release from the [releases section](#), and extract the contents.

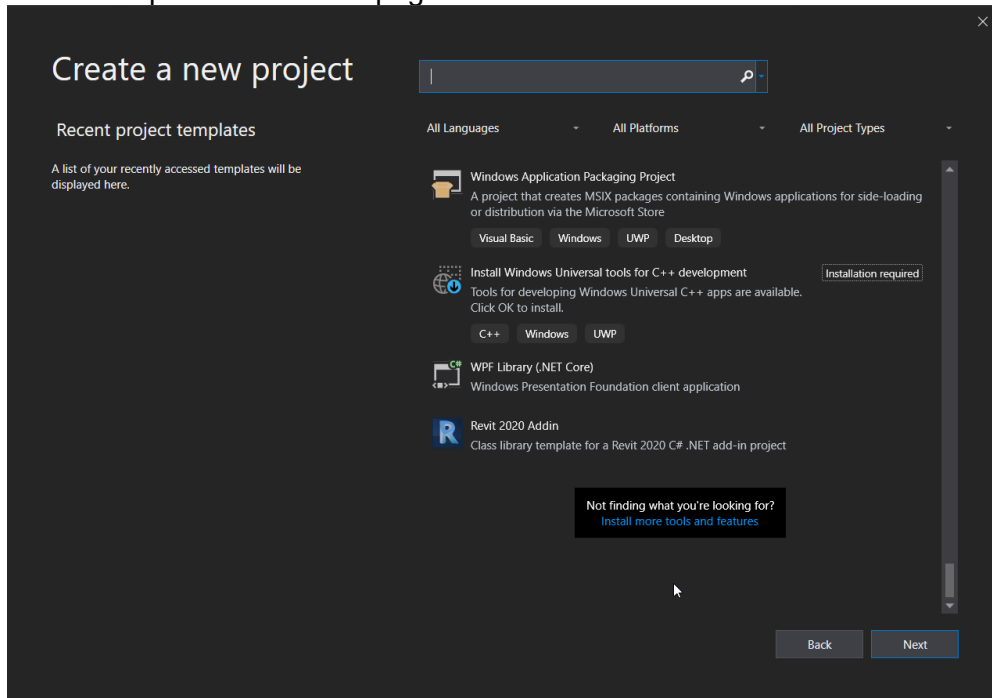


Extract into (Note: You will need to create the red directories):

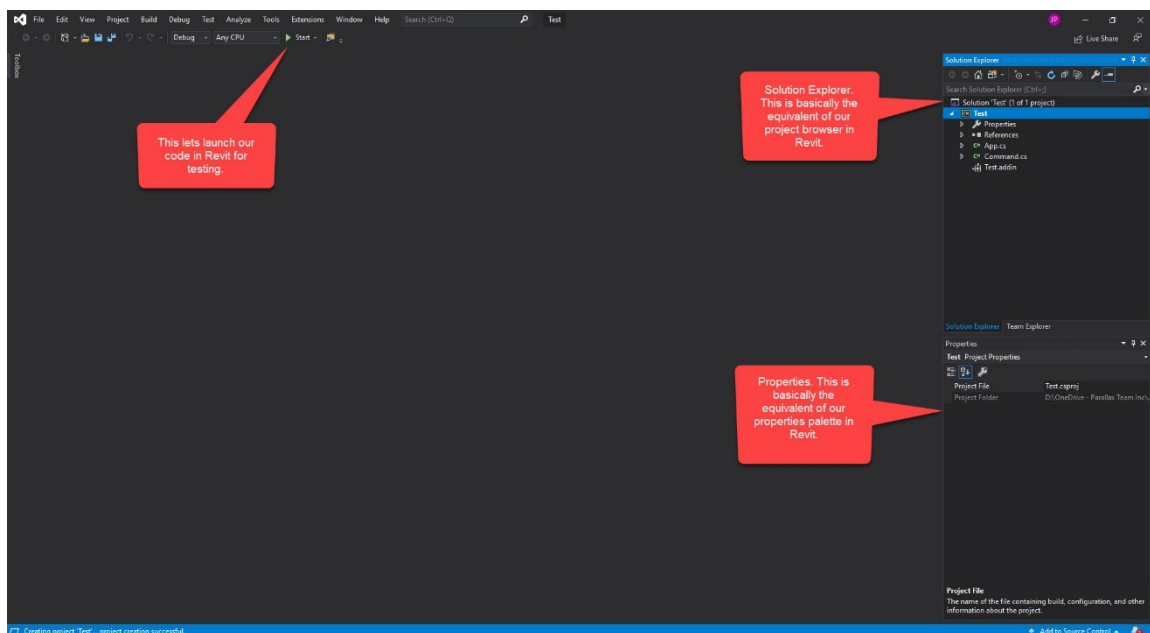
C:\Users\USERNAME\Documents\Visual Studio 2019\Templates\ProjectTemplates\Revit 2020 Addin

Using the Template

After extracting the files into the correct directory, you will then have the Revit 2020 Addin Template on the start page.

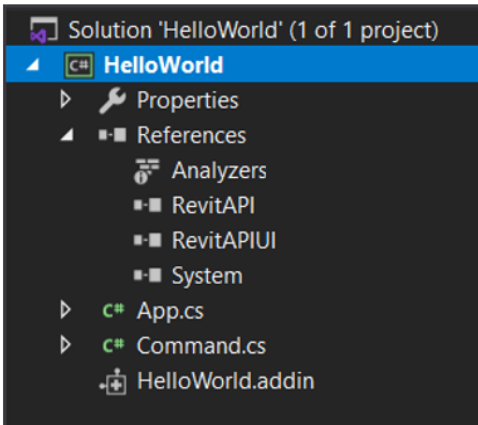


Visual Studio UI



01_Hello World

Once we create our new project, we can view the content in a similar way to Revit's project browser.



Solution: Our visual studio solution. This is like the directory that an RVT resides in.

Project: This is like our current Revit model when working in Revit

API References: Libraries that we can work with. This can be looked at like loading families in a Revit file.

Application-Level Actions: Triggers on Revit loading

Commands: Triggers on button push

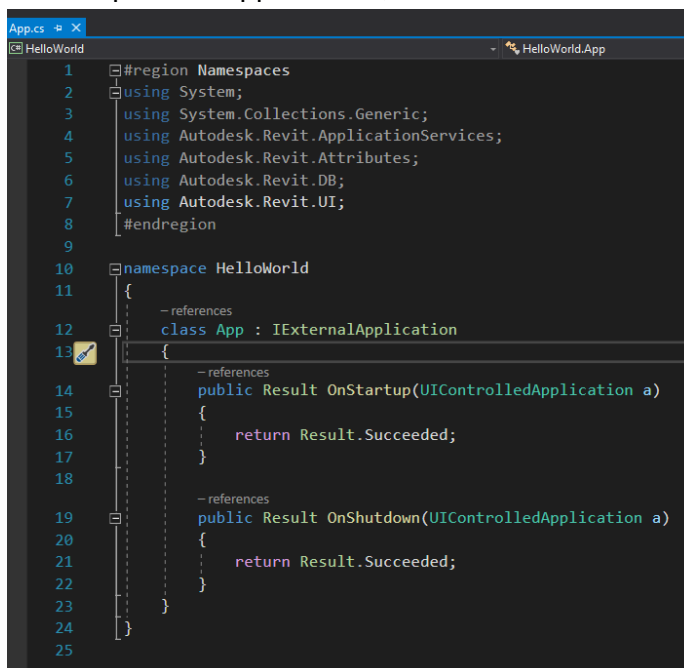
Addin Manifest: This tells Revit how to load our addin.

With the template, we can see that we have all the workings of a Revit add-in pre-baked for us in visual studio. Normally, we have to add App.cs, Command.cs, and HelloWorld.Addin ourselves.

App.cs (Application-Level Actions)

Within our visual studio project, we can now investigate some of the items that were generated from the template. First up is App.cs.

At the top of our App.cs code, we have a list of all the relevant **Namespaces** we are



using. Remember, these represent the libraries that we will have access to. This is like a family list in Revit. (light gray indicates not currently in use)

Within our code we have our own namespace of **HelloWorld**. Like the Revit API our add-in will also have namespaces and classes based on what we create.

Additionally, in the application portion of our add-in we have options to do stuff on startup and shutdown of Revit.

App.cs in 01_HelloWorld

Command.cs (Actions to be Triggered)

Within our visual studio project, we also have the command class where our code will live. Within this class we have a lot of pre-baked code that we won't be needing in this example.

```

1  #region Namespaces
2  using System;
3  using System.Collections.Generic;
4  using System.Diagnostics;
5  using Autodesk.Revit.ApplicationServices;
6  using Autodesk.Revit.Attributes;
7  using Autodesk.Revit.DB;
8  using Autodesk.Revit.UI;
9  using Autodesk.Revit.UI.Selection;
10 #endregion
11
12 namespace HelloWorld
13 {
14     [Transaction(TransactionMode.Manual)]
15     -references
16     public class Command : IExternalCommand
17     {
18         -references
19         public Result Execute(
20             ExternalCommandData commandData,
21             ref string message,
22             ElementSet elements)
23         {
24             UIApplication uiapp = commandData.Application;
25             UIDocument uidoc = uiapp.ActiveUIDocument;
26             Application app = uiapp.Application;
27             Document doc = uidoc.Document;
28
29             // Access current selection
30             Selection sel = uidoc.Selection;
31
32             // Retrieve elements from database
33             FilteredElementCollector col
34                 = new FilteredElementCollector(doc)
35                   .WhereElementIsNotElementType()
36                   .OfCategory(BuiltInCategory.INVALID)
37                   .OfClass(typeof(Wall));
38
39             // Filtered element collector is iterable
40             foreach (Element e in col)
41             {
42                 Debug.Print(e.Name);
43             }
44
45             // Modify document within a transaction
46             using (Transaction tx = new Transaction(doc))
47             {
48                 tx.Start("Transaction Name");
49                 tx.Commit();
50             }
51
52             return Result.Succeeded;
53         }
54     }
55 }
56
57
58

```

The most important part to us at this step is the Execute portion of this class (.cs). This is where we get to do our cool stuff.

For now, we will be **removing** lines 30-52 to make room for our “hello world” command.

HelloWorld.addin (Addin Manifest)

This is our file that tells Revit how to load the add-in. Without this, Revit does not know what to do.

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <RevitAddIns>
3  <AddIn Type="Command">
4      <Text>Command HelloWorld</Text>
5      <Description>Some description for HelloWorld</Description>
6      <Assembly>HelloWorld.dll</Assembly>
7      <FullClassName>HelloWorld.Command</FullClassName>
8      <ClientId>1910888d-5b39-4031-a4d8-f1c7751efdf0</ClientId>
9      <VendorId>com.typepad.thebuildingcoder</VendorId>
10     <VendorDescription>The Building Coder, http://thebuildingcoder.typepad.com</VendorDescription>
11 </AddIn>
12 <AddIn Type="Application">
13     <Name>Application HelloWorld</Name>
14     <Assembly>HelloWorld.dll</Assembly>
15     <FullClassName>HelloWorld.App</FullClassName>
16     <ClientId>27d21f47-4abf-429d-9ec5-8ffcd08d2d2b</ClientId>
17     <VendorId>com.typepad.thebuildingcoder</VendorId>
18     <VendorDescription>The Building Coder, http://thebuildingcoder.typepad.com</VendorDescription>
19 </AddIn>
20 </RevitAddIns>

```

This is an XML file that tells Revit where to look to load our stuff. Notice how it is using HelloWorld.dll and specifying both the app.cs and command.cs classes to load?

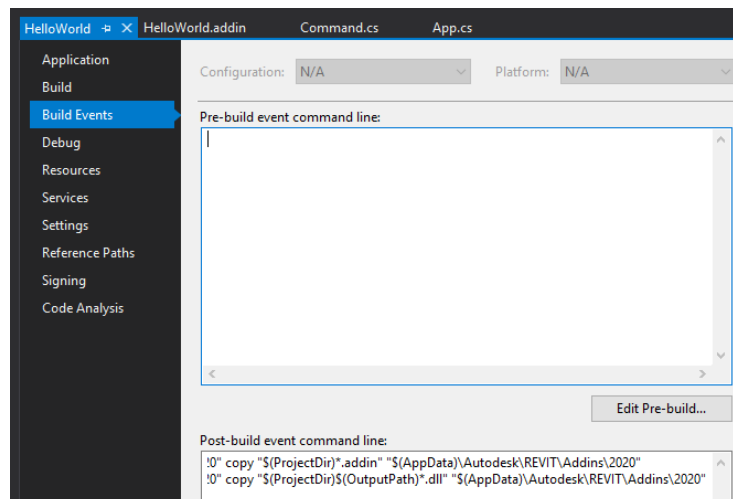
How to Load This?

With all the files available to us in Visual Studio, how exactly does Revit know what to do with it?

Once you compile your code, Revit will load add-ins from numerous locations. Probably the simplest one to use is Roaming Appdata.

C:\Users\USERNAME\AppData\Roaming\Autodesk\Revit\Addins\2020

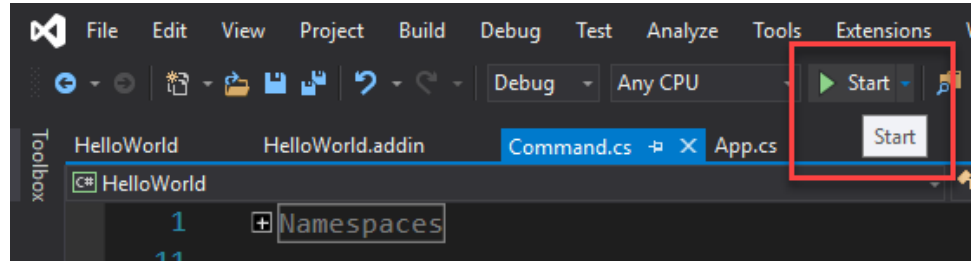
Lucky for us, the visual studio template does this copy for us upon build! (no need to memorize this or dissect as it is prebuilt into the template for us. Also, Jeremy's class really does a great job of explaining this).



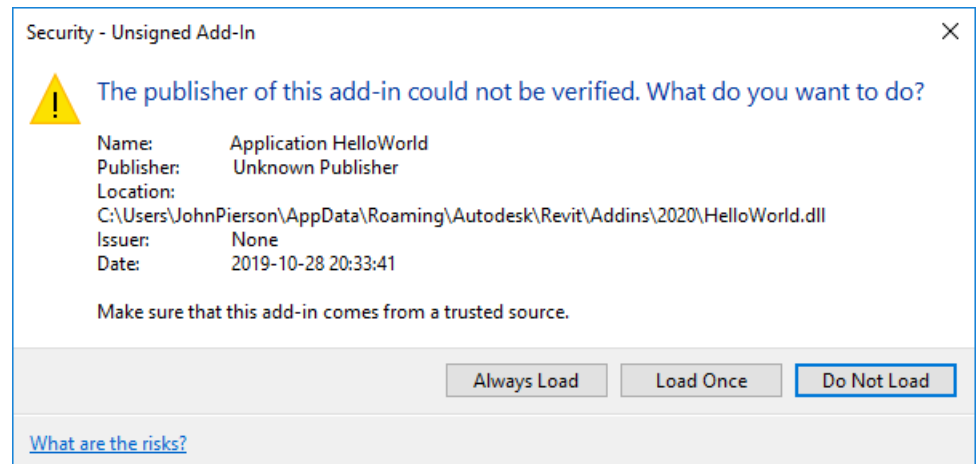
Launching Your Add-In

With that all mapped out, let's build our solution and see what happens!

By clicking start, we will build the solution and launch Revit in debug mode



Once Revit launches, we will be greeted with this lovely message. This is indicating that the code we are running is not yet signed with a Code Certificate. For now, we will click "always load". This will appear for both classes in our code, App.cs and Command.cs. (See the resources for more about code signing)

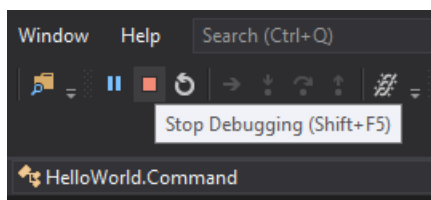
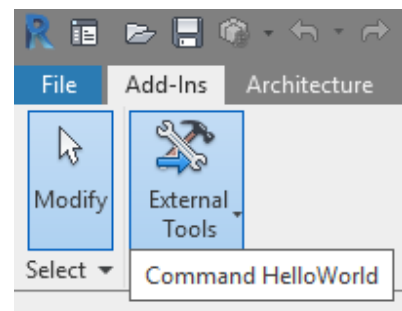


For testing purposes, we will start a new blank file.

Navigating to Add-Ins > External Tools, we will now see our new command!

However, clicking on it currently does nothing. 🤖

Let's go ahead and add some functionality to it.



In visual studio, press the stop button.

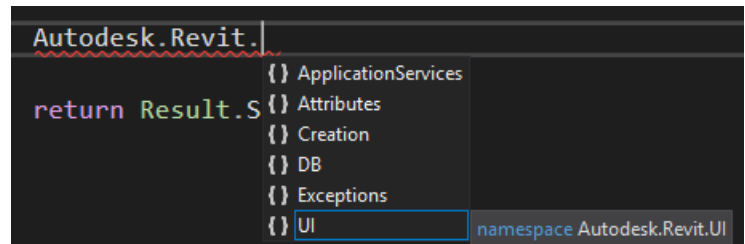
Adding Functionality

With a successful launch in our pocket, let's go ahead and add something fun to this tool. For this, we are going to show a pop-up dialog.

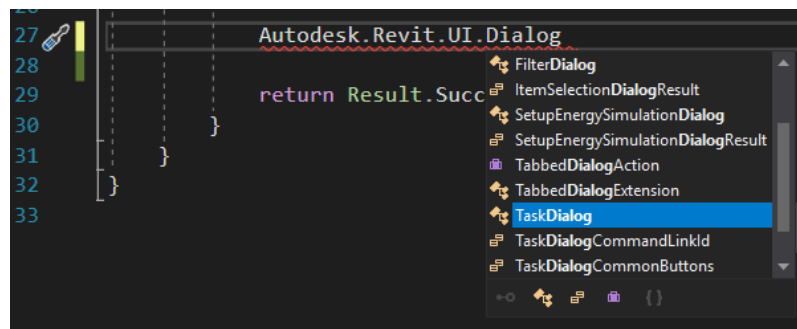
In visual studio, we can explore the Revit API in a very easy way using a feature called **Intellisense**.

Starting on line 31 of our Command.cs, let's step through with intellisense.

1. Type Autodesk and a dot ``
2. Namespaces will show up that you can "tab" to. In this case we want Revit, then UI.



3. Within UI, we have access to all the objects in that namespace. Let's select **TaskDialog**

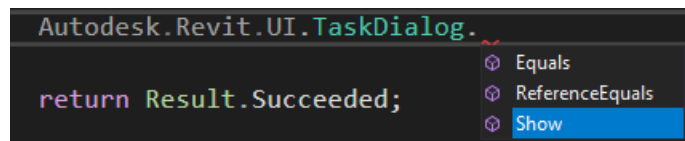


class Autodesk.Revit.UI.TaskDialog

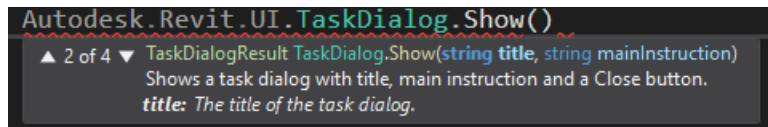
A task dialog is a dialog box that can be used to display information and receive simple input from the user. It has a common set of controls that are arranged in a standard order to assure consistent look and feel.

The description of TaskDialog from the Revit API.

4. Once we select TaskDialog, we can add another dot and see **methods** within that class. In this example, we want to **Show** a dialog.



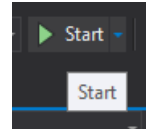
5. Once we type Show, we can add a pair of parentheses to fulfill parameters on this object's method. In this case, we have an option of the title and the main instruction to be shown. Let's go ahead and fill this out with something fun.



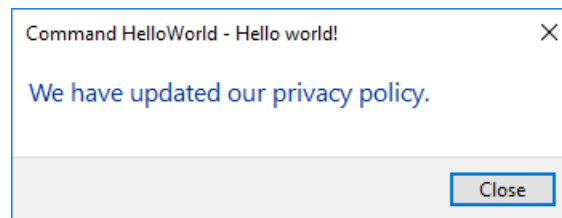
```
26
27 Autodesk.Revit.UI.TaskDialog.Show("Hello world!", "We have updated our privacy policy.");
28
```

Note: Since the data types expected are strings, we have to surround our parameters with “, to tell visual studio that we want to input text.

With this added, let’s go ahead and launch the debug mode again. *Press the green start button.*



And now, when you click on your command under external tools, you have something new and interesting.



Our first Dialog in Revit made with C#

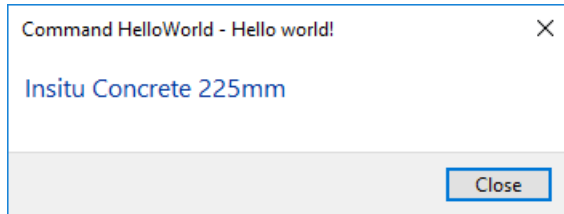
More Functionality

Now that we know how to launch a window, let’s make that window a bit more useful by prompting the user for a selection first.

On line 29, the template gave us access to the current selection methods available to us in Revit. Let’s use that to add some new functionality to our tool.

```
29 Selection sel = uidoc.Selection;
30
31 //using the selection method, ask user to pick one object and get it's element id
32 ElementId selectedElementId = sel.PickObject(ObjectType.Element).ElementId;
33
34 //get the element from the database given the ElementId
35 Element selectedElement = doc.GetElement(selectedElementId);
36
37 Autodesk.Revit.UI.TaskDialog.Show("Hello world!", selectedElement.Name);
38
39 return Result.Succeeded;
40 }
```

The above code results in this,



Next, we will obtain parameter values from that selected element. Let's plan for it being a wall for now.

```

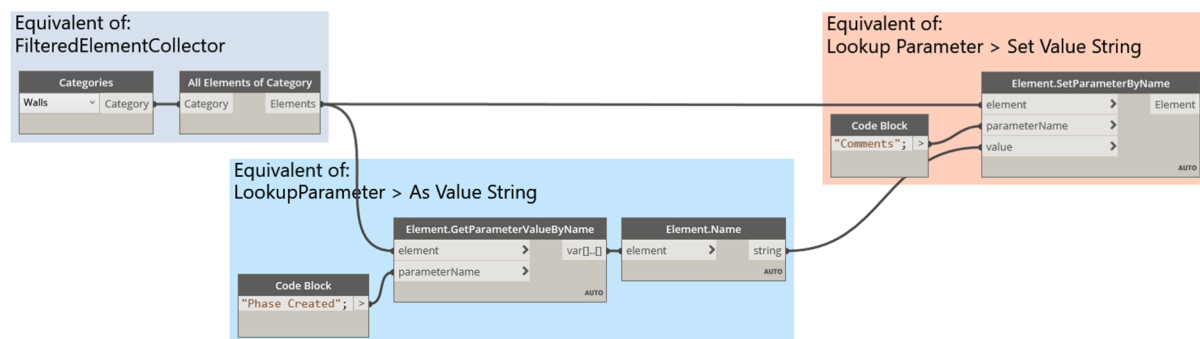
27 // Access current selection
28
29 Selection sel = uidoc.Selection;
30
31 //using the selection method, ask user to pick one object and get it's element id
32 ElementId selectedElementId = sel.PickObject(ObjectType.Element).ElementId;
33
34 //get the element from the database given the ElementId
35 Element selectedElement = doc.GetElement(selectedElementId);
36
37 //obtain a parameter's value given the name to display the value to the user
38 string unconnectedHeight = selectedElement.LookupParameter("Unconnected Height").AsString();
39
40 Autodesk.Revit.UI.TaskDialog.Show("Hello world!", unconnectedHeight);
41
42 return Result.Succeeded;
43

```

02_Phase to Comments

When people begin to be interested in the Revit API, their first step is to usually use Dynamo. Within this first step, users tend to use Dynamo to move parameters around.

For this example, we will be creating this Dynamo graph as an add-in. The graph below is getting all of our walls, obtaining the phase created name and setting it to the comments parameter. This is useful for additional filtering against. 😊



To get started we will start another visual studio project from our template.

Within this file we will have boiler-plate code. We need to go ahead and remove lines 46-52 as we will not be needing them.

As we can see we have some additional boiler plate for collecting things. (Specifically lines 33-37, which is in fact one whole statement separated by dots)

```

31 // Retrieve elements from database
32
33 FilteredElementCollector col
34 = new FilteredElementCollector(doc)
35   .WhereElementIsNotElementType()
36   .OfCategory(BuiltInCategory.INVALID)
37   .OfClass(typeof(Wall));
38
39 // Filtered element collector is iterable
40
41 foreach (Element e in col)
42 {
43     Debug.Print(e.Name);
44 }
45

```

In this case, the collector is equivalent to Dynamo's "All Elements of Type" node. We want to modify this to be of the wall category.

To do this, we will replace line 37 with ToElements().

```

33 FilteredElementCollector col
34 = new FilteredElementCollector(doc)
35   .WhereElementIsNotElementType()
36   .OfCategory(BuiltInCategory.INVALID)
37   .ToElements();
38

```

We will receive an error because we have a few other changes to make. The error we are receiving is due to our variable "col" being considered in the wrong way. We need to tell our code that this will hold a **List** of elements. In order to do this, we simply need to change line 33 to read as follows.

```

33 IList<Element> walls
34 = new FilteredElementCollector(doc)
35   .WhereElementIsNotElementType()
36   .OfCategory(BuiltInCategory.INVALID)
37   .ToElements();
38

```

At this point, we change our variable name to walls as well to be more specific and to alert users as to what we are wanting to collect.

The final step of collecting walls by category is to modify the category we are collecting. This appears on [Line 36](#) and is defaulted to INVALID. We need to change this to the category for walls, which is OST_Walls.

```

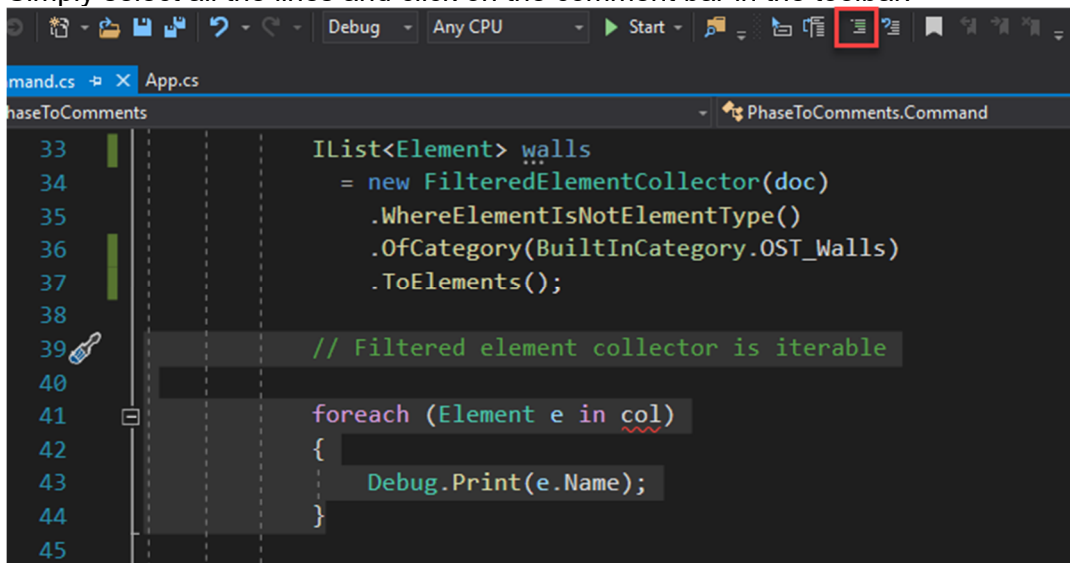
33      IList<Element> walls
34      = new FilteredElementCollector(doc)
35        .WhereElementIsNotElementType()
36        .OfCategory(BuiltInCategory.OST_Walls)
37        .ToElements();
38

```

At this point we are now collecting walls and getting a result to work with.

In order to test our code, let's go ahead and comment out [lines 41-44](#) as we will modify this later.

Simply select all the lines and click on the comment bar in the toolbar.



```

33      IList<Element> walls
34      = new FilteredElementCollector(doc)
35        .WhereElementIsNotElementType()
36        .OfCategory(BuiltInCategory.OST_Walls)
37        .ToElements();
38
39      // Filtered element collector is iterable
40
41      foreach (Element e in col)
42      {
43          Debug.Print(e.Name);
44      }
45

```

We will also add a debug point to our code for it to pause at so we can view the results. This is achieved by clicking in the gray sidebar in visual studio on [line 48](#).


```

33  IList<Element> walls
34      = new FilteredElementCollector(doc)
35          .WhereElementIsNotElementType()
36          .OfCategory(BuiltInCategory.OST_Walls)
37          .ToElements();
38
39      // Filtered element collector is iterable
40
41      //foreach (Element e in col)
42      //{
43          //    Debug.Print(e.Name);
44      //}
45
46
47
48      return Result.Succeeded;
49  }

```

Now we can run our code.

One we click on the command in Revit we will be thrown to Visual Studio and we can see what our variable called **walls** is achieving.

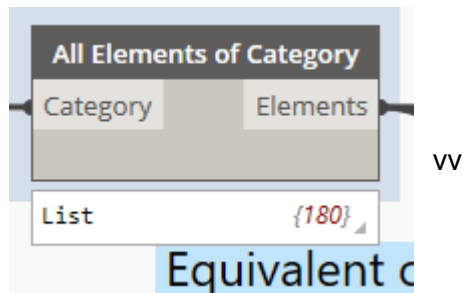
```

32
33  ▶ | IList<Element> walls
34      = new FilteredEle
35          .WhereElementIsNotElementType()
36          .OfCategory(BuiltInCategory.OST_Wall
37          .ToElements();
38
39      // Filtered element collector is iterable
40
41      //foreach (Element e in col)
42      //{
43          //    Debug.Print(e.Name);
44      //}
45
46
47
48      return Result.Succeeded;
49

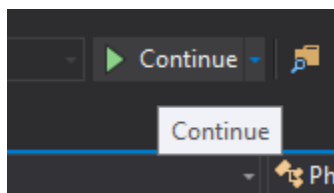
```

walls Count = 180

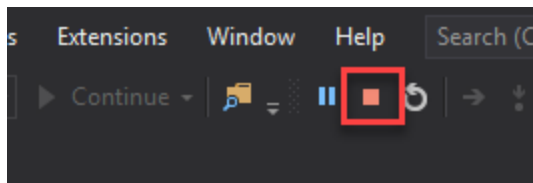
As you can see, we are actively collecting 180 walls! Thankfully, this is the same amount Dynamo would return as well. So we did something correct!



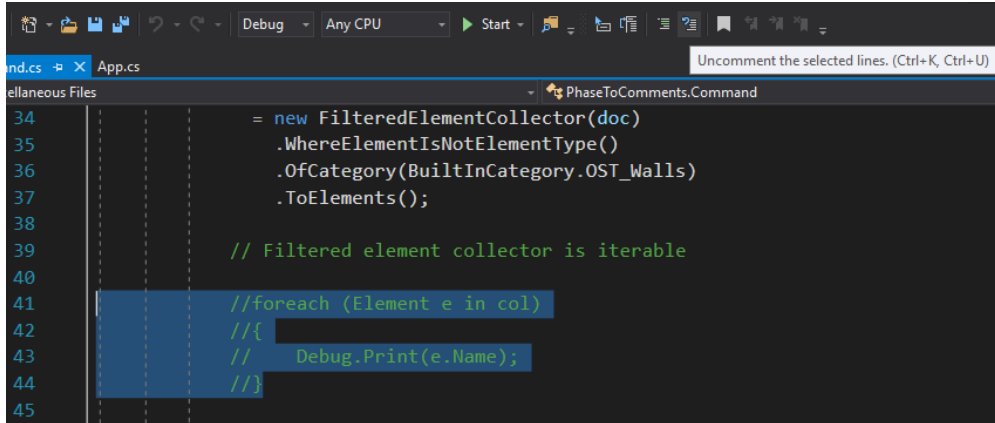
In order to continue, we will press the play button again.



Once we continued the execution of our code we can now stop the execution and do something with the collected walls.



In order to modify our walls, we need to uncomment [lines 42-44](#) and modify them within the foreach loop. (We will discuss a foreach loop next)

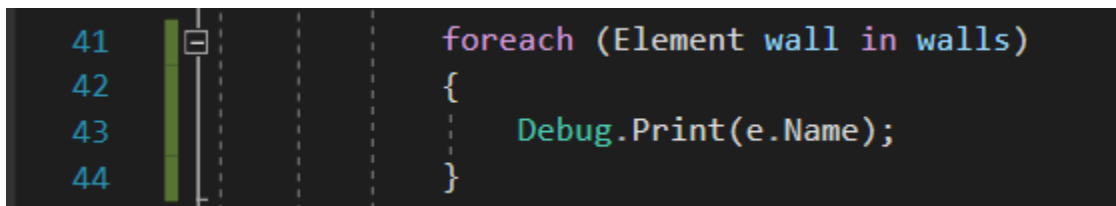


```

34     = new FilteredElementCollector(doc)
35       .WhereElementIsNotElementType()
36       .OfCategory(BuiltInCategory.OST_Walls)
37       .ToElements();
38
39     // Filtered element collector is iterable
40
41     //foreach (Element e in col)
42     //{
43     //    Debug.Print(e.Name);
44     //}
45
  
```

In order to properly go through each wall in a list, we need to iterate through each individual one. Think about anytime you have asked a group of people a question and they all answered at once. Did you understand everyone's answer? Probably not. But if you walk up to each person and ask them individually, you get an answer that makes sense.

Now, within the foreach loop we can modify each wall on its own. In order to do this, we need to tell the foreach loop to iterate through our walls. We can do this by changing [line 41](#) to read as follows.



```

41     foreach (Element wall in walls)
42     {
43         Debug.Print(e.Name);
44     }
  
```

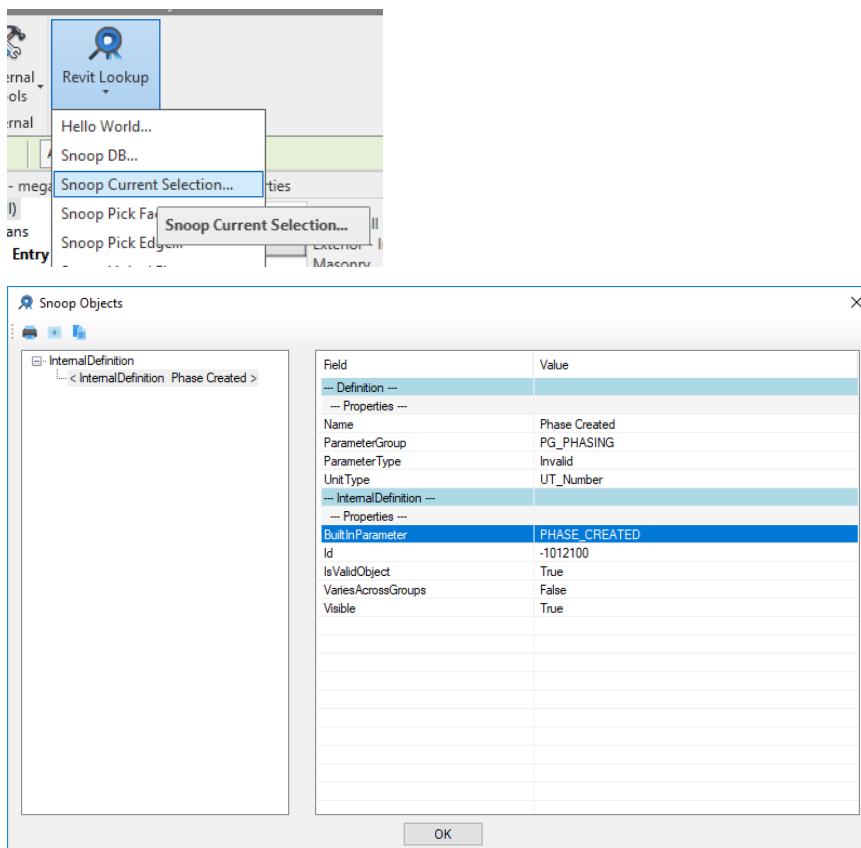
Now we are telling Visual studio to touch each wall and store it as a temporary variable of wall for us to use.

Next, we can remove [line 43](#) and replace with our own logic.

To get started, we need to obtain the Phase Created parameter as a text value (string). This can be achieved using the `get_parameter` method.

```
41 foreach (Element wall in walls)
42 {
43     string phaseCreated = wall.get_parameter(BuiltInParameter.PHASE_CREATED).AsString();
44 }
```

But how do we know what name to lookup? We use the Revit Lookup tool! Within the tool we can “snoop” a current selection of wall and see the parameter name we need to use.



With the parameter name for Phase Created and Comments now known to us, we can write the rest of our logic in Visual Studio.

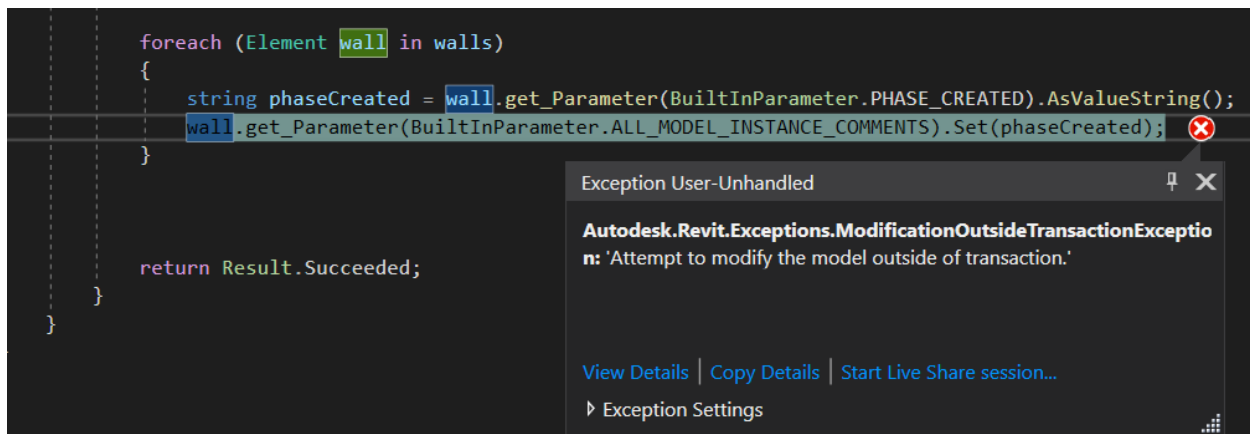
```

41     foreach (Element wall in walls)
42     {
43         string phaseCreated = wall.get_Parameter(BuiltInParameter.PHASE_CREATED).AsString();
44         wall.get_Parameter(BuiltInParameter.ALL_MODEL_INSTANCE_COMMENTS).SetStringValue(phaseCreated);
45     }
46

```

On line 44 we are now telling the wall to change the comments to the phase created parameter we previously obtained.

Let's go ahead and run the tool on our sample file to see what it does.



This error means that we are trying to modify Revit in a weird way. We need to use something called a transaction to update the Revit database.

Transactions?

Transactions can be conceptualized in the same manner that they obtain their namesake from. Think about buying a candy bar at a candy store. For you to obtain the candy bar, you need to start a transaction with the cashier. Within that transaction certain things take place. This includes, finding out the total cost, handing over the money, ringing up the item and closing out the sale. The same thing goes for Revit. We start the transaction, do some things and apply the changes to the Revit model. Note: A transaction in an add-in will almost always result in having an undo available in Revit.



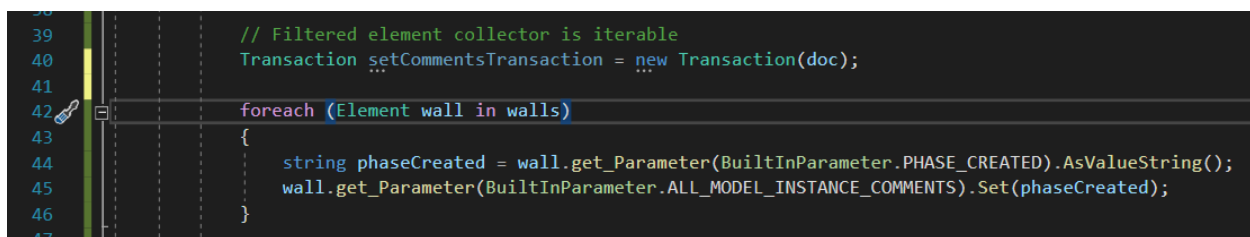
Charlie Knows About Transactions!

With this in mind, we can go ahead and stop executing our code and add in the Transaction logic.

Back in Visual Studio we can add the Transaction logic on [Lines 40, 41 and 46](#). This way one transaction will work for all 108 elements.

In the Revit API a transaction is an object that you can create. In order to do this, we can declare a transaction object as a variable as follows.

```
| Transaction setCommentsTransaction = new Transaction(doc);
```



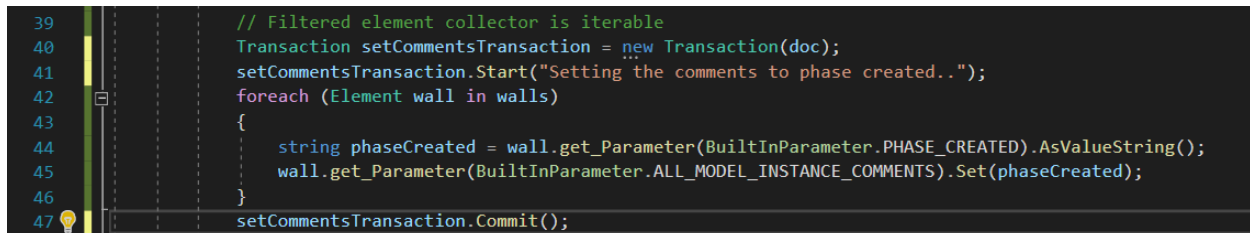
```

39 // Filtered element collector is iterable
40 Transaction setCommentsTransaction = new Transaction(doc);
41
42 foreach (Element wall in walls)
43 {
44     string phaseCreated = wall.get_Parameter(BuiltInParameter.PHASE_CREATED).AsValueString();
45     wall.get_Parameter(BuiltInParameter.ALL_MODEL_INSTANCE_COMMENTS).Set(phaseCreated);
46 }
47

```

On [line 40](#), we have now declared a transaction object and named a variable for it. In addition to this, we have assigned it to our current Revit document (doc) which is part of our boilerplate code.

Now we can use the transaction.



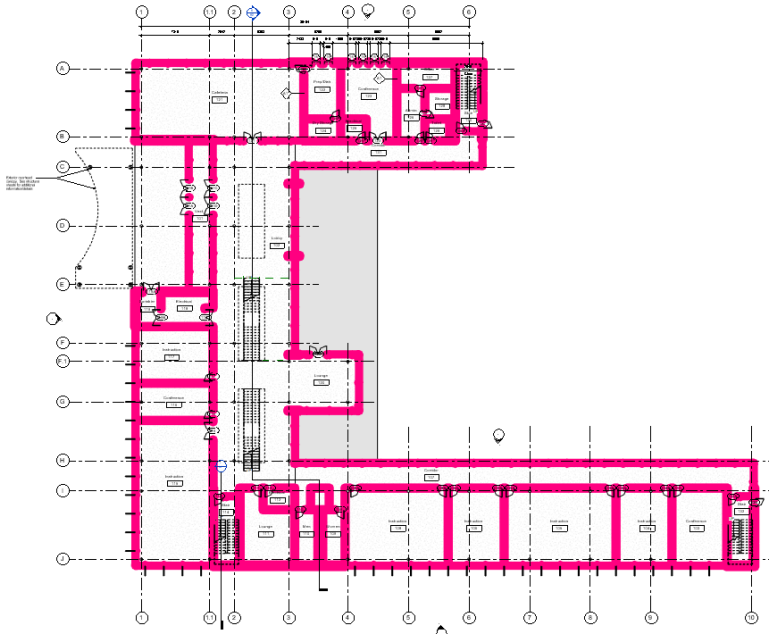
```

39 // Filtered element collector is iterable
40 Transaction setCommentsTransaction = new Transaction(doc);
41 setCommentsTransaction.Start("Setting the comments to phase created..");
42 foreach (Element wall in walls)
43 {
44     string phaseCreated = wall.get_Parameter(BuiltInParameter.PHASE_CREATED).AsValueString();
45     wall.get_Parameter(BuiltInParameter.ALL_MODEL_INSTANCE_COMMENTS).Set(phaseCreated);
46 }
47 setCommentsTransaction.Commit();

```

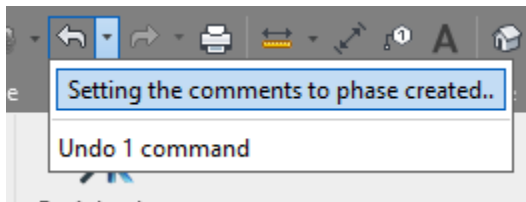
In order to use our transaction, we need to tell it to start and to finish. During the start process we can also name it as well, resulting in the name of the command in the undo panel.

With [lines 41](#) and [47](#) fulfilled, let's go ahead and run it again. This time it will take a few moments longer to run, but once it finishes, we should see something funky going on!

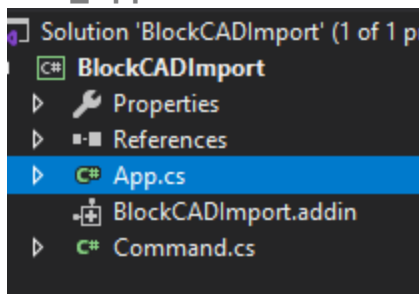


Once it is complete, your building should have magenta walls all over the place! This is due to a custom view filter in the view, **01 – Entry Level**, that is working based on the wall's comments.

Additionally, we now have an undo available to us.



03_Application-Level Tools



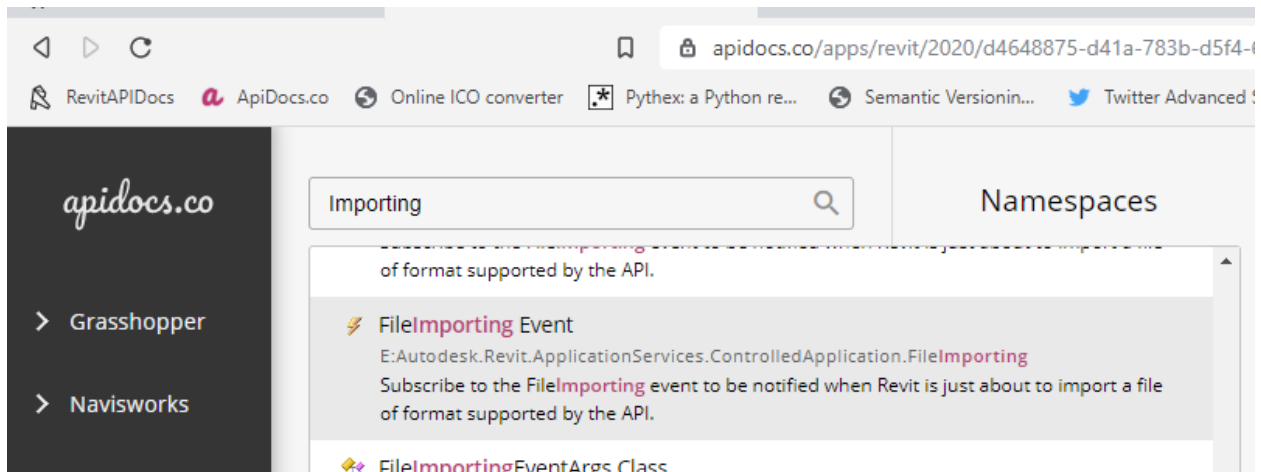
One of the biggest questions I receive is, "When should I use an add-in versus when should I use Dynamo?"

The typical answer is, it depends. But, one huge thing that add-ins can do that Dynamo can't do (as easily) is trigger on Revit launching.

Remember that other file named App.cs from before? This allows us to do things on a Revit application wide level.

Within App.cs we can set up logic to interact with Revit when users do things. One that recently came up was when the Import CAD command was fired off. Luckily this is doable with another great tool called events.

Events in Revit are awesome little functions that happen when a user does something. Like saving, printing, opening, etc. Within these possibilities is the ability to trigger when the CAD Importing operation has started.



Searching on APIDocs.co we can see that this event can be assigned to the ControlledApplication object in the Revit API. Let's look at how to implement that now.

Within App.cs we have a section called OnStartup. This allows us to assign this event to Revit starting. It also gives us access to a variable called, a for UIControlledApplication. (This is Revit API speak for Revit's UI Interactions.)

```

10 namespace BlockCADImport
11 {
12     0 references
13     class App : IExternalApplication
14     {
15         0 references
16         public Result OnStartup(UIControlledApplication a)
17         {
18             return Result.Succeeded;
19         }
20
21         0 references
22         public Result OnShutdown(UIControlledApplication a)
23         {
24             return Result.Succeeded;
25         }
26     }
27 }

```


Within the OnStartup method we can start to define this logic. Using the information obtained from apidocs.co, we know we need to find the ControlledApplication object.

Let's find this using Intellisense,

```

14 public Result OnStartup(UIControlledApplication a)
15 {
16     a.Control
17     re Control
18     }
19     ControlledApplication
    
```

ControlledApplication UIControlledApplication.ControlledApplication { get; }
Returns the database level ControlledApplication represented by this UI-level ControlledApplication.

With that found, we can now navigate the methods available to us with the arrow keys,

Eventually, we stumble upon the FileImporting event. *(Note: We could have searched for this as well)*

```

14 public Result OnStartup(UIControlledApplication a)
15 {
16     a.ControlledApplication.
17     return Result.Succeeded;
18 }
19
20 public Result OnShutdown(UIControlledApplication a)
21 {
    
```

FileExporting
FileImported
FileImporting
GetHashCode
GetLibraryPaths

EventHandler<Autodesk.Revit.DB.Events.FileImportingEventArgs> ControlledApplication.FileImporting
Subscribe to the FileImporting event to be notified when Revit is just about to import a file of format supported by the API.

In order to use this event, we need to **subscribe** it to a **handler** method. This is like something like IFTTT. Where we can say "If this happens, then do this".

The syntax for subscribing isn't immediately apparent, but once you know, you know. In order to subscribe we must use **+=** along with the handler method name. Luckily, Visual Studio can create this handler for us. (by pressing tab)

```

14 public Result OnStartup(UIControlledApplication a)
15 {
16     a.ControlledApplication.FileImporting +=
17     return Result.Succeeded;
18 }
    
```

ControlledApplication_FileImporting; (Press TAB to insert)

After we hit tab, we should have the following in our App.cs file.

```

10 namespace BlockCADImport
11 {
12     0 references
13     class App : IExternalApplication
14     {
15         0 references
16         public Result OnStartup(UIControlledApplication a)
17         {
18             a.ControlledApplication.FileImporting += ControlledApplication_FileImporting;
19             return Result.Succeeded;
20         }
21         1 reference
22         private void ControlledApplication_FileImporting(object sender, Autodesk.Revit.DB.Events.FileImportingEventArgs e)
23         {
24             throw new NotImplementedException();
25         }
26         0 references
27         public Result OnShutdown(UIControlledApplication a)
28         {
29             return Result.Succeeded;
30         }
31     }

```

Now that we have that code initialized, we need to make it do something.

As you can see we received a variable named “e” that handles the object of Autodesk.Revit.DB.Events.FileImportingEventArgs. This is great, because now we can use intellisense to see what all we can do with the importing action.

Adding a new line on [line 22](#), we can now type `e.` to find what is available to us.

```

20 private void ControlledApplication_FileImporting(object sender, Autodesk.Revit.DB.Events.FileImportingEventArgs e)
21 {
22     e.
23     th
24     }
25

```

Intellisense suggestions for `e.`:

- `Cancel` (selected)
- `Cancelable`
- `Dispose`
- `ImplementedException()`
- `void Autodesk.Revit.DB.Events.RevitAPIPreEventArgs.Cancel()`
- `When the event is cancellable, may call the Cancel() method to cancel it.`

As you can see, we have the option to flat-out cancel the operation. Let’s select that and hit enter.

Once we have the cancel method typed in, we need to remove [line 23](#).

Our code should now look like this,

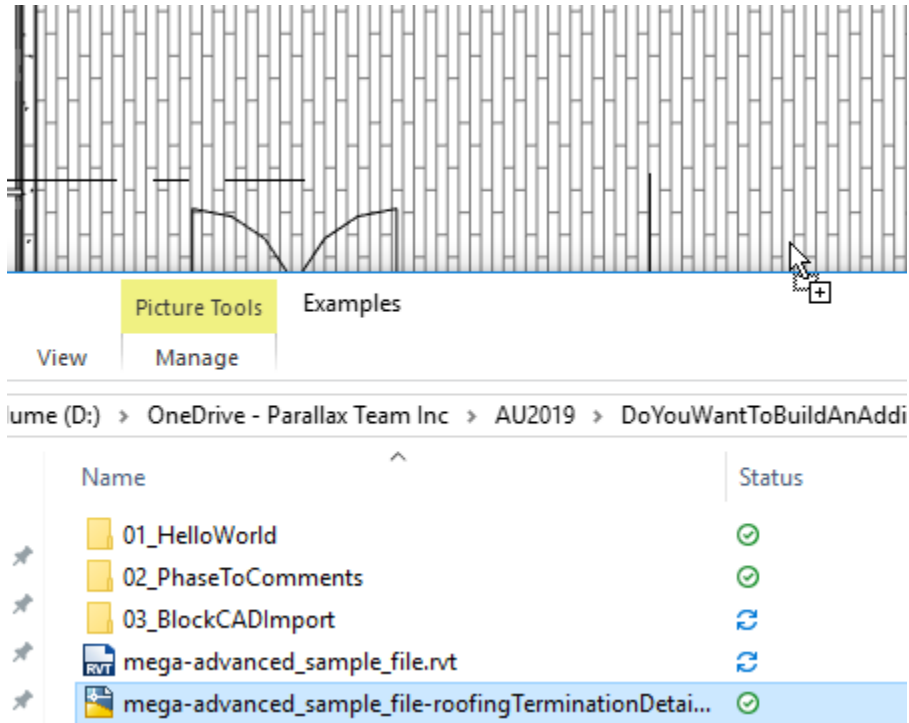
```

20 private void ControlledApplication_FileImporting(object sender, Autodesk.Revit.DB.Events.FileImportingEventArgs e)
21 {
22     e.Cancel();
23 }
24
25

```

Let’s go ahead and run it and do some testing.

With Revit open, try to drag the CAD file in the dataset into your current Revit model.



As you can see this command is completely blocked. We have no CAD import now and it just didn't happen. The downside is, we are not telling the user what happened and that can lead to confusion.

Let's go ahead and stop running our code and add some additional logic to this.

In our first example, we used something called a TaskDialog to display information to our end user. Using that same idea, we can also alert users to something that is happening in our add-in.

On [line 23](#) let's add that logic now.

```

20 1 reference
21 private void ControlledApplication_FileImporting(object sender, Autodesk.Revit.DB.Events.FileImportingEventArgs e)
22 {
23     e.Cancel();
24     TaskDialog.Show("CAD Import Failed", "Yeah, sorry no cad imports for you");
25 }

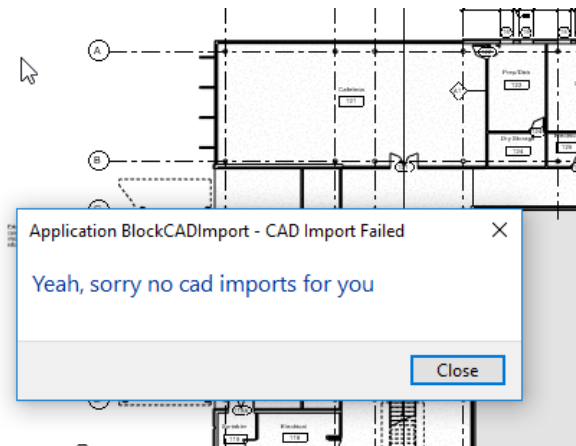
```

With this logic embedded we will now let a user know what is happening after we cancel that import.

We can test this for ourselves by clicking the start button again.

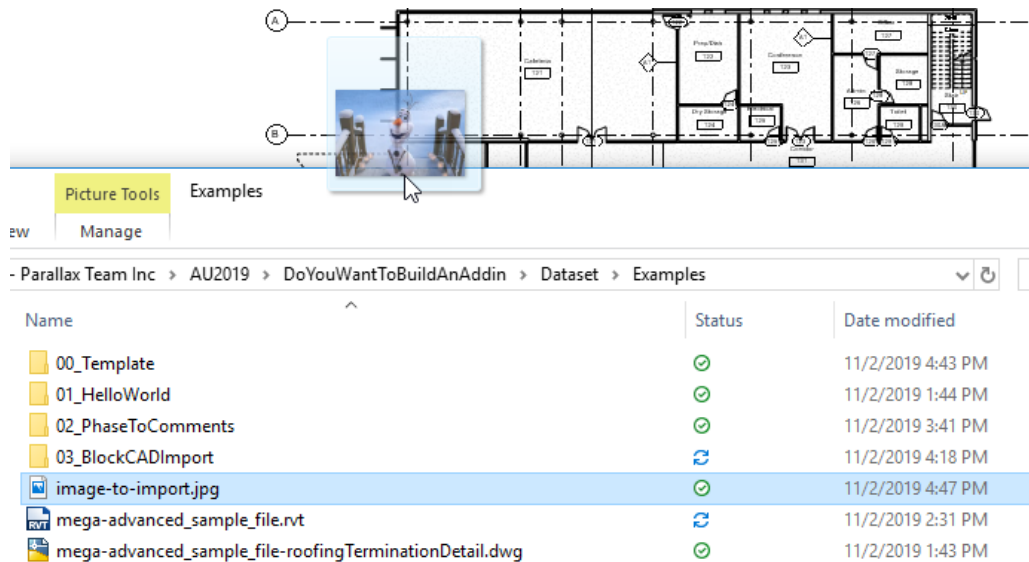
Now try dragging that same CAD file into your Revit model.

You will now notice the import was cancelled and your message is displayed.



This is nice, because you can now provide more info to the end user and go from there.

Next, let's try to import an image. *(There is also a sample jpg in the data set for this.)*



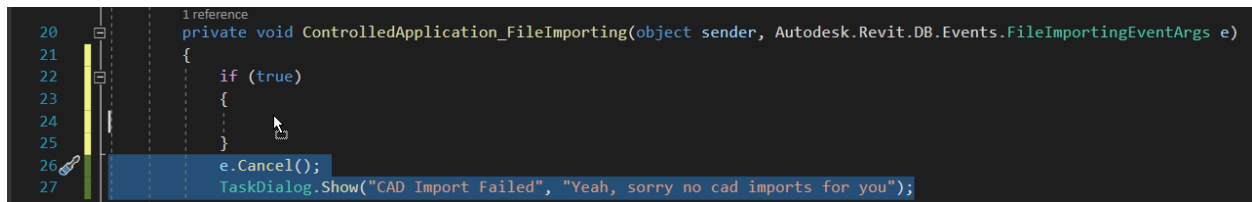
You will notice that your new logic for blocking imports encompasses image imports as well! Doh!

Luckily, we can fix this to allow for images to be imported, while blocking CAD.

Back in Visual Studio, we will add another line to our method. This will be [line 22](#) now. Typing if and pressing tab twice will give us an if statement to use.

An if statement is known as a conditional and allows us to test certain conditions before moving on.

With this code generated, we can now move our logic on [lines 26-27](#) within the if statement. (Select the lines and then drag them over.

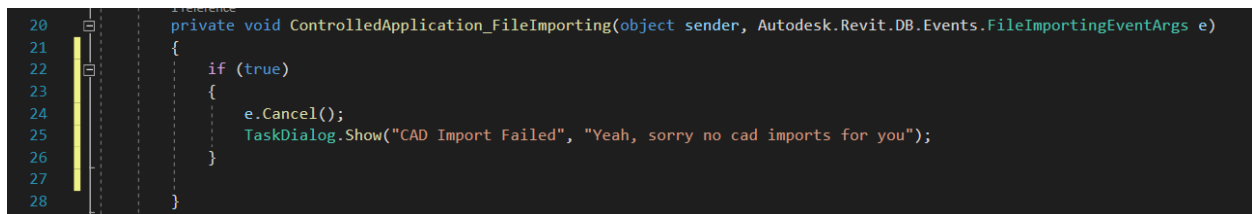


```

20 private void ControlledApplication_FileImporting(object sender, Autodesk.Revit.DB.Events.FileImportingEventArgs e)
21 {
22     if (true)
23     {
24
25     }
26     e.Cancel();
27     TaskDialog.Show("CAD Import Failed", "Yeah, sorry no cad imports for you");

```

You may notice your indenting getting messed up here, a quick way to fix this is to select lines 22-26 and press CTRL+K,D on your keyboard.



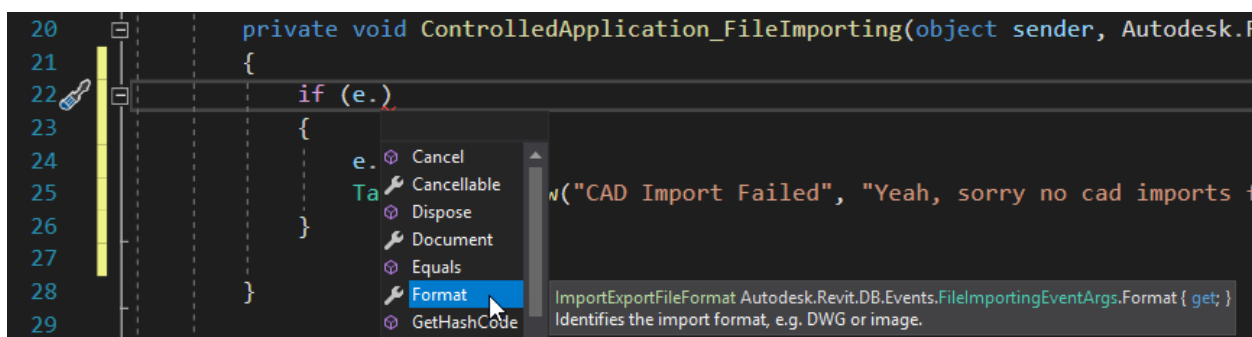
```

20 private void ControlledApplication_FileImporting(object sender, Autodesk.Revit.DB.Events.FileImportingEventArgs e)
21 {
22     if (true)
23     {
24         e.Cancel();
25         TaskDialog.Show("CAD Import Failed", "Yeah, sorry no cad imports for you");
26     }
27 }
28

```

At this point, our If statement is always true. We need to replace this with our own logic that we are testing. In this case, we are going to use the variable of `e` again to see what is available to us.

Using Intellisense, we notice that we have an option for seeing the format of the import. This is exactly what we need to continue. Select this option.



```

20 private void ControlledApplication_FileImporting(object sender, Autodesk.Revit.DB.Events.FileImportingEventArgs e)
21 {
22     if (e.)
23     {
24         e.
25     }
26 }
27
28
29

```

- Cancel
- Cancellable
- Dispose
- Document
- Equals
- Format**
- GetHashCode

ImportExportFileFormat Autodesk.Revit.DB.Events.FileImportingEventArgs.Format { get; }
Identifies the import format, e.g. DWG or image.

In order to compare that format to what we want, we can use the word **is**. What is will let us do is determine if the import object **is** of the type DWG.

```
22  if (e.Format is)  
23  {  
24      e.Cancel();  
25      TaskDialog.Show("CAD Import Failed", "Yeah, sorry no cad imports for you");  
26  }  
27  
28  }
```

ImportExportFileFormat Enumeration

Revit 2020 API

See Also

Describes formats supported by import / export.

Namespace: **Autodesk.Revit.DB**

Assembly: RevitAPI (in RevitAPI.dll) Version: 20.0.0.0 (20.0.0.377)

Since: 2010

Syntax

C#

```
public enum ImportExportFileFormat
```

Members

Member name	Description
DWG	DWG format

In order to specify DWG, we can look at [ApiDocs.co](https://api.docs.co) for what options are available to us for Format.

With that in mind, we can check to see if the import **is** of the type `ImportExportFileFormat.DWG`.

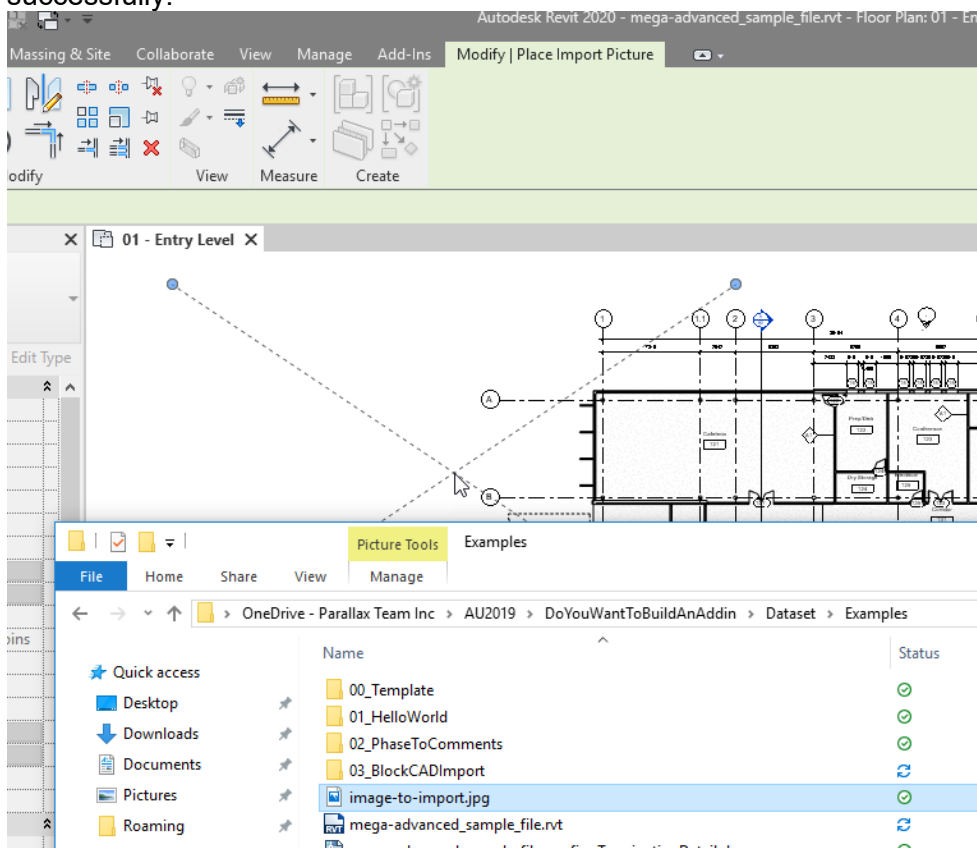
```

20 private void ControlledApplication_FileImporting(object sender, Autodesk.Revit.DB.Events.FileImportingEventArgs e)
21 {
22     if (e.Format is ImportExportFileFormat.DWG)
23     {
24         e.Cancel();
25         TaskDialog.Show("CAD Import Failed", "Yeah, sorry no cad imports for you");
26     }
27 }
28

```

Now, run the code and see what happens in both scenarios.

You should see the DWG being blocked with our nifty dialog box, and the JPG being imported successfully!



Now you know about one of Revit add-in's best features, events! You also know about if statements and comparison of object types now because of this example. 😊

Conclusion

Well, there we have it. All the info that I could cram in regarding getting started with Revit Add-Ins based on my experience as a Revit and Dynamo user. I hope that this material helps you in your Revit API journey and gave you at least a few new ideas/tools to work with. I hope that you go off and do awesome things in Revit now and share them with your firms and the AEC community. I also wanted to thank you for considering my class and for making all this worth it. Go do awesome things!



John Pierson
Design Technology Specialist
PARALLAX TEAM
johnPierson@parallaxTeam.com
sixtysecondrevit@gmail.com
@60secondrevit