SD224848-L

Pushing Revit to the Next Level: An Intro to Revit Plugins with C#

Jeremy Graham
HDR

Learning Objectives

- Understand how to build a Revit plug-in using Visual Studio
- Understand how to access the Revit API with C# by learning the basics of the programming language.
- Find different areas of the Revit API to use with C# to learn how to work with Revit Elements.
- Learn how to create and use Revit API objects when creating Revit plug-ins such as Revit Parameters and FilteredElelementCollectors.

Description

Revit provides an amazing set of tools to develop and document sophisticated building models. These tools can be extended even further by tapping directly into the Revit Application Programming Interface and creating commands by building Plugins with C#.

In this lab, we will learn how to build our own plugins from scratch to extend the functionality of Revit. We will start by learning about some of the basics of the C# programming language before booting up Microsoft Visual Studio and set up our first command. Once we understand how to build and use Revit plug-ins, we will delve deeper into the Revit API to learn about how to work with FilteredElementCollectors, Parameters and Transactions.

Speaker(s)

Jeremy Graham is the Computational Design Leader at HDR, driving Data-driven Design (D3) within the Australian market. Jeremy develops computational models and tools that assist designers and clients in utilising big data for real-time space planning and decision making. Jeremy has a strong passion for creating, whether it’s designing a building or a new tool from start to end with emerging technologies.

Jeremy has made a significant contribution to the computational design and BIM community in the Asia-Pacific and US, with speaking engagements at BILT Asia, BILT ANZ and produced online courses with LinkedIn that teach programming to architects and engineers utilising Python and C#.

Building Plugins

Before we get stuck in to learning about C# and the Revit API, it's important to understand what we will be learning and the advantages of doing so. Let's have a look at each of the main components that we will be learning about in this class

C#

C# is a general purpose, high-level, object-oriented programming language developed by Microsoft as part of the .Net Initiative. It is relatively easy to learn with huge amounts of resources and support available on the web. As part of the .Net Initiative, C# written applications run on the .Net Framework.

.Net Framework

The .Net framework is a platform which provides libraries required to develop windows and web applications along with the Common Language Runtime which essentially manages the execution of .Net applications. When building C# applications, a version of the .Net framework is targeted allowing other windows application to utilize the resulting compiled files that target the same, or earlier, version of the .Net framework. When we create our first plugin, we will target the .Net framework that is suitable to run with Revit.

Revit API and Plugins

The Revit API or Application Programming Interface is a library that allows developers to integrate applications in to Revit. It provides the code base required to interact with Revit by making calls to Revit through methods provided by the Revit API.

The Revit API supports languages compatible with the .Net framework such as C#. By accessing the Revit API, developers have the ability to integrate new applications with Revit and extend the capabilities of the software, beyond the out-of-the-box functions.

C# Key Concepts

In order to understand the plugins, we are building, we need to know the language we will be writing it with, in this case C#.

Object-Oriented Programming

C# is an object-oriented programming language. This means everything in C# is an object. An object can contain properties, which describe the object, and methods which enable the object to perform a function. This wraps the programming logic up into objects that can interact with one another depending on their function.

Objects are often designed to describe the role that the object performs. For example, say we had an object that represented a cat. The cat object would have methods that allow the object to do something, such as meow, and properties that describe the cat, such as it's color, grey for example. In the Revit API, Revit elements are created as objects, for example a view can be created as an object which will contain methods such as adding filters, and properties related to that view such as its name.

Classes

To create objects, a class needs to be defined which acts as the blueprint for an object. When a class is created, it will define all the properties and methods that an object will contain once the object is created or, instantiated. These different components that make up a class are known as members. The image below shows an example class structure for a Cat class and some of the members it may contain.

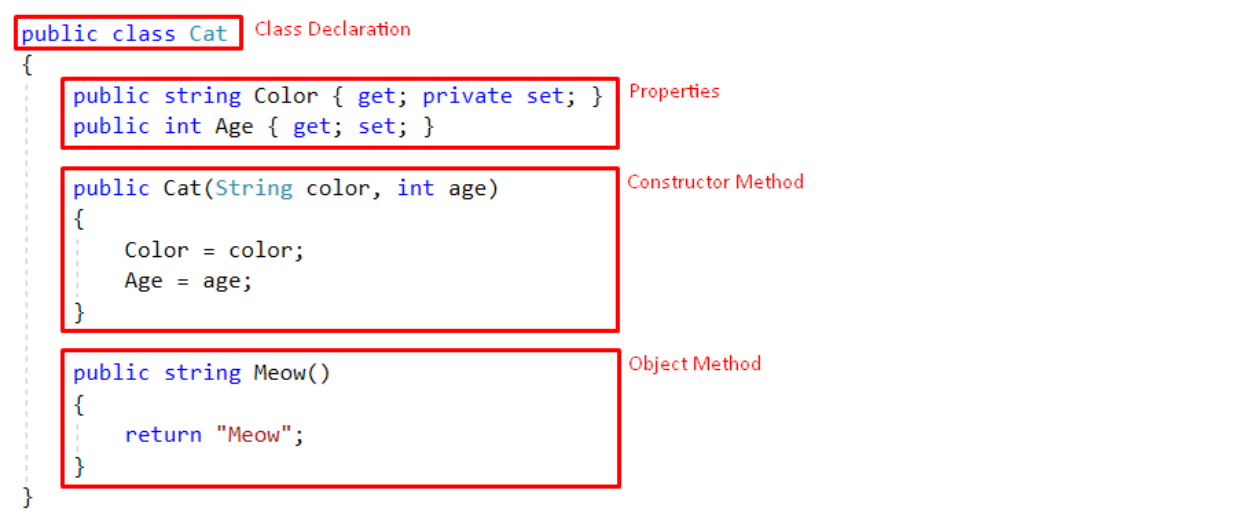


Figure 1 Cat Class

While we won't be learning how to create classes in this lab, it is useful to understand the anatomy of a class and the members they contain.

Modifier

When declaring classes or properties, modifiers are assigned to indicate the accessibility of the class or member. This can be public, private or internal for example. In the Cat class example, the class is Public, meaning it can be accessed from any other code. The Cat class property Color set accessor however is set to private which means no code outside of the class can set the property.

```
public class Cat
{
    public string Color { get; private set; }
    public int Age { get; set; }
```

Figure 2 Modifiers

Statements

C# code is littered with semicolons (;) and curly brackets ({}), and this is because they define statements and their blocks. Each line in the Cat class example that performs an action is a statement in C#, for example declaring a class or getting and setting a property. A statement can be a single line of code that ends in a semicolon (;), or multiple lines of statements that are enclosed in a statement block. Any statements that require an associated block of code, such as the class declaration or constructor method, require curly brackets after the declaration, known as a statement block. Any code added inside of a statement block is actioned as part of the statement.

```
public class Cat                    Declaration Statement
{
    public string Color { get; private set; }    Statement Block
    public int Age { get; set; }

    public Cat(String color, int age)
    {
        Color = color;              Assignment Statement
        Age = age;
    }
```

Figure 3 Statements

Dot Notation

Dot Notation is used to access the methods and attributes of an object. This is done by using a dot (.) after an object, followed by the method or property name. When accessing a property of an object, the name of the property is used. When accessing the method of an object, the method name is followed by a pair of parentheses. If a method requires a parameter, such as an item added to a list, this is included within the parenthesis.

Different Method Types

A class can contain a few different methods which can be used in different ways, these are constructor methods, object methods and static methods.

Constructor methods are called directly from classes and are used to create an instance of the class to create an object. There can be several different constructor methods defined in a class which take different parameters, these are known as overloaded methods. When creating an object using a constructor, the new keyword needs to be used. For example, the cat class may have two different constructor methods, depending on which is used to construct a cat object changes the object's initial properties.

```
Cat cat = new Cat();          Instantiate Cat Object with no Parameters
Cat cat2 = new Cat("Grey", 2); Instantiate Cat Object with Parameters
```

Figure 4 Constructor Methods

Static methods are those that are defined within a class but do not require an instantiated object to be used so they can be called without an object. For example, a cat class may define a static method that counts the number of cats which can be called by simply accessing the cat class.

```
Cat.CountCats();
```

Figure 5 Static Methods

Object methods are those that have already been described, that is they can be accessed from an object and allow the object to perform some sort of action.

```
Cat cat = new Cat();

Cat cat2 = new Cat("Grey", 2);

cat2.Meow();   Object Method
```

Figure 6 Object Methods

Variables

Variables are simply names that point to a location in memory which may store a value or reference to an object. These variables can then be used in different operations.

When constructing a variable, that data type that it will be storing needs to be defined before the name of the variable. This is because C# is strongly typed programming language, which means the variable type needs to be declared when creating the variable. For example, the variable below is named Molly and is referencing a type of cat which is being constructed using the constructor method. Once the cat is constructed, the Molly variable can be used to access it's methods and properties as shown.

```
Cat Molly = new Cat();
```

Cat Variable

Figure 7 Variables

One thing to keep in mind that we can't have more than one variable with the same name.

C# Basics

Now that we understand the anatomy of classes and objects, let's have a look at some of the basic types in C# that are used quite often. Some of these are value types in that they don't need to be created using constructors, they can simply be assigned to a variable of that type.

Comments

Comments can be added to C# scripts by using either double forward slash (//) for single line comments or forward slash star combinations for multiline comments (/* */).

```
//This is a comment

/* This is
 * a multiline
 * comment */
```

Figure 8 Comments

Strings

Strings can be thought of as a piece of text made up of characters. Strings are created by inserting characters between single or double quotation marks. Below is an example of creating the string “Hello”.

```
string name = "Molly";
```

Figure 9 Strings

Numbers

There are quite a few different number types that can be used with C#. The two most common types are integers and doubles. Integers can be thought of a whole number such as 1, 2, or -1. Doubles can be thought of as decimal point numbers such as 1.2, 4.5 or -1.4. Examples of these are shown below.

```
int integer = 1;
double dub = 1.0;
```

Figure 10 Numbers

Lists

Lists are basically used as a container of objects. When constructing a list, the type of object that the list is holding needs to be specified. For example, the image below demonstrates instantiating a list object named Cats which contains a list of Cat objects.

```
List<Cat> Cats = new List<Cat>();
```

Figure 11 Lists

Objects can then be added to a list by accessing the add method from the list object. The example below demonstrates adding a cat object to the list.

```
Cat Molly = new Cat();
List<Cat> Cats = new List<Cat>();
Cats.Add(Molly);
```

Molly Added to Cats List

Figure 12 List Add Method

Booleans

Booleans are either one of two states, true or false. Booleans are often used in expressions which return true or false such as relational operators. Relational operators are used to compare

values. For example, we can check if one variable equals another value which will return true or false based on the values compared. The example below checks whether the variable num is equal to 2 which would return false.

```
int num = 1;

bool check = 1 == 2;
```
Operation returning False

Figure 13 Booleans

Enumerations

Enumerations are a list of named constants that can be assigned to variables as a type. These names, like items in a list, have an integer value. Using enumerations allows for easier selection of those integer values. For example, each month of the year can be represented by a name, August for example, or by an integer, 8 for 8th month of the year. This is a simple way of associating names with an actual value.

```
enum Month { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
```
Enumeration Declaration

```
Month month = Month.Mar;
```
Assigning an Enumeration

Figure 14 Enumerations

C# Working with Data

Conditional Statements

Conditional statements allow us to change the course of our code based on whether an expression evaluates to true or false. We can do this using IF and ELSE conditional statements.

The IF statement will check if a value or expression evaluates to true and if so, perform some action. The action is any code input inside of the if statement scope defined by the curly brackets. In the example below, the if statement is checking if the variable num is equal to 2, and if that is true, which it is, it will change the variable to 4.

```
int num = 2;

if (num == 2)
{
    num = 4;
}
```
IF Statement

Figure 15 IF Statement

Looping

Looping is a programming concept that allows us to loop, or iterate, over a collection of objects. This allows us to perform a function, or set of operations, for every item in a list.

One type of loop is the foreach loop. The syntax to create a foreach is shown below. This requires the keyword foreach followed by a set of parentheses, inside the parentheses which requires the object type that is being looped over, followed by a temporary variable which will be applied to each object in each loop, followed by the keyword in and then the list to loop over, in this example cats. As this is a statement, we then use curly brackets where is where we can write code that will be executed each loop over the list.
This is very useful for performing an operation on many objects. In each loop, in the example below, the cat's color is retrieved and appended to a list called colors. Once the loop ends, the color of every cat object in the cats list will have been appended to this colors list.

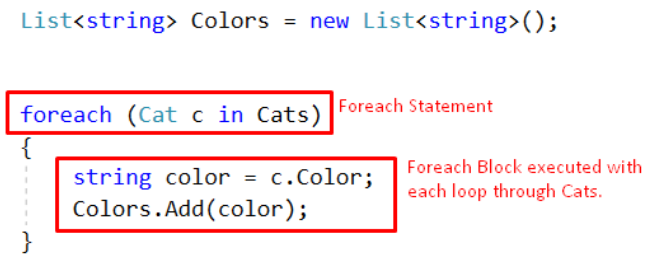


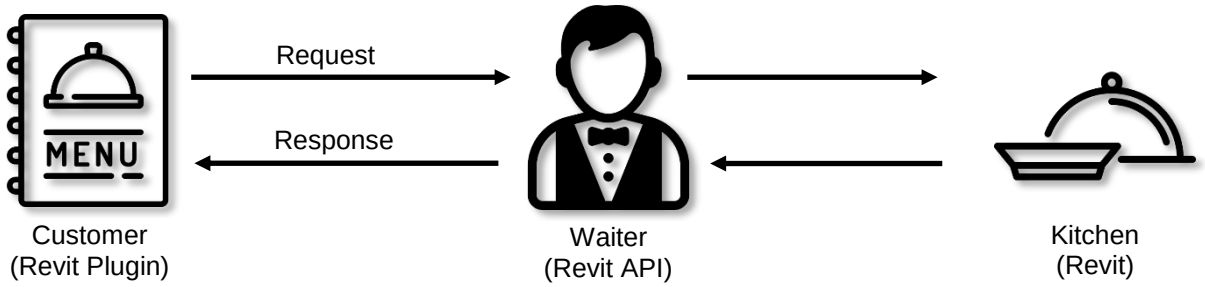
Figure 16 Foreach Loop

Revit API

What is an API?

An API, or Application Programming Interface, is a set of operations provided by a piece of software, web application or web service that allow other applications to interact with them. This allows for different applications to essentially communicate with each other.

One way of thinking about an API is like a waiter in a restaurant. If the kitchen were an application, like Revit, and we are the customer, an outside application, the only way we could get food from the kitchen would be via a waiter, or API. For us to get an item from the menu would be to request it via the waiter who would go to the kitchen, retrieve the item and then bring it to us. API's work in much the same way.



The Revit API comes with Revit and allows us to create applications that communicate with Revit through classes and methods accessed through our plugins. For example, if we wanted to create a view from our plugin, we would access the view class in the Revit API and call the constructor methods to construct a view.

Viewing the Revit API

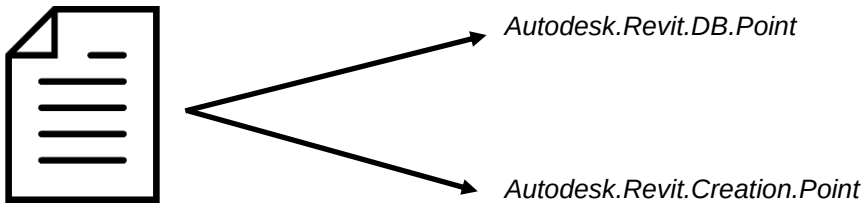
There are a few different ways to view the Revit API, these include:

- **RevitAPI.chm**: The RevitAPI.chm file that comes along with the install of the Revit Software Developer Kit. This is found in the utilities section when installing Revit and provides a windows-based documentation file for the Revit API.
- **RevitAPI.dll**: The RevitAPI.dll is the actual Revit API file that is used when accessing the API. This can be attached to developer environments and viewed directly which we will do later in the lab.
- **RevitAPIDocs**: The online resource *revitapidocs.com*, developed by Gui Talarico, provides an online view of the RevitAPI which is easy to access and search.

Namespaces

Within the RevitAPI library, the code is grouped in to different areas known as Namespaces. Namespaces are used to group related code into sections identified by their preceding namespace. This allows classes of the same name to occur more than once, differentiated by their namespace.

For example, say there were two Point classes in the Revit API library. These Point classes could exist in the one library if they were in different namespaces. These may look like:



Revit API

In the first instance, the Point class resides in the *Autodesk.Revit.DB* namespace and the other is in the *Autodesk.Revit.Creation* namespace. We can access either namespace the access one of the two Point classes.

There are many different classes accessible in the Revit API through quite a few different namespaces. To access the different namespaces in the API and use the classes, we need to utilize the **Using** keyword in C#.

This is done by using the keyword using, followed by the namespace to import from, as shown in the example below. Now all the classes from that namespace will be available, such as the Point class.

```
using Autodesk.Revit.DB;
```

Figure 17 Namespaces

Setting up a Revit Plugin

Now that we know some of the basics about C# and APIs, let's look at setting up the environment which will act as the scaffolding for our plugins.

Starting Visual Studio

To develop plugins, we need an IDE, or Integrated Developer Environment, in which we can write our code. An IDE is basically a tool that assists in writing and testing software. For these exercises, we will be using Visual Studio (VS), a free IDE from Microsoft.

Let's start by creating a new project. This can be done by navigating to the file menu and new > project.

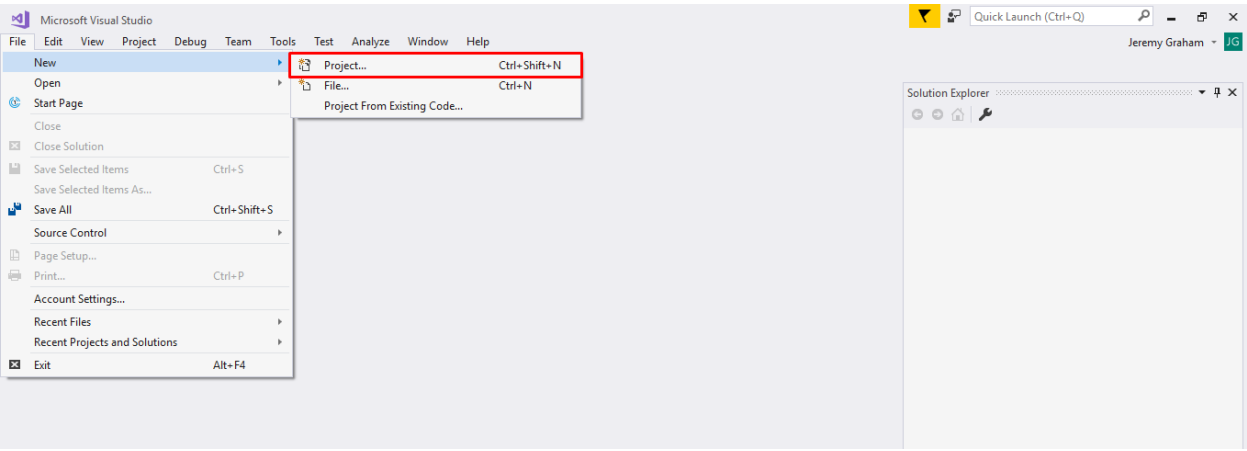


Figure 18 Visual Studio Project Start

When creating a new project, VS offers many templates depending on what type of application you may be developing.

For a Revit plugin, we will simply be creating a new class library as Revit accesses the commands we build through C# classes. The C# class template we will be using is found in the Visual C# section of the templates and is marked with .Net framework as Revit requires plugins to target the .Net Framework. **Therefore, select Class Library (.NET framework) as shown below.**

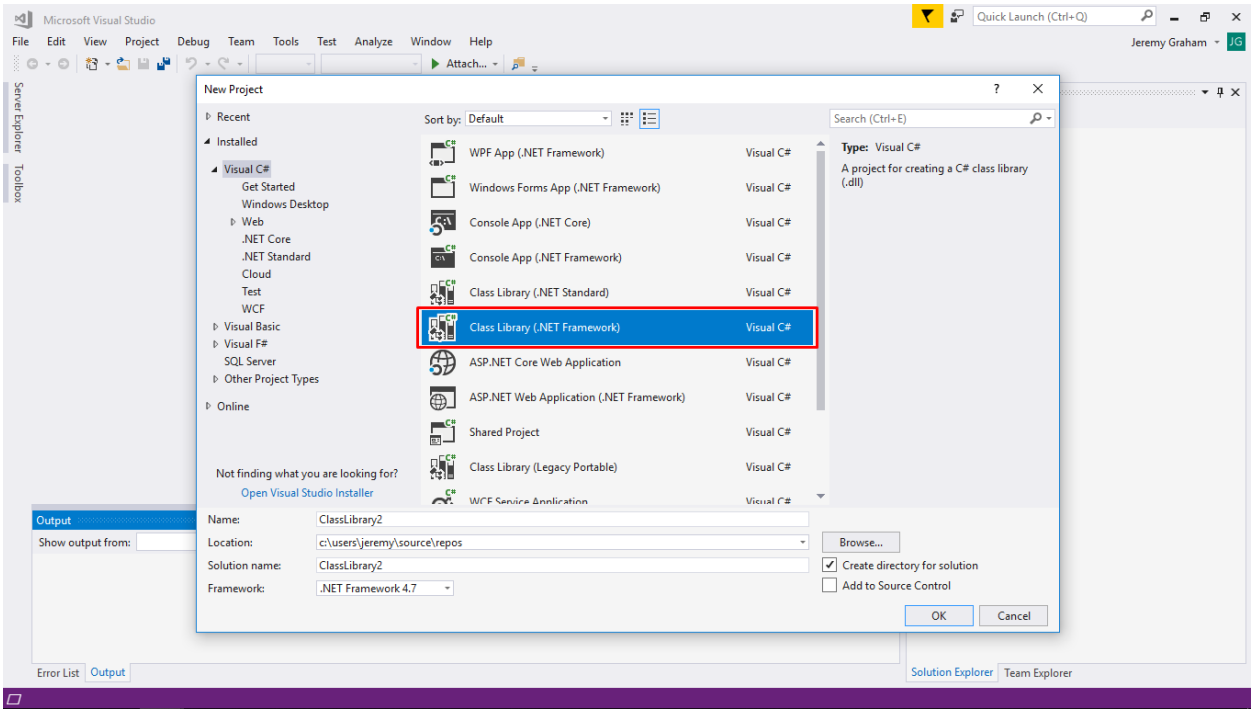


Figure 19 Class Library

Our plugins need to target the .Net framework as Revit utilizes this framework. The version of .Net framework that Revit uses varies between releases. Therefore, we need to ensure that whatever .Net framework we target is suitable for the Revit version we are developing for. The different Revit versions and the .Net Framework they are utilizing is shown below.

Revit Version	.Net Framework Version
Revit 2019	.Net 4.7
Revit 2018	.Net 4.6
Revit 2017	.Net 4.6
Revit 2016	.Net 4.5
Revit 2015	.Net 4.5

The .Net Framework is backward compatible so a plugin targeting .Net version 4.7 will work for all current Revit versions.

As we are building a plugin for Revit 2019, let's leave the targeting framework to 4.7 as shown below. So next, **make sure create directory for solution is selected and Add to Source Control is not. Then set the project name to the plugin name you would like, such as *AUPlugin* and hit OK.**

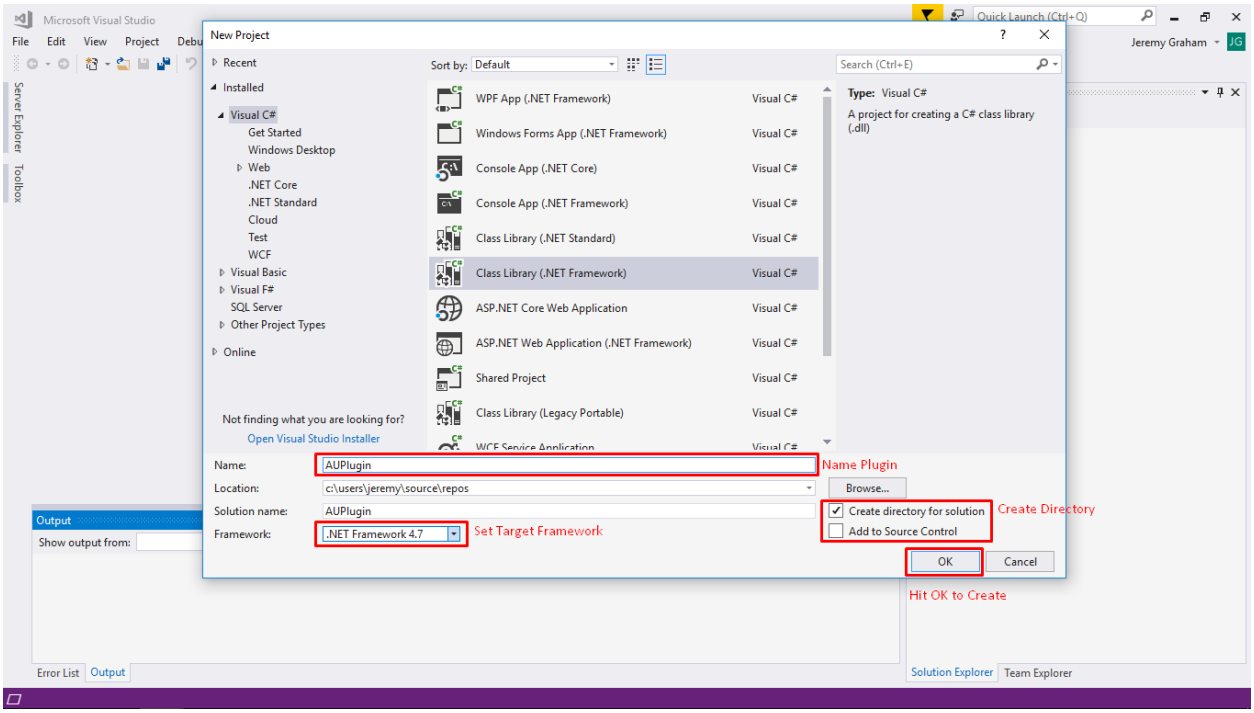


Figure 20 Create Project

The plugin project file is now setup and contains one class. We will use this class to create our first plugin command. The first command will simply show a dialog, so let's **rename the class ShowDialog by right-clicking the class name and selecting rename.**

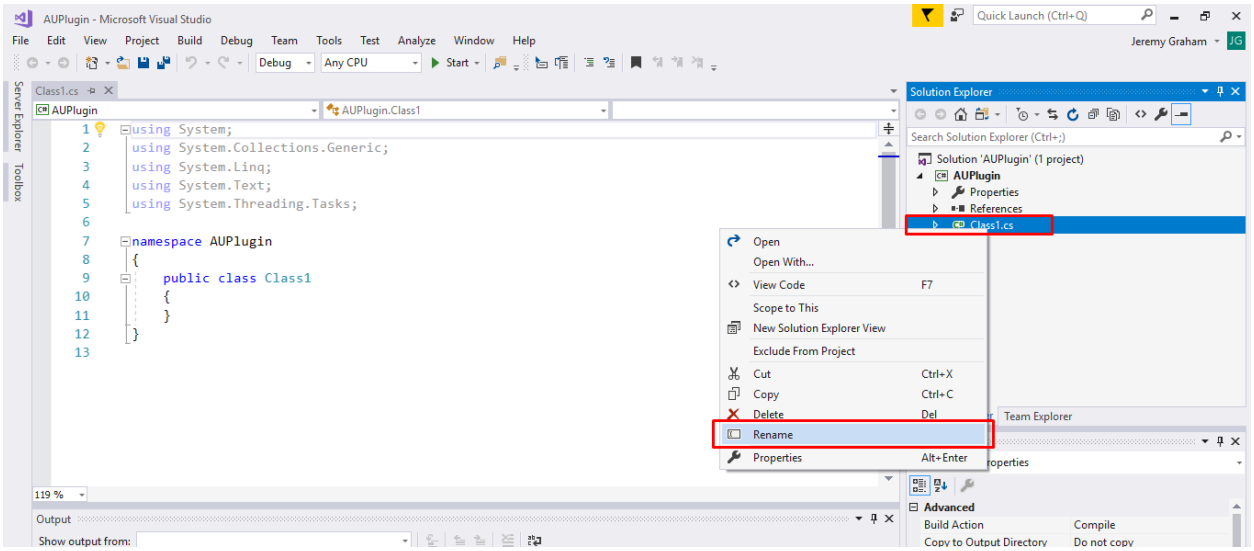


Figure 21 Rename Class

When prompted to rename all references of the name, select Yes.

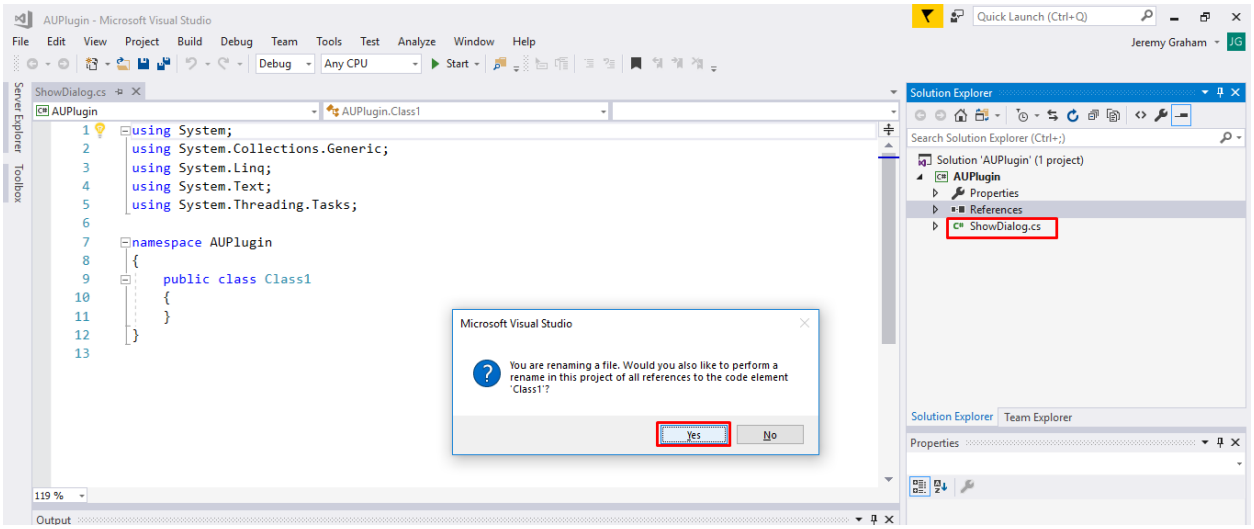


Figure 22 Rename All References

Adding the Revit API

In order to work with the Revit API, we need to reference the Revit API files, so we can access its contents. There are two files we need to reference:

- **RevitAPI.dll**: This file provides access to Revit at a database level which includes all the classes and functions required to create and work with Revit elements.
- **RevitAPIUI.dll**: This file provides access to the Revit user interface which includes all the classes and functions required to add interface elements.

Both files are .dll or Dynamic Link Library files which are files of compiled code and come with the Revit install. These files can be referenced in to our project, so the code is accessible from our project. **Let's add the Revit API files to our project, in the solution explorer tab, right-click the References section and Reference from the Add submenu.**

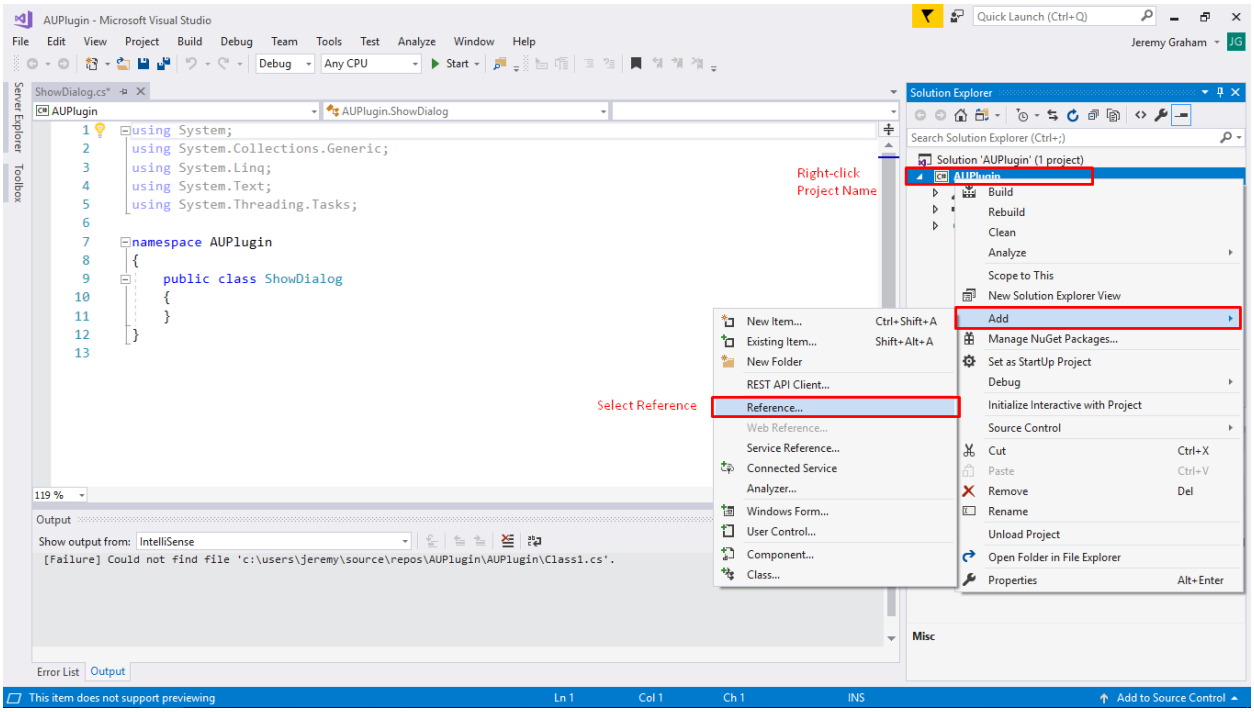


Figure 23 Add Reference

This provides access to many libraries that can be added as part of the .Net framework. We will need to browse to add the Revit API files so, **select the Browse button on the bottom right.**

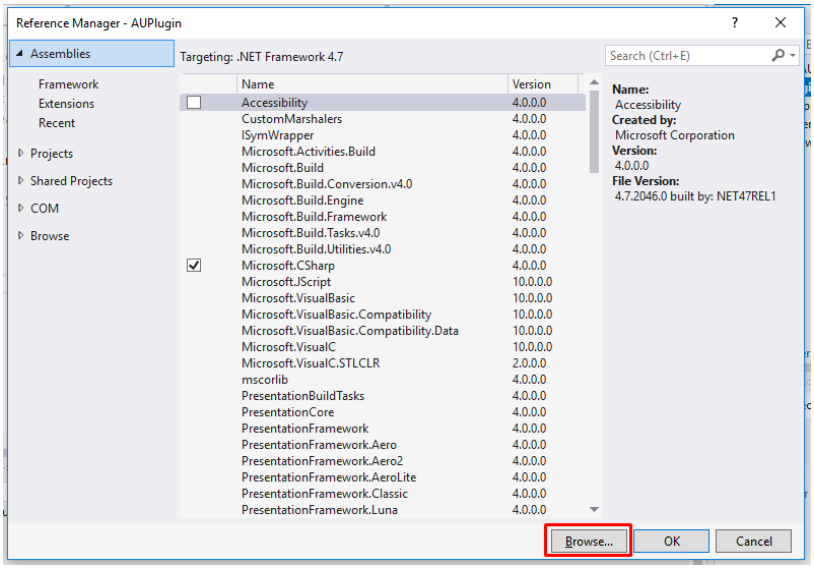


Figure 24 Browse for References

Now we can navigate to Revit API files. These are found in the default install location for Revit which is - *C:\Program Files\Autodesk\Revit 2019*\ . In this file, select both the RevitAPI.dll and RevitAPIUI.dll files by hold ctrl and left click then hit the add button.

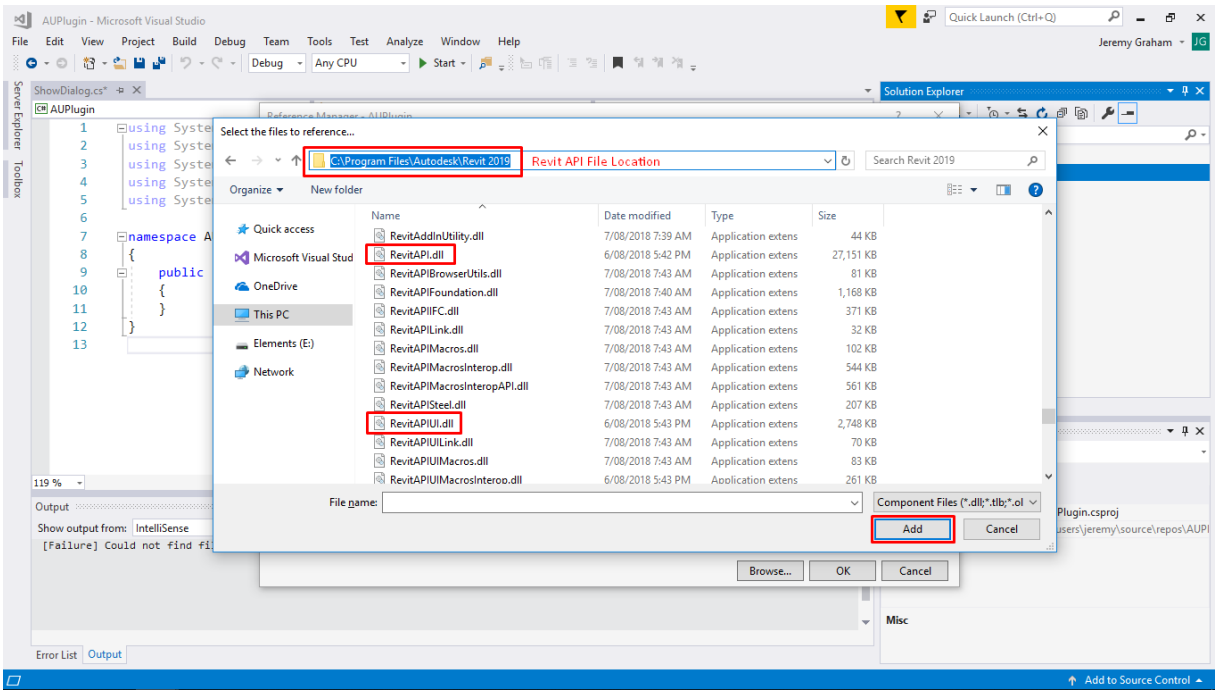


Figure 25 Add API Files

Then with both of these selected in the references windows, select Ok. Both files are now accessible from our Revit Plugin.

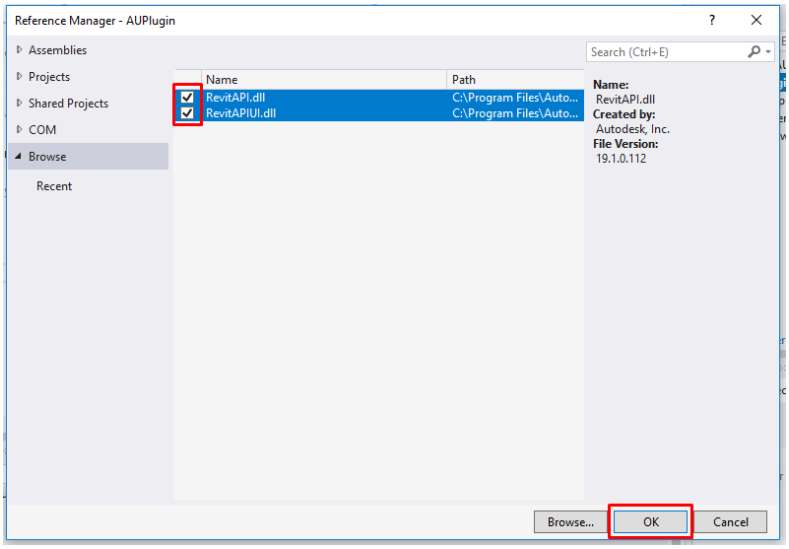


Figure 26 Add Selected References

Once the two API files are added, we need to ensure they aren't copied to the output folder when we compile our code as when we execute our code in Revit, Revit does not need a reference to the Revit API, it already has reference to the API. **To not copy the files when we compile, select both files in the solution explorer and change *Copy local* to false.**

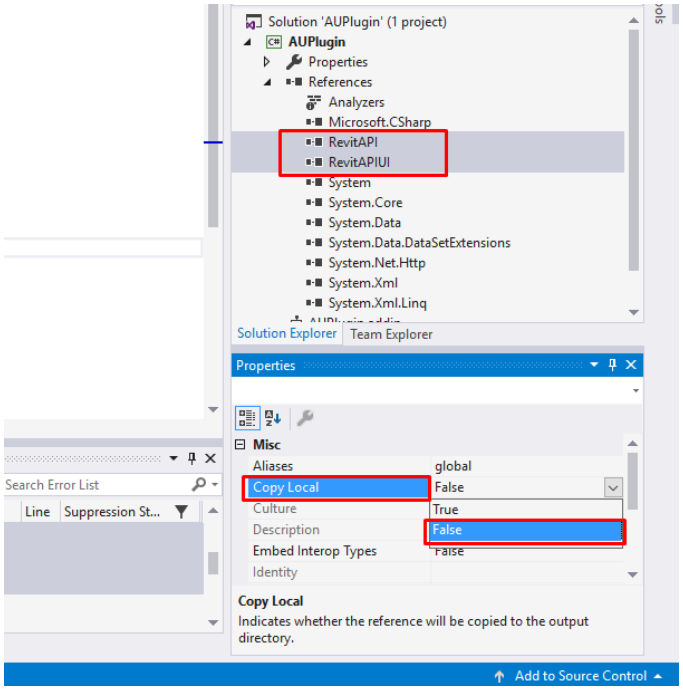


Figure 27 Copy Local False

Accessing the Revit API

At this point in our plugin, we have an empty class file as shown below. The structure of a class file starts with importing library namespaces into the class so that the classes and functions from that library are available within the class.

There are already some default imports that are commonly used in classes such as the *System.Collections.Generic* namespace which provides access to the List class. This is imported with the using keyword followed by the namespace to access.

Let's start writing our command by importing the necessary Revit namespaces so we can access the classes we need to. These are the *Autodesk.Revit.DB* namespace and *Autodesk.Revit.UI* namespace, as shown below.

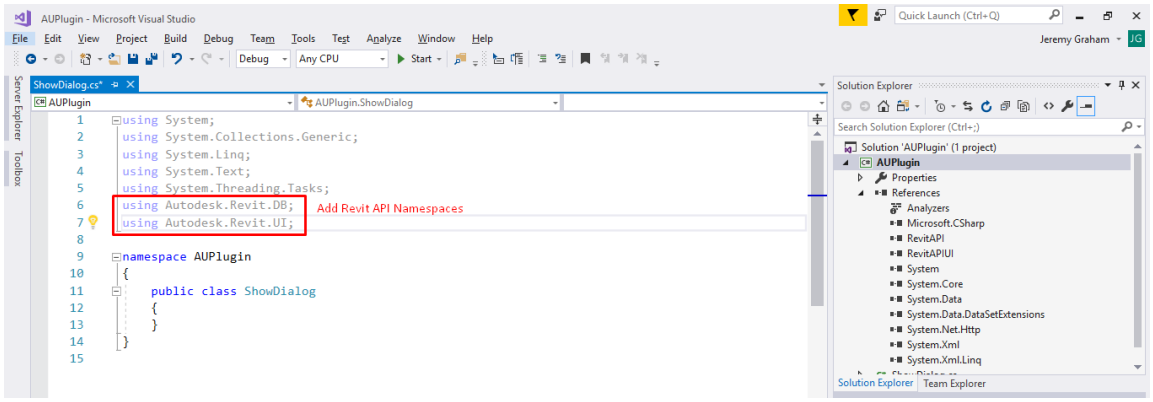


Figure 28 Adding Revit API Namespaces

The IExternalCommand Interface

When Revit loads a Revit plugin, it will access classes in the plugin that implement the IExternalCommand and turn them in to Revit commands. By implementing, this means any class that inherits, or adopts, the methods of an interface. An interface acts as a blueprint for classes by providing a set of methods that a class needs to implement or have in the class.

To implement the IExternalCommand with the class we just created, we need to do so in the class file. **Make sure the class file is open, as shown below, and after the class name is declared on line 11, add colon then IExternalCommand as written below.** You may also notice the *Autodesk.Revit.UI* namespace has lit up, this is because the IExternalCommand interface is in the *Autodesk.Revit.UI* namespace.

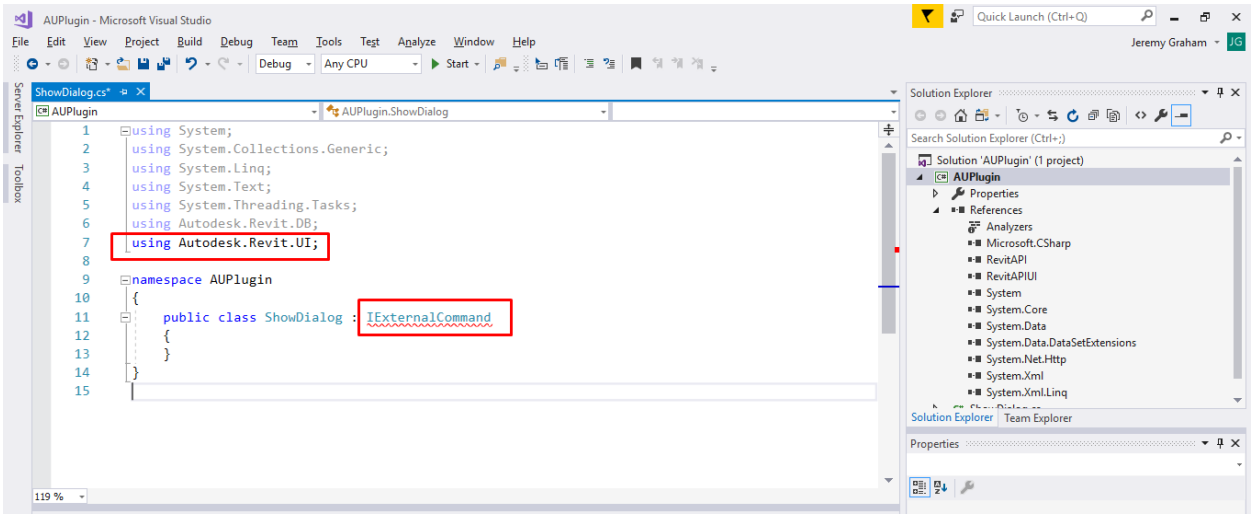


Figure 29 Implementing the IExternalCommand

The command now utilises the IExternalCommand Interface. This will cause an error as can we see with the red line under the IExternalCommand which has occurred because the class does contain the methods required by the IExternalCommand Interface. **To implement these methods, right click the IExternalCommand name and select Quick Actions and Refactorings which will suggest fixes for our error. One of these is *Implement Interface*, select this to automatically create the required method of the IExternalCommand which is the Execute method.**

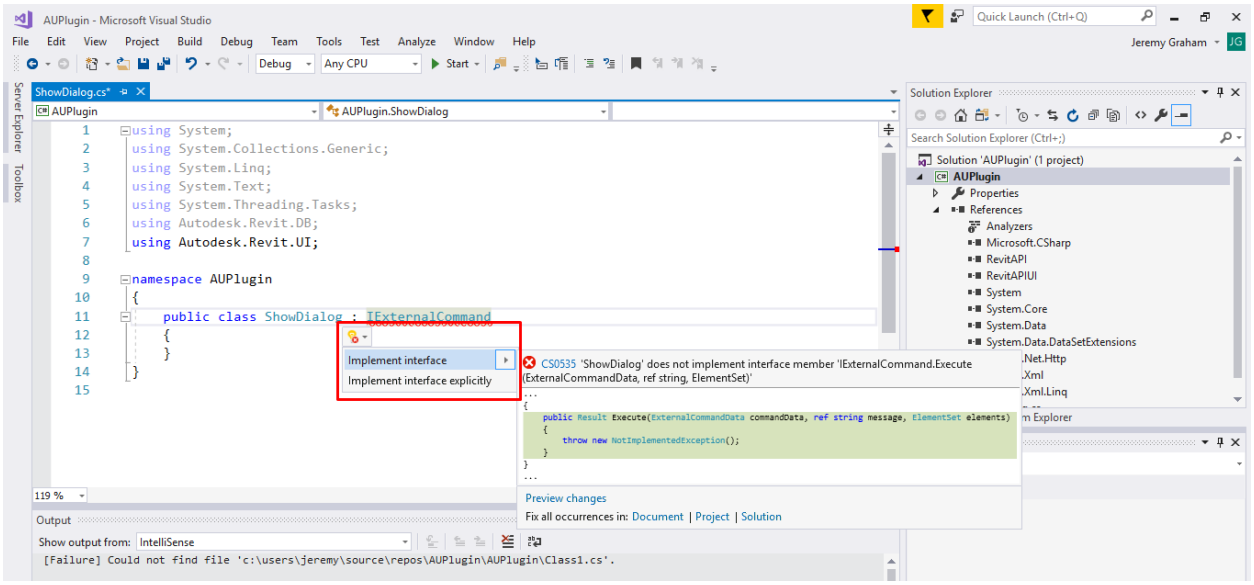


Figure 30 Implement IExternalCommand

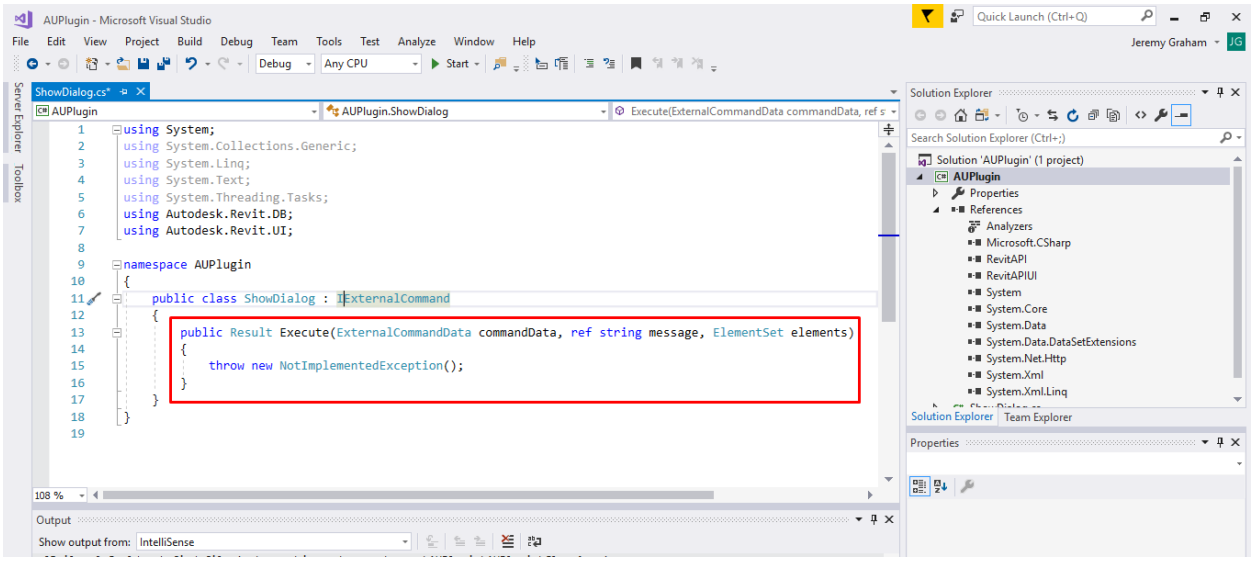


Figure 31 Implemented Execute Method

Execute Method

The Execute method is the single method required by the IExternalCommand Interface. This method is executed by Revit when we call our command. So, any code within this method will run.

The method takes three parameters which are all passed to the method by Revit when it runs. These are:

- **ExternalCommandData**: This object contains reference to the Revit Application and view required by the external command. We can therefore access all Revit data through this object.
- **ref string**: This can be used to relay a message back to the user if the command fails or is cancelled. If the command is cancelled or fails, and the string parameter has been set in our command, this will be returned to the user as a message. If it is not set, then the command will simply exit. The ref keyword simply marks the string as a reference, meaning that updating it inside of the Execute method, updates a string variable outside of the method scope, stored by Revit. If it is updated inside of the method, it will update the variable stored by Revit.
- **ElementSet**: This acts as a list of Revit elements and can be used to display elements back to the user. If the command fails, and elements have been added to the elementset in our command, these will be highlighted to the user.

The execute method does not necessarily need to make use of the string and element set parameters however, these are useful for relaying information back to the user.

The Execute method needs to Return a result, as marked in the method declaration. This is a type of enumeration which is either Succeeded, cancelled and failed. Succeeded means the command executed successfully, cancelled means the user cancelled the command and any string set to the input string parameter will be sent back to the user. And failed means there was an unhandled error in the command and any string and elementset set to the input parameters will be returned to the user.

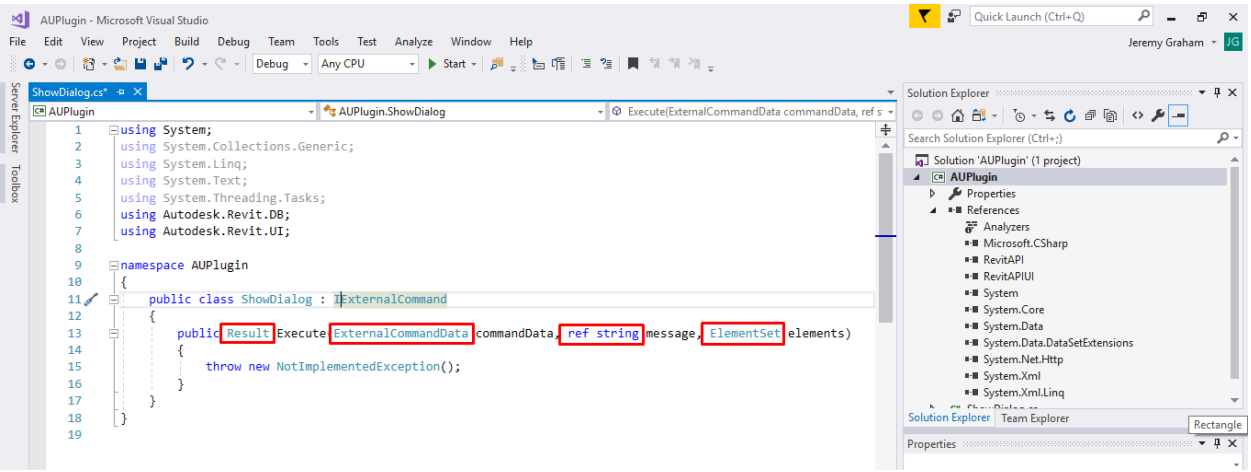


Figure 32 Execute Method Declaration

Attributes

There is one more thing we need to add to the Execute method before we add code to the function, and that is adding a Transaction Attribute tag. Attributes are simply tags that define information about the class and can affect it's behavior when it runs. The Transaction attribute tag is used to tell Revit how the class should work with Transactions. Transactions are used in Revit to make changes to the model which we will learn more about later. For now, let's set the transaction behavior of our class to Manual which means we can make a transaction with the Revit model, however we won't be doing so until the third exercise.

To do this, on the line before the class declaration, we need to mark the class with the attribute in square brackets with *[TransactionAttribute(TransactionMode.Manual)]*. This is shown below.

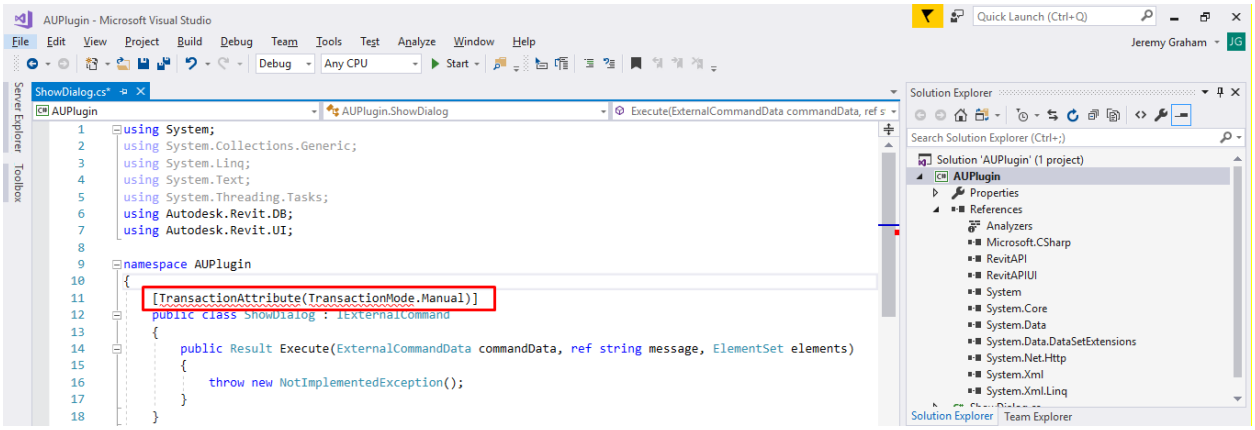


Figure 33 Adding Transaction Attribute

This will appear as an error as the correct namespace needed to be access from the command. **By clicking the lightbulb again, select to automatically add the Using *Autodesk.Revit.Attributes* namespace to our command.**

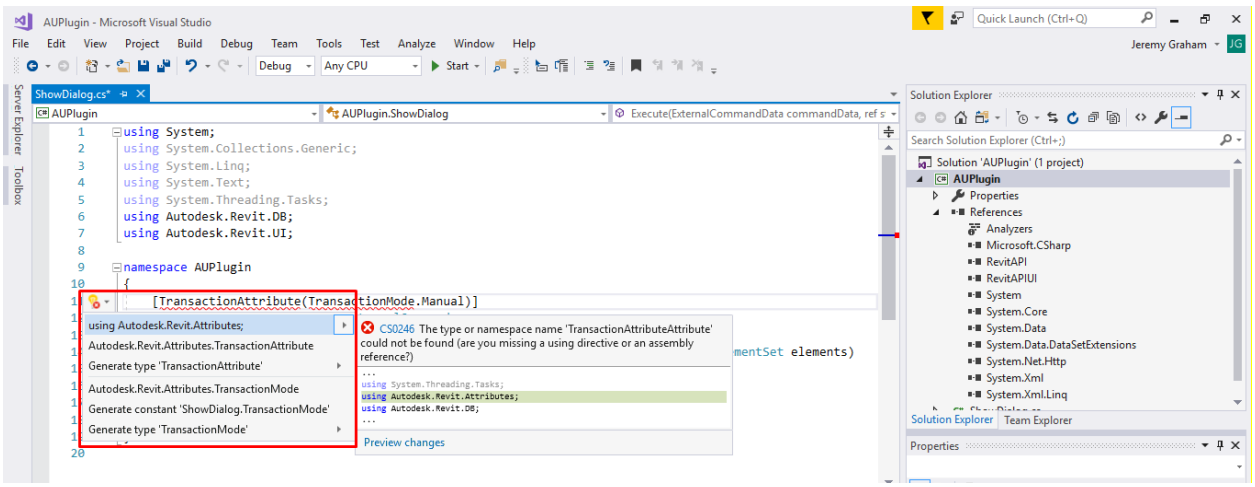


Figure 34 Add Transaction Using Statements

TaskDialogs

The first command we will create will simply display a task dialog. The TaskDialog class from the Revit API allows us to display a dialog to the user in the style of Revit dialogs.

Let's add this into our execute method to run when we run the command. **First remove the line which says throw new exception on line 17, and let's replace it with a call to create a new TaskDialog. To display a TaskDialog, access the Show method from the TaskDialog class as shown below.**

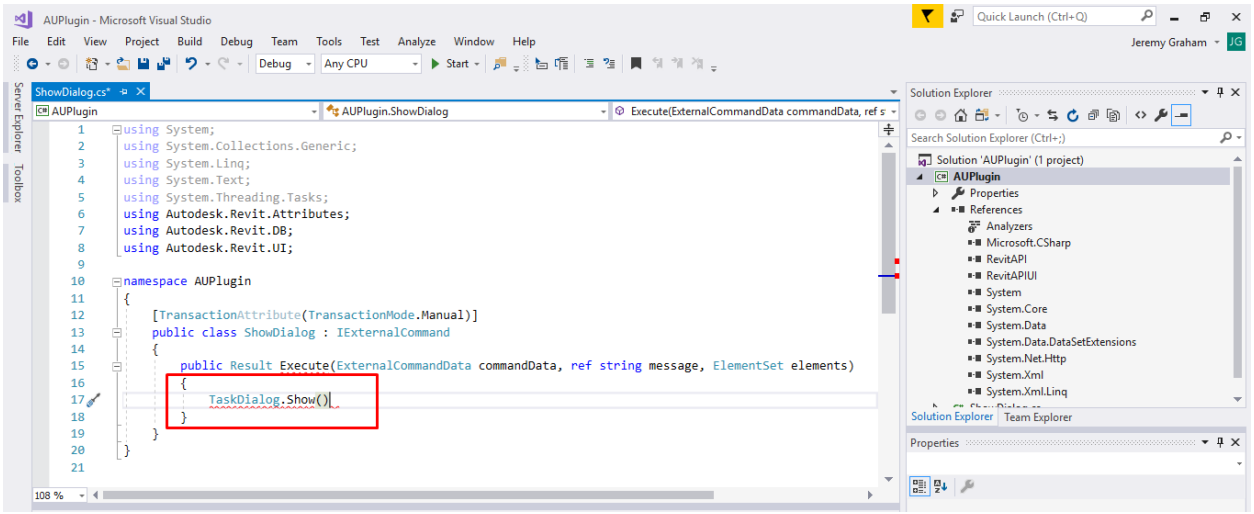


Figure 35 Show TaskDialog Method

This takes two parameters, the name of the dialog window and the message to show in the dialog, both which are strings. For this example, let's use the name "*MyFirstPlugin*" with the message *"My first Revit Dialog!"*.

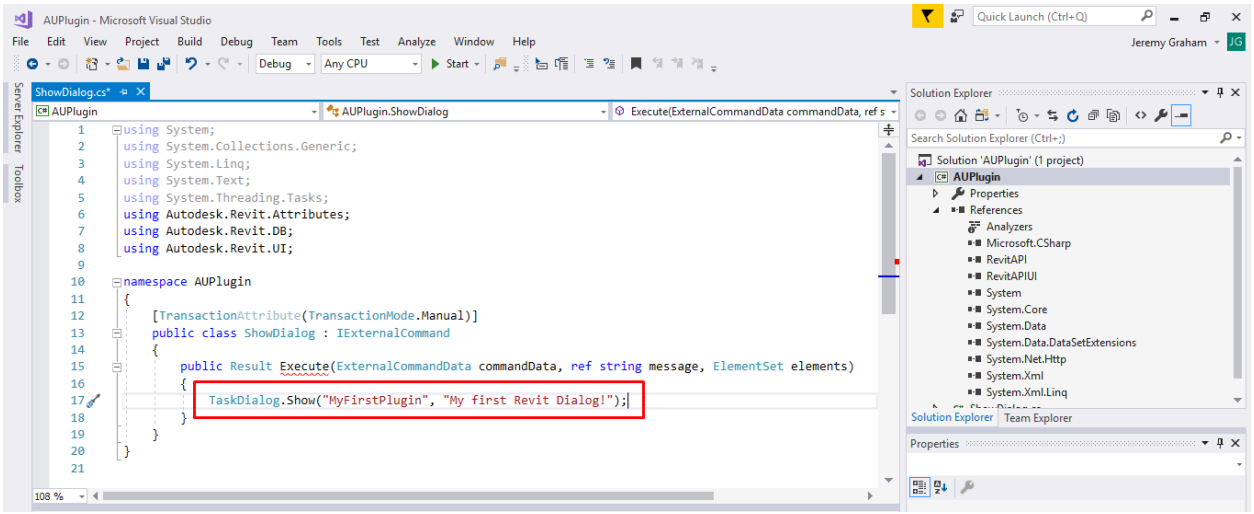


Figure 36 Add TaskDialog Parameters

After the dialog is shown successfully, the command is finished so we need to return a successful result by using the return keyword, followed by the Result enumeration Succeeded, after the TaskDialog is shown.

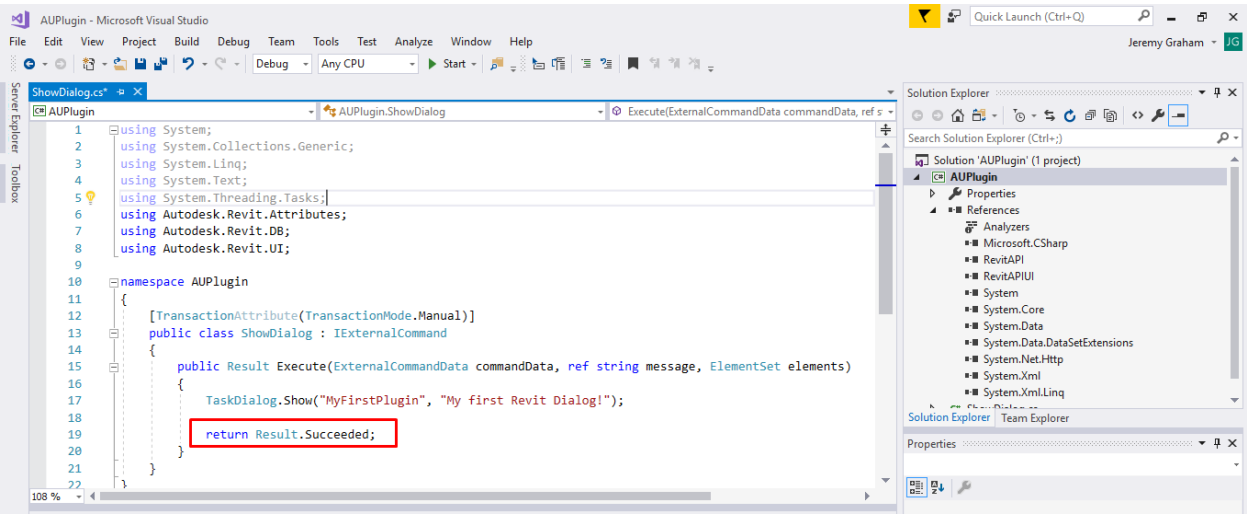


Figure 37 Returned Succeeded Result

Manifest

Now that we have coded our first command, we need to register the command for it to show up in Revit, we do this by creating a manifest file. When Revit boots up, it will read manifest files located in one of two specific files to determine what plugins to load and with what options.

To create the project manifest right click the project name and select add > new item.

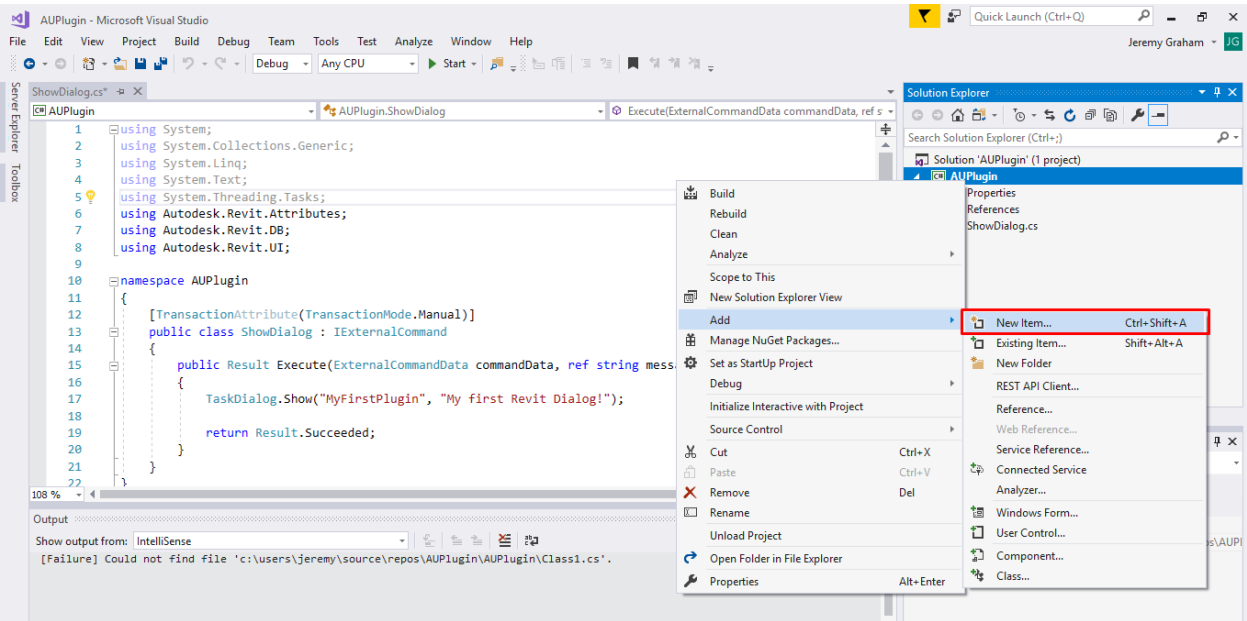


Figure 38 Add New Item

From the list of items to add available, select Application Manifest file and rename it to the same name as the plugin so Revit knows that they are related. The extension of the file also needs to be changed to *.addin* as this is the file type Revit requires.

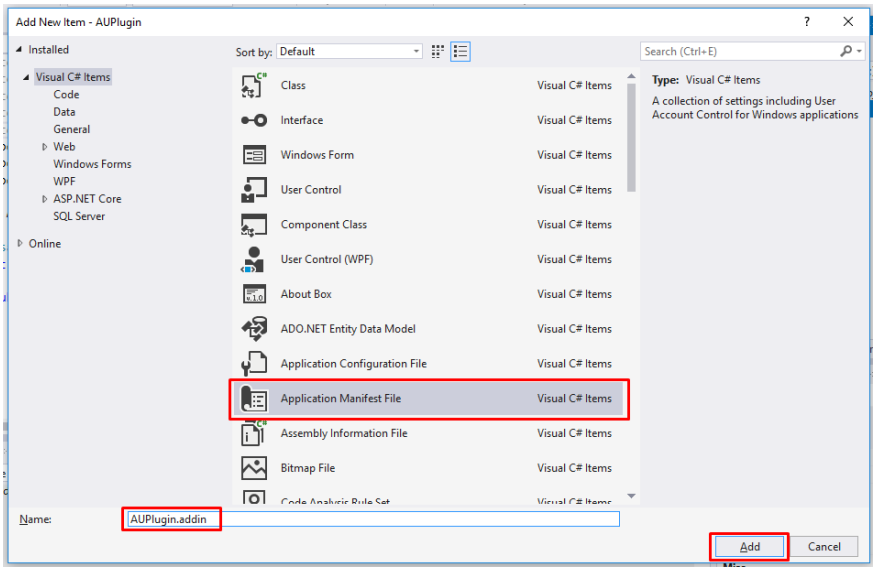


Figure 39 Add Manifest File

Once the addin file is created, double click the file to open it up. This is an XML file which is a language made up of markups, or tags. Remove the default text from the file so we can put in the code in required by Revit addins. This can be found in the txt file named manifest.txt in the exercise files. **Copy the text from the file manifest.txt and paste it into the new addin file.**

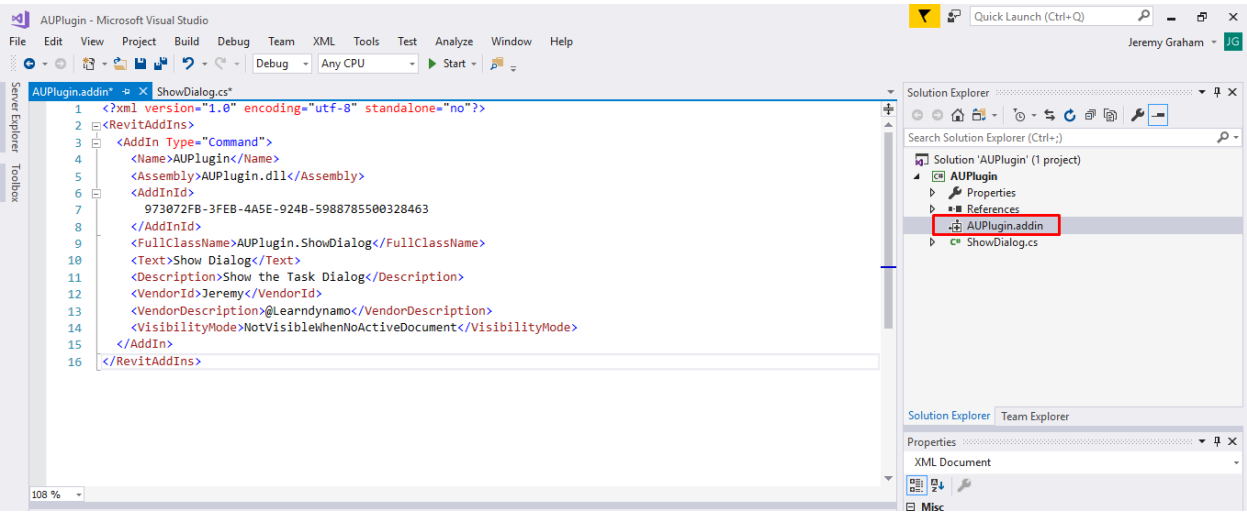


Figure 40 Replace Manifest File Code

The addin file now contains a series of tags that Revit will read when the plugin starts up. These are not the only tags we can add, there are many more however, these will provide the information Revit needs:

- **<RevitAddIns>**: Indicates a set of tags relating to RevitAddins.
- **<AddIn Type="Command">**: Specifies that the type of addin being loaded is a command.
- **<Assembly>**AUPlugin.dll**</Assembly>**: Specifies the .dll file that corresponds to this manifest file. This is so Revit knows which file needs to be loaded. This can be a full file path to a seperate file. No file path indicates the dll file is in the same location as this manifest file.
- **<AddInId>**76eb700a-2c85-4821-a78d-31429ecae9ed**</AddInId>**: A GUID which needs to be unique for each command.
- **<FullClassName>**AUPlugin.ShowDialog**</FullClassName>**: The class that this command relates to including the namespace.
- **<Text>**ShowDialog**</Text>**: This is the text that will appear as the name for the command in Revit.
- **<Description>**Show a Dialog**</Description>**: The description that will appear with the command in Revit.

More information about these tags and additional tags that can be added can be found here:

http://help.autodesk.com/view/RVT/2019/ENU/?guid=Revit_API_Revit_API_Developers_Guide_Introduction_Add_In_Integration_Add_in_Registration_html

Lastly, we need to ensure the manifest file is included when we compile our code. **To do this, select the manifest file and in the properties window in the bottom right corner, *change Copy to output directory* to *Copy if newer*.**

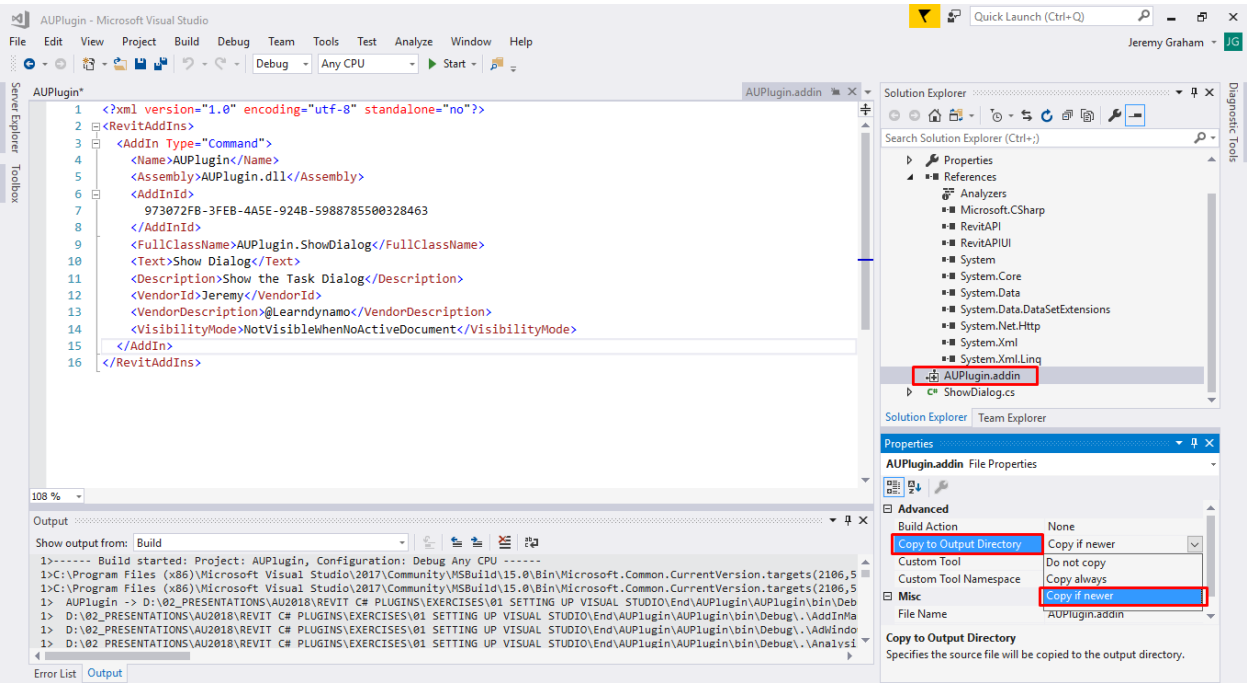


Figure 41 Copy Manifest

Building the Command

Once the command and manifest file have been completed, it's time to build the project to create the DLL file that Revit can run. To test our code, this is best done in Debug mode which will let us know if any errors occur in our code. To do that, we need to change some options in the Visual Studio file.

Navigate to the Project tab > AUPlugin Properties.

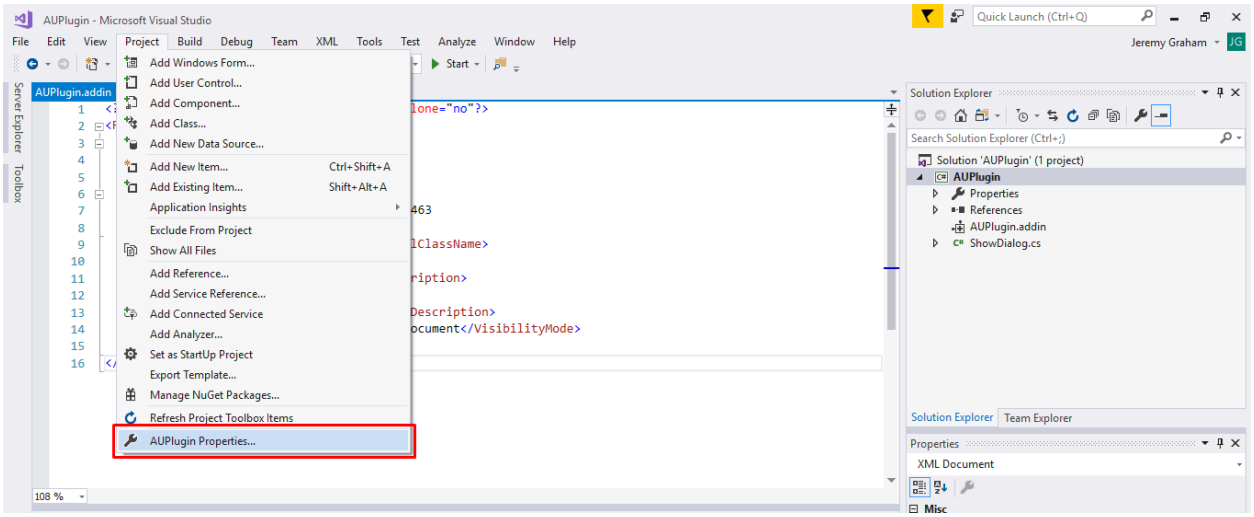


Figure 42 Plugin Properties

In these properties, we can add an external program to start up when we debug the code, we want Revit to start. **To add this program, navigate to the Debug tab, activate the Start External Program option and browse to the file path to the Revit.exe file. This is found at *C:\Program Files\Autodesk\Revit 2019\Revit.exe*.**

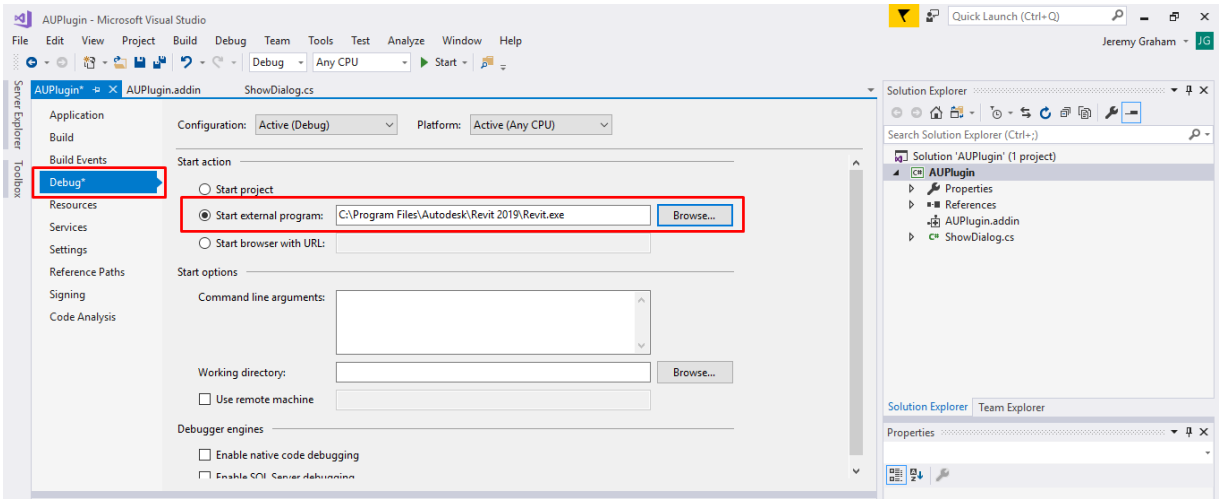


Figure 43 Add External Program

Now Revit will start up each time we debug our code. For Revit to load our compile code, we also need to ensure they are in the correct location when Revit loads. One of these files is *C:\Users\Jeremy\AppData\Roaming\Autodesk\Revit\Addins\2019.*

To automatically transfer the compiled code into this directory, we can add a build event. **Navigate to the Build Events tab and copy the text from *BuildEvent.txt* file in the exercise folder and paste it in to the Post-build event command line as shown below.**

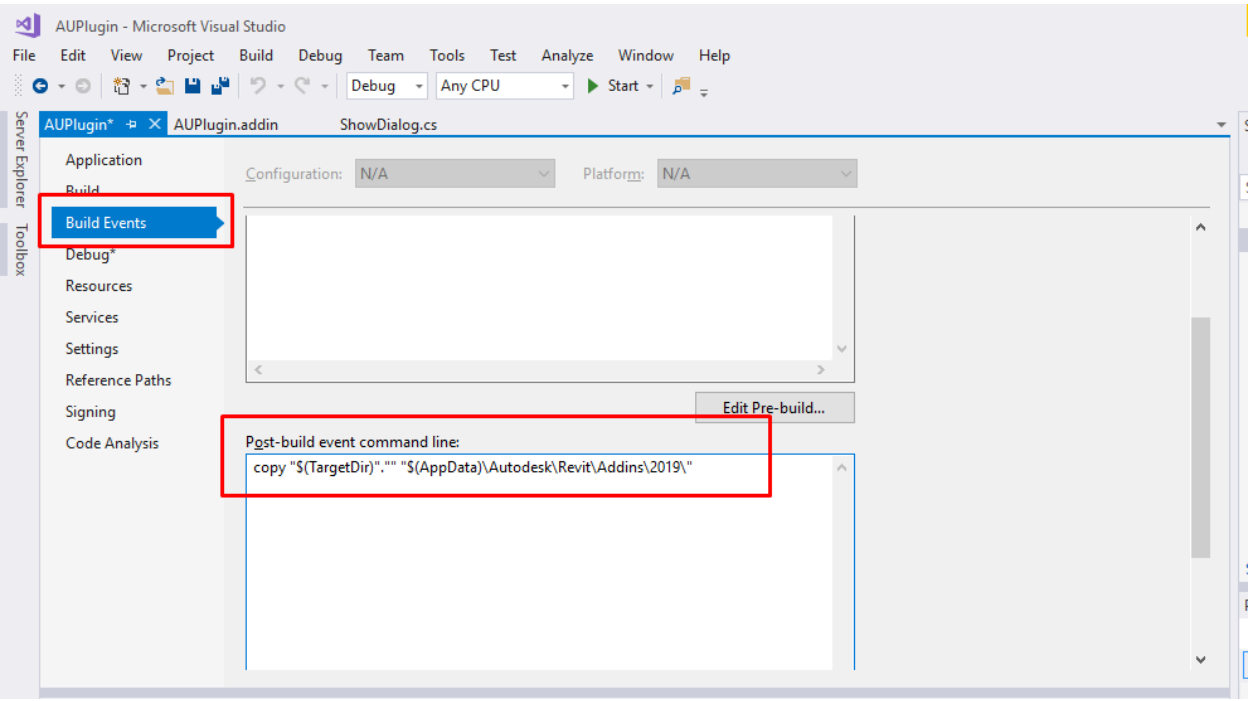


Figure 44 Add Post-Build Event

This will copy the dll file that our code compiles to, along with the addin file, into the folder that Revit checks when booting up. **So, all we need to do now is test the code by hitting the Start button at the top of Visual Studio.**

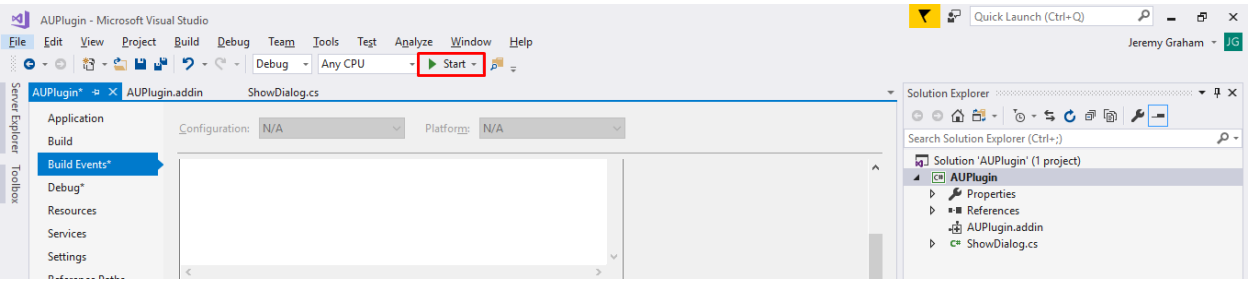


Figure 45 Start to Debug

During the boot up, a window will be displayed as shown below which is a warning about our plugin not being signed. **This is okay as we are not distributing our plugin so hit okay.**

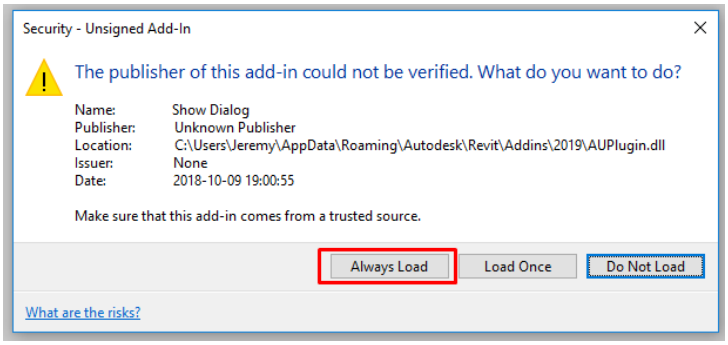


Figure 46 Load Plugin

Open a new Revit project and try the command which will be in the Add-ins tab under External Tools. If successful, the task dialog will be displayed as shown below and you would have created your first Revit plugin!

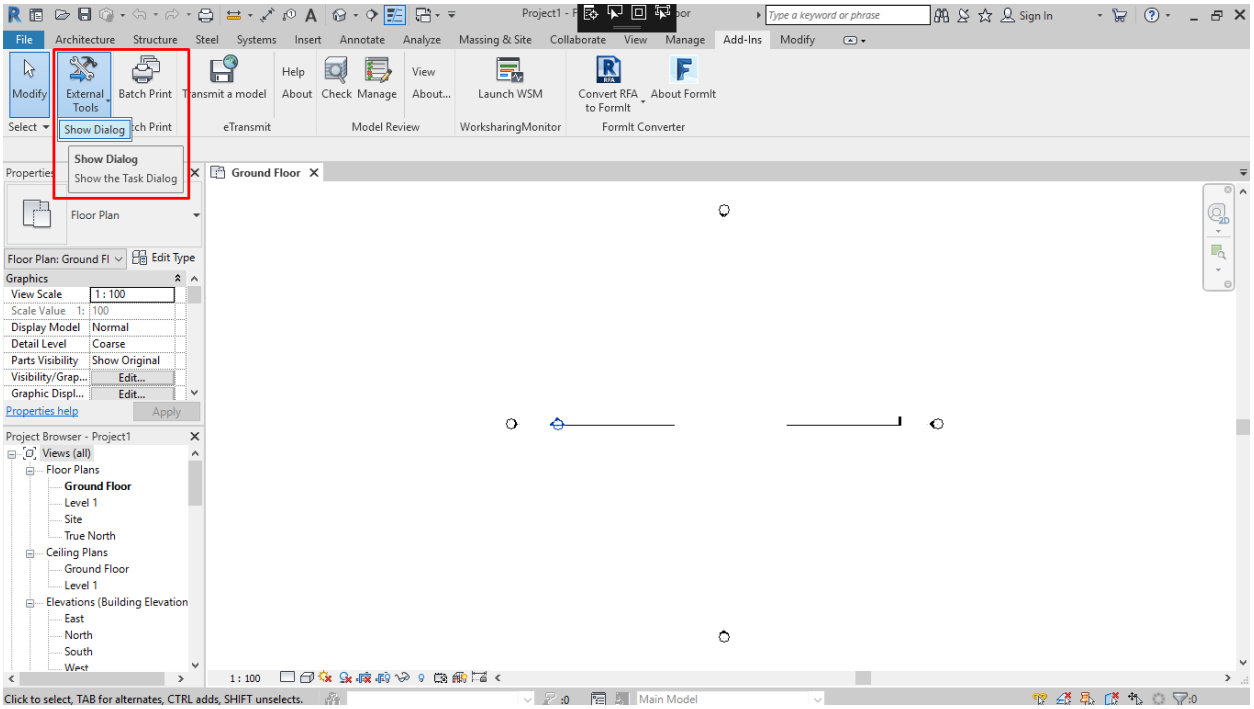


Figure 47 Run Command

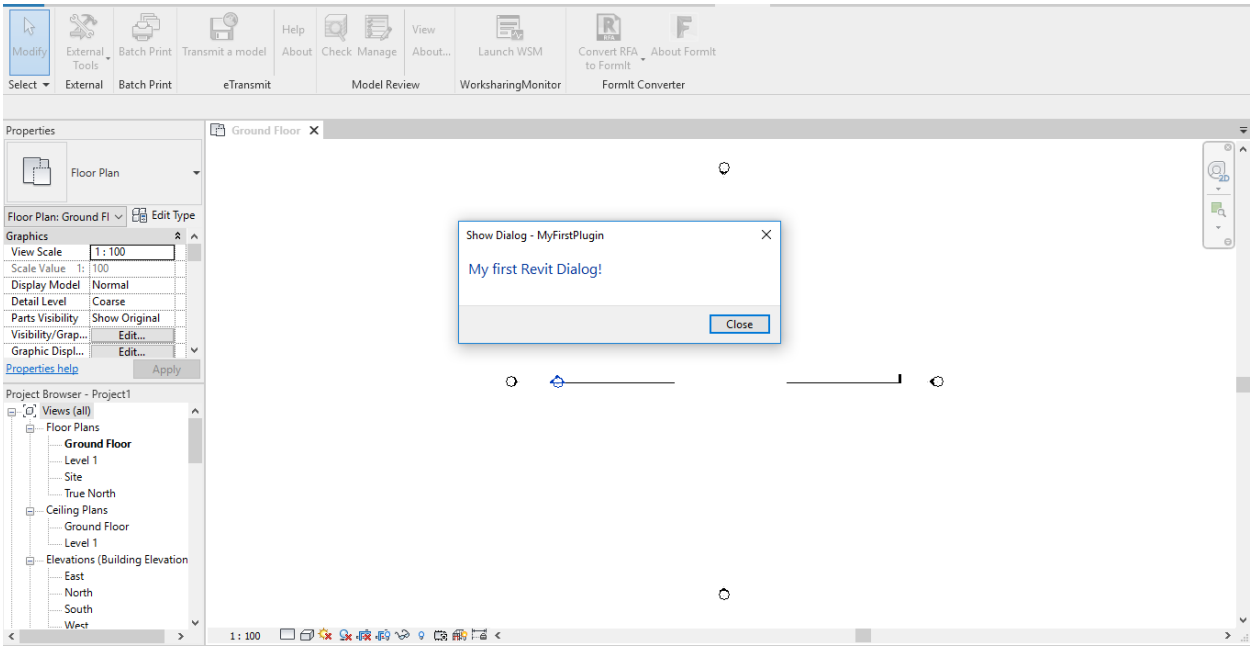


Figure 48 My First Revit Dialog

Revit Elements

When working in Revit, we create buildings made up of many different objects such as walls, windows and furniture. All of these objects in Revit derive from a base class in the Revit API known as the element class. We now know how to create plugins, let's dig a little deeper into the Revit API and look at working with Revit's base type - the element. In doing so, we will create another command in our plugin that allows us to filter the Revit document for all Windows. **Open up the Start Exercise file for the exercise named Filtering for Elements, this contains a class name FilteringElements which is current an empty command.** We will use this to start filtering the Revit document.

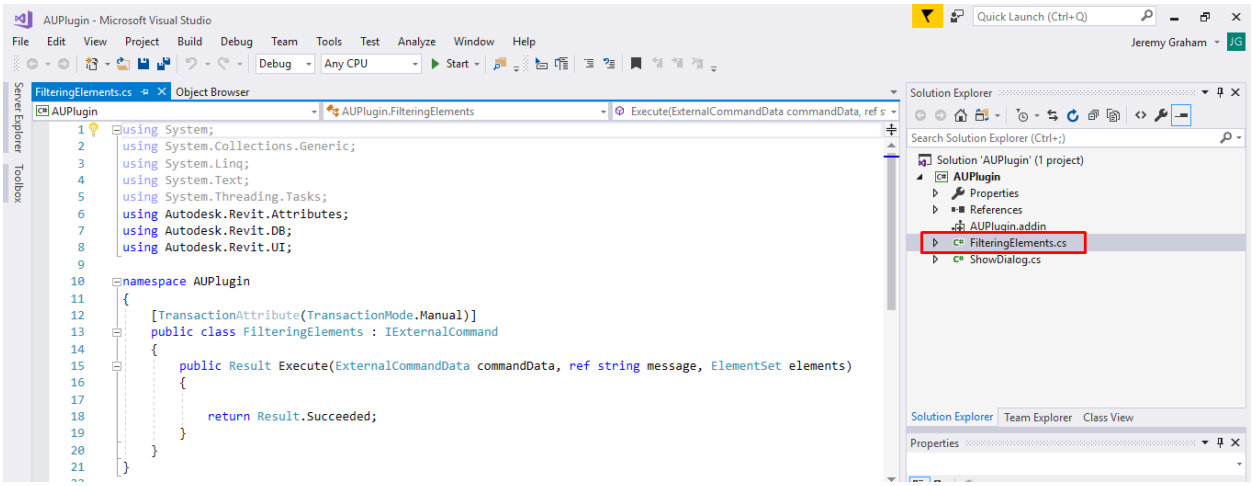


Figure 49 Filtering Elements Class

What are Elements

An element is any model or drawing component that makes up a Revit building model. Most classes, that define Revit objects, inherit from the element class. Therefore, we can access the same properties from objects in Revit to retrieve information about an element such as it's ElementId, parameters or family type.

We can view the element class properties and methods by searching for element in the object browser. This is found in the View menu >Object Browser.

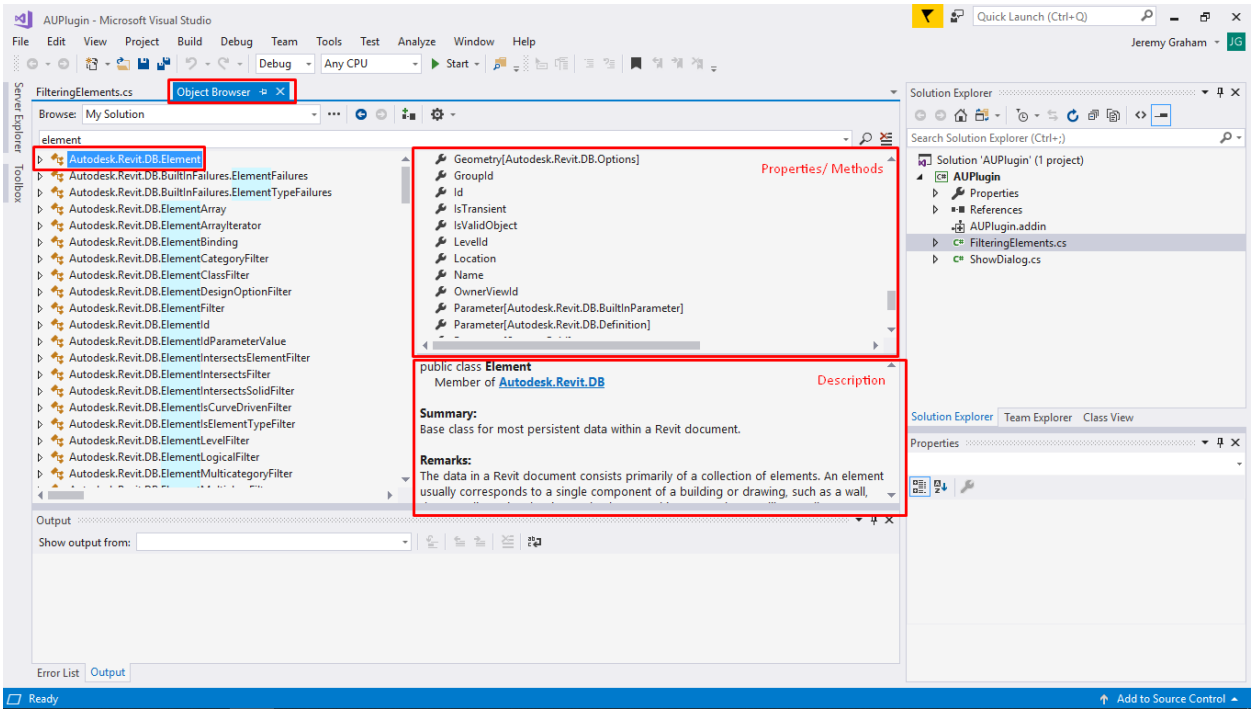


Figure 50 Element Properties and Methods

Classifying Elements

Elements can be classified in different ways, this includes by category, family, symbol or instance.

- **Category**: elements have a property that store the category it belongs to and is used to identify what the element type is. Categories describes group of similar elements in which the element type is associated, for example a wall will belong to the wall category or doors will belong to the door category. Every built-in category in Revit, that is default categories, will have an associated BuiltinCategory enumeration. For example, the wall category can be retrieved by its Builtincategory which is *BuiltinCategory.OST_Walls*.
- **Family Symbol**: In Revit, different elements are created by using families. Families are identified by their different family types. For example, the wall family may have several different family types which describe a different type of walls such as *Generic 100mm Wall* or *Generic 200mm Wall*. The different types are known as Family Symbols in the Revit API. Elements can be differentiated by these Family Symbols.

- **Family Instance**: A family instance is a constructed object of a family symbol. Think of this as a constructed object from a class or element in Revit such as a chair in the model. Each family instance will have its own instance properties such as i.d which can be used to differentiate elements in the Revit API.

Accessing Elements

To access elements within our plugin, we need to access the Revit project file that we are working in which is the Document object. We can access the current Document in the Revit API by accessing the ExternalCommandData that is passed to the Execute method in our plugin command.

To start, create a UIDocument variable named uidoc as the Document is referenced in a property of the UIDocument object. The UIDocument provides access to project level user interface methods and properties, such as refreshing the view, getting the selected elements or prompting the user to select an element.

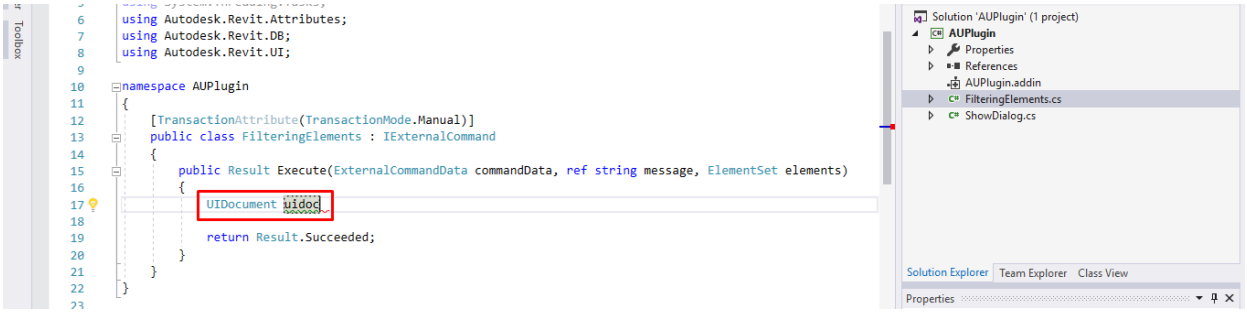


Figure 51 UIDocument Variable

To access the UIDocument from the ExternalCommandData, we first need to retrieve the Application from the ExternalCommandData object property. This then provides access to the UIDocument through the ActiveUIDocument property from the Application. The Application provides access to application wide settings such as project location settings, language and application events. **With this, we can access the Active UIDocument all in one line as shown below.**

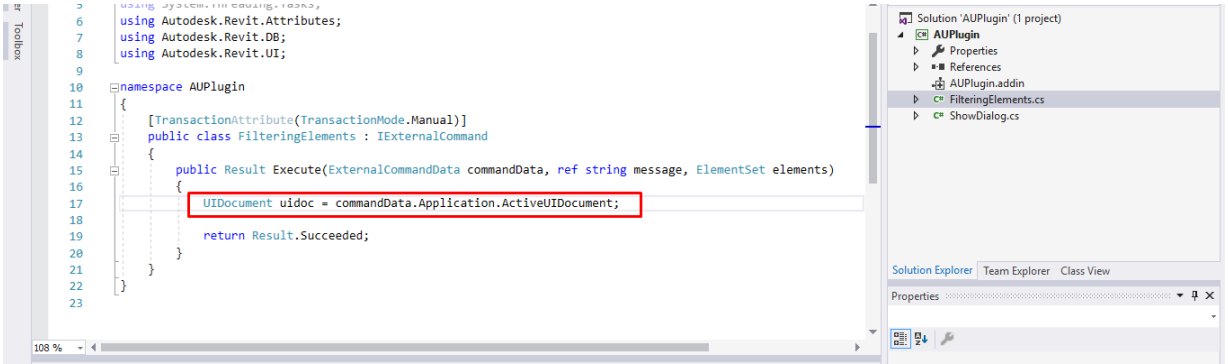


Figure 52 Retrieve UIDocument

Now with the UIDocument object, we can retrieve the Document object by accessing the Document property on the UIDocument. Let's do this with the variable doc.

Figure 53 Retrieve Document

Now with the document object we can access elements in the Revit file with a FilteredElementCollector.

Filtering for Elements

When using the Revit API, elements often need to be retrieved and used in some way. The best way to do this is by using the FilteredElementCollector. A Filtered Element object, when created, is used to search through all elements in a project and these are then filtered by applying filters to the object.

There are 3 different ways to filter a Revit document:

1. filtering the whole document
2. filtering a specific view in a document
3. filtering a specific set of elements in a Revit document.

So let's create a FilteredElementCollector to filter for all the windows in a document.

In the class that we created earlier, start by creating a new FilteredElementCollector with the variable collector. This will require the keyword new as it is a constructor method. And we will be filtering the document, so the parameter will be the document that we have retrieved.

```
10  namespace AUPlugin
11  {
12      [TransactionAttribute(TransactionMode.Manual)]
13      public class FilteringElements : IExternalCommand
14      {
15          public Result Execute(ExternalCommandData commandData, ref string message, ElementSet elements)
16          {
17              UIDocument uidoc = commandData.Application.ActiveUIDocument;
18
19              Document doc = uidoc.Document;
20
21              FilteredElementCollector collector = new FilteredElementCollector(doc);
22
23              return Result.Succeeded;
24          }
25      }
```

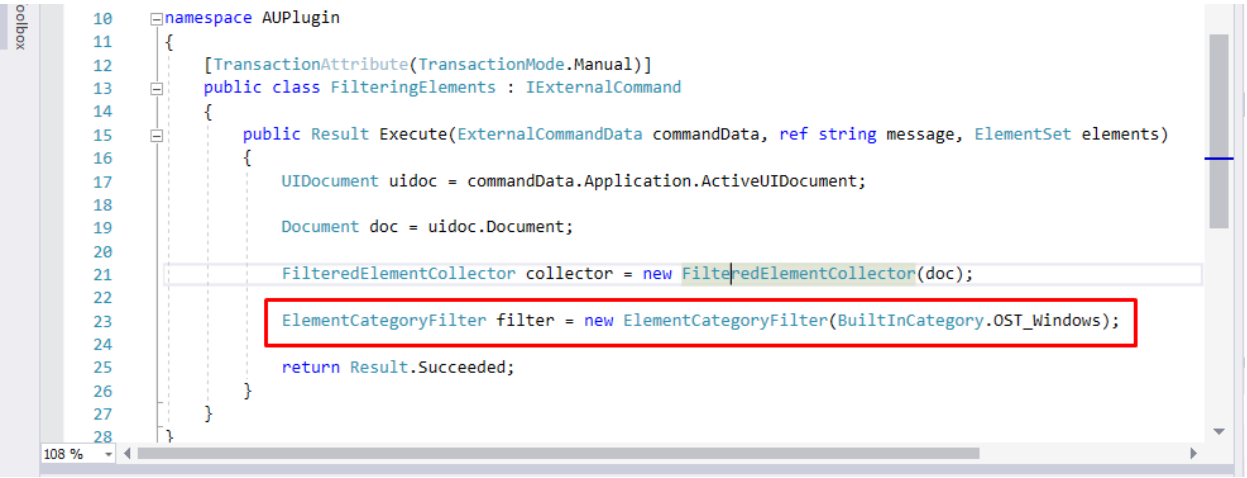
Figure 54 Create FilteredElementCollector

Next, we need to create a filter to apply to the FilteredElementCollector. As an element passes through the collector, the filter is used to check if it meets certain criteria depending on what the filter is, for example, checking if the element is of a certain category or family type. There are 3 Types of filters we can use; quick, slow and logical and when we apply these, we can retrieve the filtered elements.

Just as they sound, quick filters are faster than slow filters so applying these allows the FilteredElementCollector to run faster. Logical filters can be used to combine filters. A list of all filters available can be found here:

http://help.autodesk.com/view/RVT/2019/ENU/?guid=Revit_API_Revit_API_Developers_Guide_Basic_Interaction_with_Revit_Elements_Filtering_Applying_Filters_html

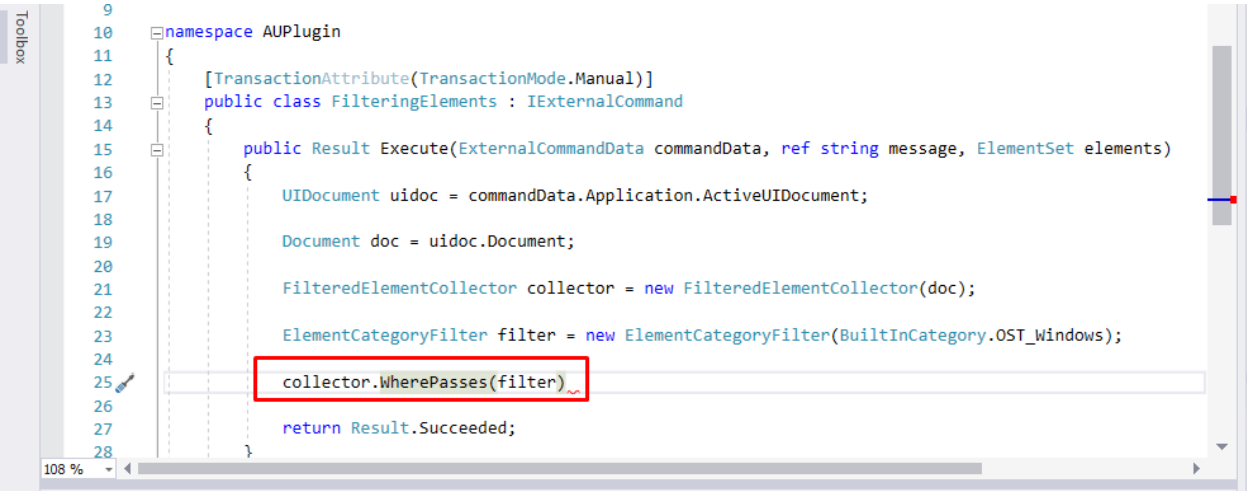
In this exercise, we will be filtering for windows. For this, we can use an ElementCategoryFilter. **To apply the filter, let's start by creating an ElementCategoryFilter variable named *filter* and assign to this a new ElementCategoryFilter. The parameter for this method is a builtincategory. So add in a *builtincategory OST_Windows* enumeration.** This means the filter will filter for any elements that are assigned to the built in Windows category.



```
namespace AUPlugin
{
    [TransactionAttribute(TransactionMode.Manual)]
    public class FilteringElements : IExternalCommand
    {
        public Result Execute(ExternalCommandData commandData, ref string message, ElementSet elements)
        {
            UIDocument uidoc = commandData.Application.ActiveUIDocument;

            Document doc = uidoc.Document;

            FilteredElementCollector collector = new FilteredElementCollector(doc);

            ElementCategoryFilter filter = new ElementCategoryFilter(BuiltInCategory.OST_Windows);

            return Result.Succeeded;
        }
    }
}
```

Figure 55 Create ElementCategoryFilter

After we have created the filter, we need to apply it to the collector. We can do this by accessing the WherePasses method from the collector object. This takes a filter for a parameter which we can use the filter for. On a new line, access this method from the collector we have created, using the filter as the parameter.



```
namespace AUPlugin
{
    [TransactionAttribute(TransactionMode.Manual)]
    public class FilteringElements : IExternalCommand
    {
        public Result Execute(ExternalCommandData commandData, ref string message, ElementSet elements)
        {
            UIDocument uidoc = commandData.Application.ActiveUIDocument;

            Document doc = uidoc.Document;

            FilteredElementCollector collector = new FilteredElementCollector(doc);

            ElementCategoryFilter filter = new ElementCategoryFilter(BuiltInCategory.OST_Windows);

            collector.WherePasses(filter)

            return Result.Succeeded;
        }
    }
}
```

Figure 56 Add Filter to Collector

The FilteredElementCollector object provides shortcut methods to add some quick filters. By applying more quick filters, we can speed up the FilteredElementCollector. Let's apply another quick filter which will only filter for elements that are not element types, that is it will only collect family instances. **To this, after the WherePasses method, access the WhereElementIsNotElementType method which takes no parameter.**

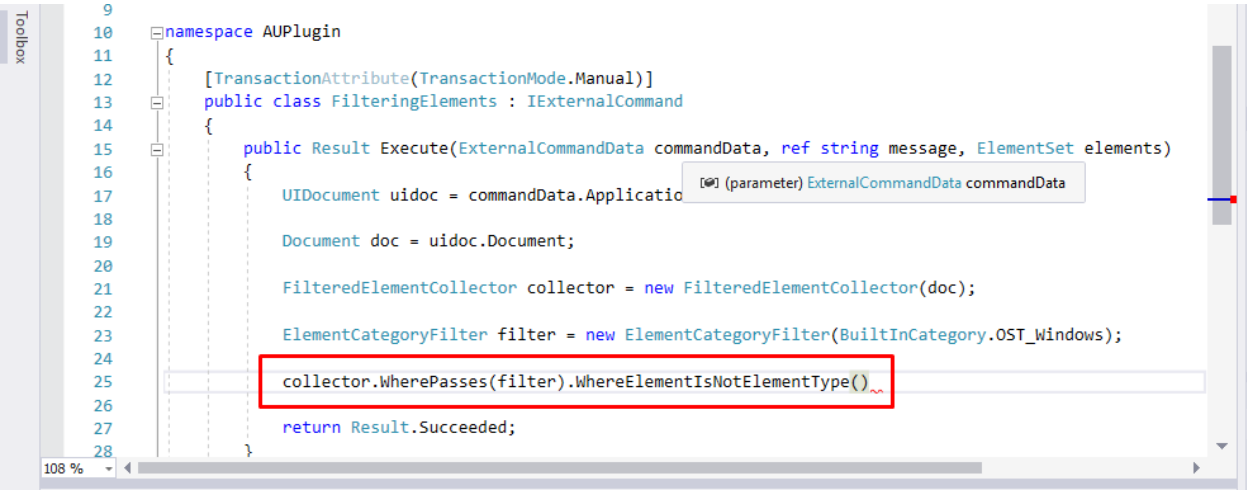


Figure 57 Add Shortcut Filter

We have now filtered the elements but we need to retrieve them from the collector. We can do this by using the ToElements method and ToElementIds method to retrieve them as elements or ElementIds respectively. **Let's retrieve them as Elements by adding the ToElements method to the end of the collector, which takes no parameters.**

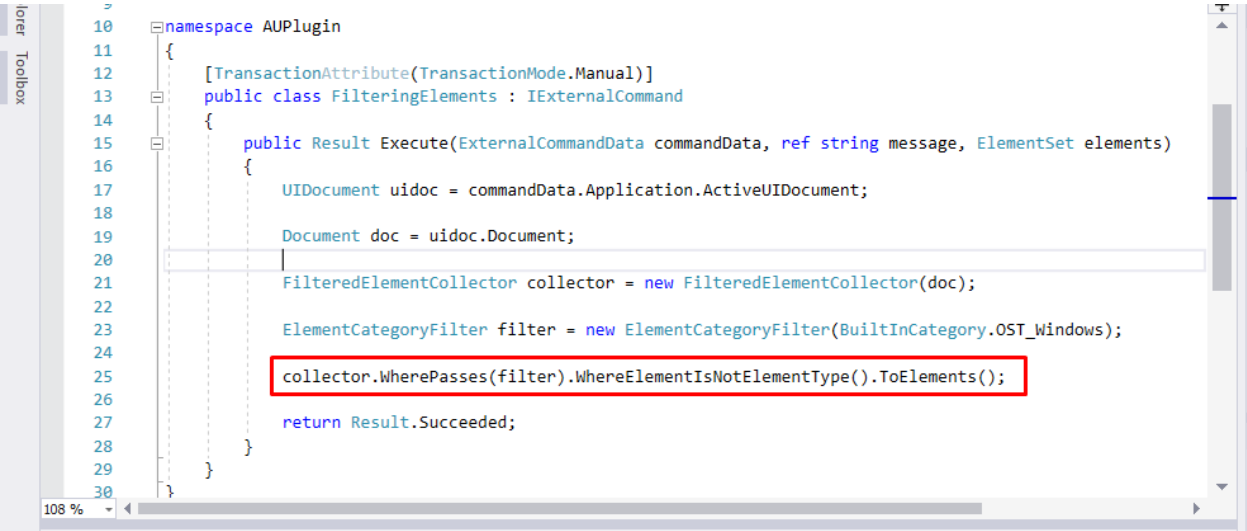


Figure 58 Retrieve Elements from FilteredElementCollector

This will return the elements, but we currently won't be storing the result anywhere, so let's create a variable to do that. The ToElements method returns an IList of elements which we can see when hovering over the ToElements method.

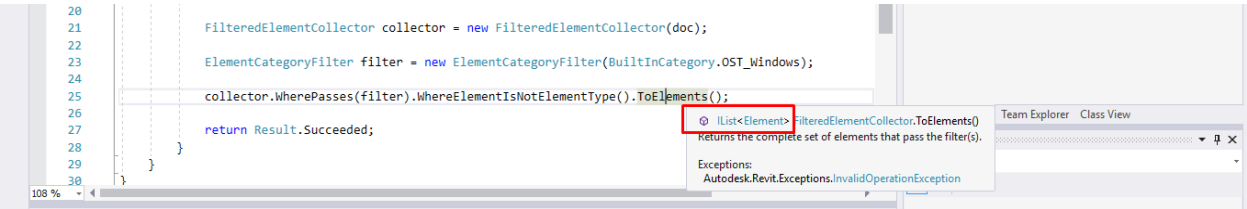


Figure 59 Collector Return Type

Therefore, **let's precede the collector with a new IList variable named windows which we will assign the collected elements to.**

Figure 60 Add IList variable

The Windows variable now stores reference to the list of elements filtered from the document. To show the result back to the user, let's report the number of windows collected. **Create a new TaskDialog with the name *Windows* and for the message, let's add a string displaying the number of windows by accessing the Count property of the IList then adding the string " Windows Found!".**

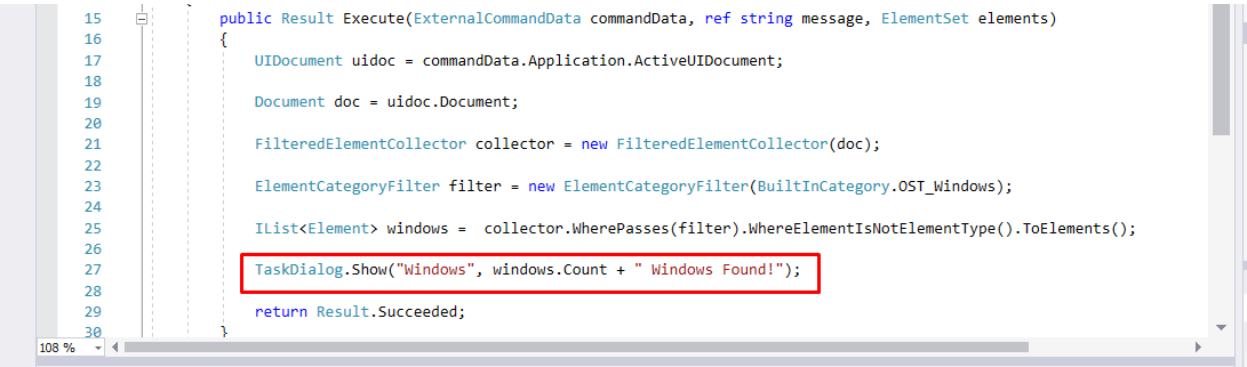


Figure 61 Add Task Dialog

Now that the command is good to go. I have already updated the manifest file for this command but if you started from scratch, the image below shows the contents that I have added in.

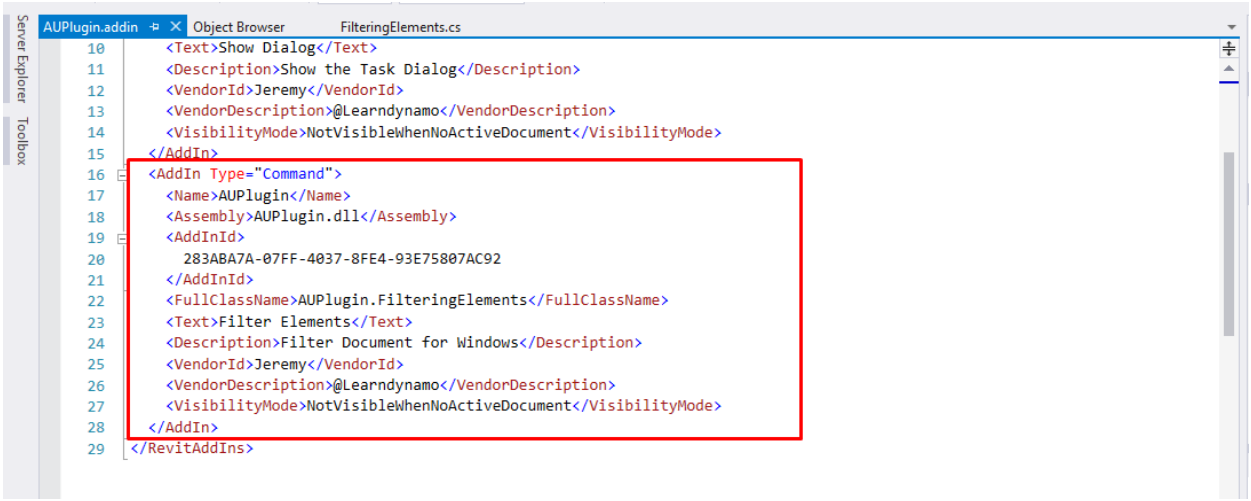


Figure 62 Updated Manifest

The new command is ready to test on the Revit Exercise file named AUFilteringWindows. Go ahead and debug the code and open the Revit project. Once open, running the command will display the number of windows in the project.

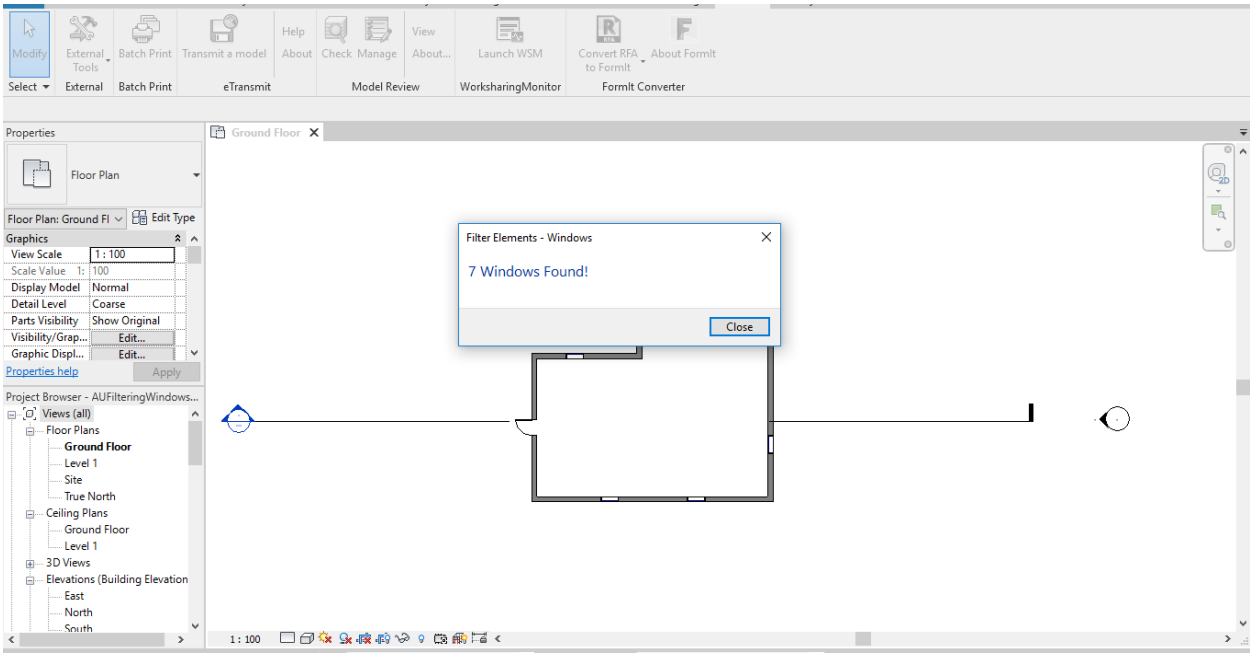


Figure 63 Filtering Document for Windows

Editing Elements

Element information can be retrieved from the model quite easily as it does not change the model. To change the Revit model in any way, we need to work with Transactions in the Revit API. In this section, we will look at how to change an element's parameters by creating and committing a Transaction.

Open the exercise file for exercise 3 - Setting Parameters. In here I have gone ahead and copied the *FilteringWindows* class and named the copy *EditParameters*. I have also added in this new command in to the manifest file.

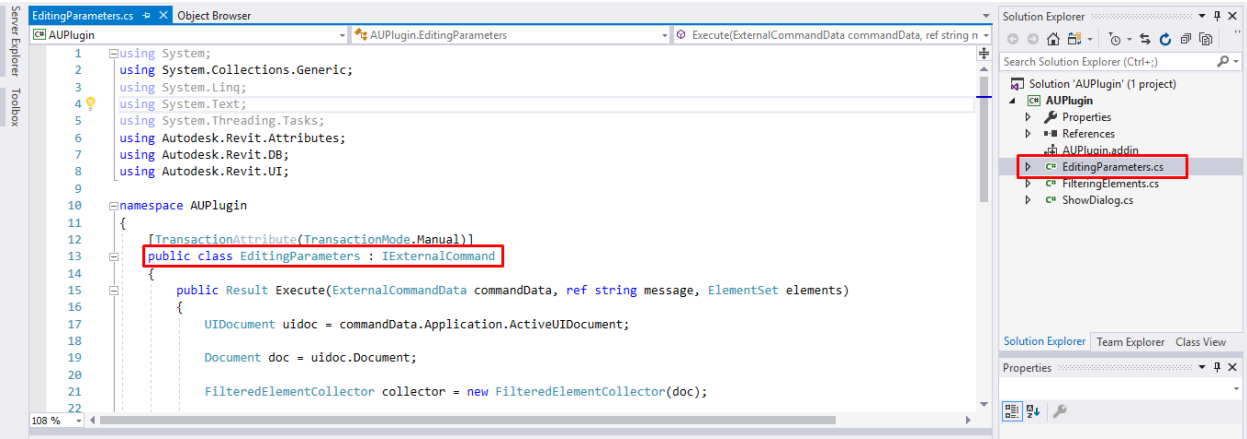


Figure 64 Edit Parameters Class

We will use this new class to edit the windows that we have collected. Before we do however, let's have a look at retrieving a parameter and displaying its value.

Parameters

In this exercise, we will be retrieving the parameters of all the windows we collected and setting new parameter values directly from our plugin.

parameters come in the form of parameter objects that can be retrieved from an element. All objects that we work with in Revit, that implement the element class, can have parameters associated with them. Each parameter object has a definition property that returns a definition object describing the parameter name and type.

So, for instance, say a door had a parameter named head height, the Definition will store it's name, head height, and the type of unit it contains, length in this example.

Each parameter also has a value which is either an integer, double, string, elementID or None, meaning nothing. These values are what we see in the Revit interface.

Parameters can be retrieved in several different ways by accessing properties and methods of an element object. These include:

- **Element.Parameters**: Get All parameters associated with an element as a list.
- **Element.GetOrderedParameters()**: Get all the parameters that we can see associated with an element in the user interface.
- **Element.Parameter**: Get a single parameter from an element by using a builtinParameter. BuiltInParameters are similar to BuiltInCategories in that native parameters have an associated BuiltInParameter.
- **Element.LookupParameter()**: Get a single parameters from an element by using the string name for the parameter.

Getting Parameter Values

Using the LookUp method, let's continue the exercise by getting a parameter from the windows and display its value.

Let's start by looping through each of the windows collected to retrieve the Head Height parameter with a foreach loop, using ele as the variable.

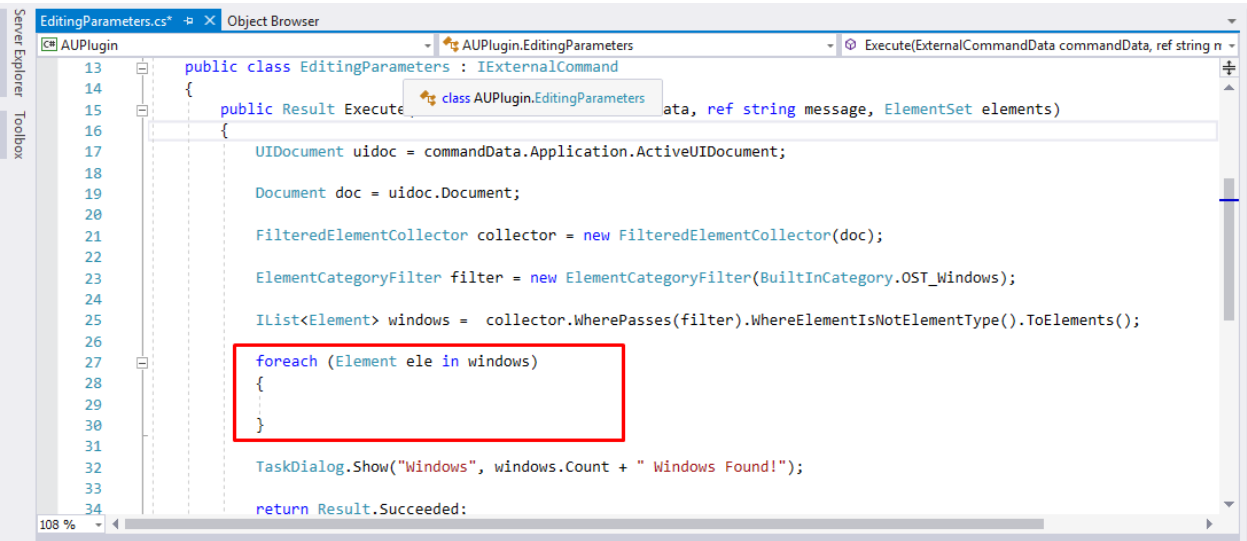


Figure 65 Foreach Loop through Window Elements

Next let's use a Parameter variable named para to retrieve the Head Height parameter from each element by using the LookupParameter method.

```
public class EditingParameters : IExternalCommand
{
    public Result Execute(ExternalCommandData commandData, ref string message, ElementSet elements)
    {
        UIDocument uidoc = commandData.Application.ActiveUIDocument;

        Document doc = uidoc.Document;

        FilteredElementCollector collector = new FilteredElementCollector(doc);

        ElementCategoryFilter filter = new ElementCategoryFilter(BuiltInCategory.OST_Windows);

        IList<Element> windows =  collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();

        foreach (Element ele in windows)
        {
            Parameter para = ele.LookupParameter("Head Height");
        }

        TaskDialog.Show("Windows", windows.Count + " Windows Found!");

        return Result.Succeeded;
```

Figure 66 Lookup Head Height Parameter

This will now retrieve the parameter from the window elements as shown in the interface below.

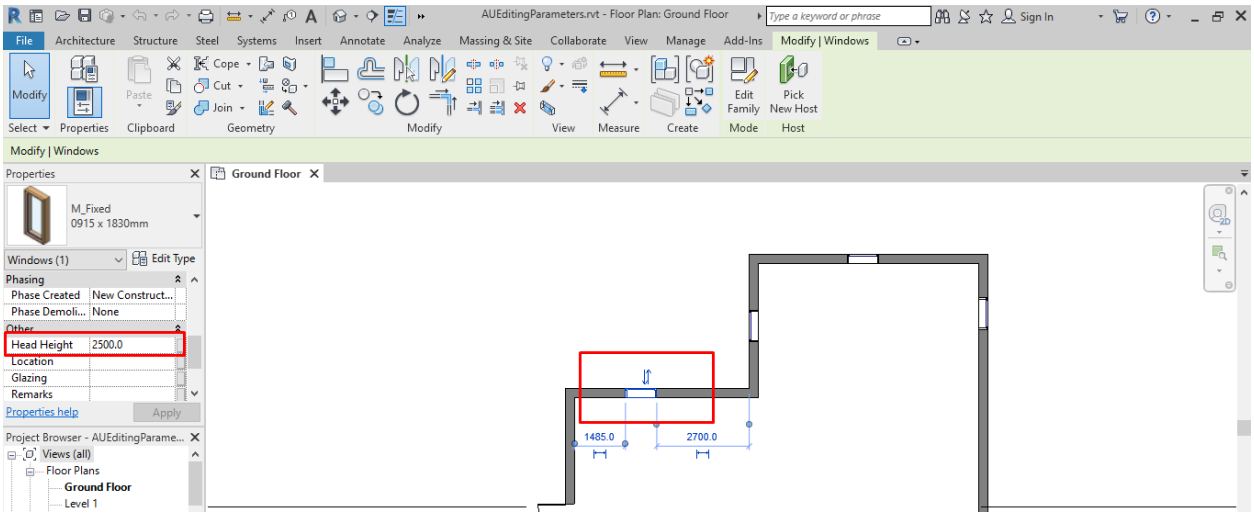


Figure 67 Head Height Parameter

Once we have the parameter from the element, we can access information about the type of value that it is storing and its current value.

To view the type of value that a parameter stores, we can access the storage type property from the parameter object. To do that, create a string variable named storage, and assign to it a call to the StorageType property, followed by ToString method so it can be displayed as a string.

```
17          UIDocument uidoc = commandData.Application.ActiveUIDocument;
18
19          Document doc = uidoc.Document;
20
21          FilteredElementCollector collector = new FilteredElementCollector(doc);
22
23          ElementCategoryFilter filter = new ElementCategoryFilter(BuiltInCategory.OST_Windows);
24
25          IList<Element> windows =  collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();
26
27          foreach (Element ele in windows)
28          {
29              Parameter para = ele.LookupParameter("Head Height");
30
31              string storage = para.StorageType.ToString();
32          }
33
34          TaskDialog.Show("Windows", windows.Count + " Windows Found!");
35
36          return Result.Succeeded;
37      }
```

Figure 68 Get Storage Type

The StorageType is an enumeration that can be one of 5 values shown below.

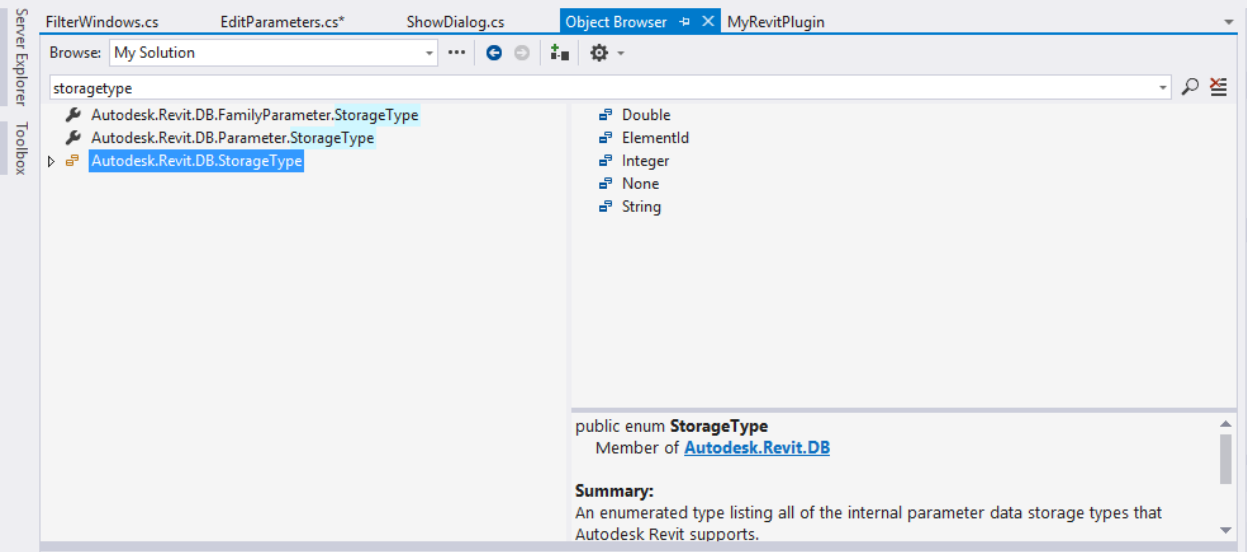


Figure 69 Storage Type Enumeration

Depending on what type of StorageType that a parameter holds, impacts how we can retrieve the parameters value. There are 6 different methods to retrieve the value as shown below, these are preceded by the As word. The correct method to use relates to the StorageType the parameter has. For example, if the StorageType is an integer, the AsInteger() method can be used to retrieve the value, using any other of the methods will cause an error.

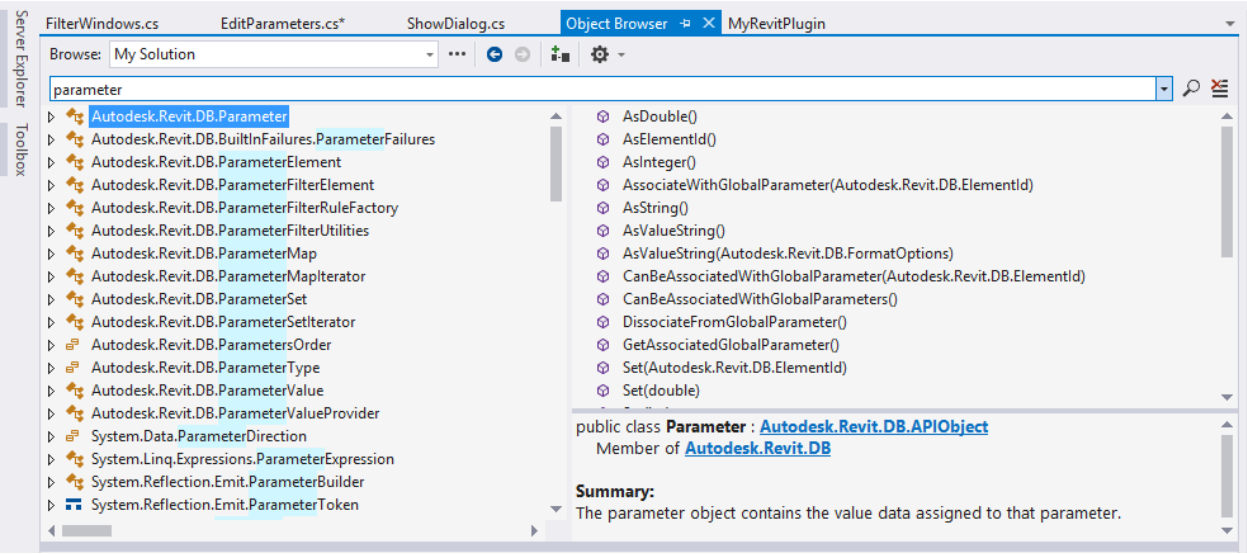


Figure 70 Retrieving Parameter Value Methods

You may notice there are 6 methods rather than the 5 enumeration types, this is because the AsValueString method is overloaded to take formatting options. This can be use to format the string that is retrieved using this method. The AsValueString method is used to retrieve the value as a string display with the units, such as feet or mm.

The Head Height parameter is storage a type of double. **We will soon display this information to check, as we already know the type is a double, let's use the AsDouble method to retrieve the value and store it under a double variable named value.**

```
21          FilteredElementCollector collector = new FilteredElementCollector(doc);
22
23          ElementCategoryFilter filter = new ElementCategoryFilter(BuiltInCategory.OST_Windows);
24
25          IList<Element> windows =  collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();
26
27          foreach (Element ele in windows)
28          {
29              Parameter para = ele.LookupParameter("Head Height");
30
31              string storage = para.StorageType.ToString();
32
33              double value = para.AsDouble();
34          }
35
36          TaskDialog.Show("Windows", windows.Count + " Windows Found!");
37
38          return Result.Succeeded;
39      }
40  }
```

Figure 71 Get Parameter Value

To view the values of each window, let's cut the Task Dialog from the previous exercise, below the for loop, and rename it *parameters.* For its message, let's display both values by adding the strings together *"Parameter is a " + storage + "with value: " + value.ToString());*

```
21          FilteredElementCollector collector = new FilteredElementCollector(doc);
22
23          ElementCategoryFilter filter = new ElementCategoryFilter(BuiltInCategory.OST_Windows);
24
25          IList<Element> windows =  collector.WherePasses(filter).WhereElementIsNotElementType().ToElements();
26
27          foreach (Element ele in windows)
28          {
29              Parameter para = ele.LookupParameter("Head Height");
30
31              string storage = para.StorageType.ToString();
32
33              double value = para.AsDouble();
34
35              TaskDialog.Show("Parameters", "Parameter is a " + storage + "with value: " + value.ToString());
36          }
37
38
39
40          return Result.Succeeded;
```

Figure 72 Show Task Dialog

The command is now set up to collect each window in the project, retrieve the Head Height parameter from each window and it then display its Storage Type and value. I have added this command into the manifest, with the following tags.

```
23      <Text>Filter Elements</Text>
24      <Description>Filter Document for Windows</Description>
25      <VendorId>Jeremy</VendorId>
26      <VendorDescription>@Learndynamo</VendorDescription>
27      <VisibilityMode>NotVisibleWhenNoActiveDocument</VisibilityMode>
28    </AddIn>
29    <AddIn Type="Command">
30      <Name>AUPlugin</Name>
31      <Assembly>AUPlugin.dll</Assembly>
32      <AddInId>
33        C4B022DB-6929-4D69-8179-8ED95FE345DF
34      </AddInId>
35      <FullClassName>AUPlugin.EditingParameters</FullClassName>
36      <Text>Edit Parameters</Text>
37      <Description>Edit Parameter for Windows in the Document</Description>
38      <VendorId>Jeremy</VendorId>
39      <VendorDescription>@Learndynamo</VendorDescription>
40      <VisibilityMode>NotVisibleWhenNoActiveDocument</VisibilityMode>
41    </AddIn>
42  </RevitAddIns>
```

Figure 73 EditParameters Manifest

When debugging the command, make sure the Revit exercise file is open and run the command. If successful, a series of Task Dialogs will appear which will display the parameter

information for each window in the project. As planned, the Storage Type for the Head Height parameter is a double which is how we have been able to display the value.

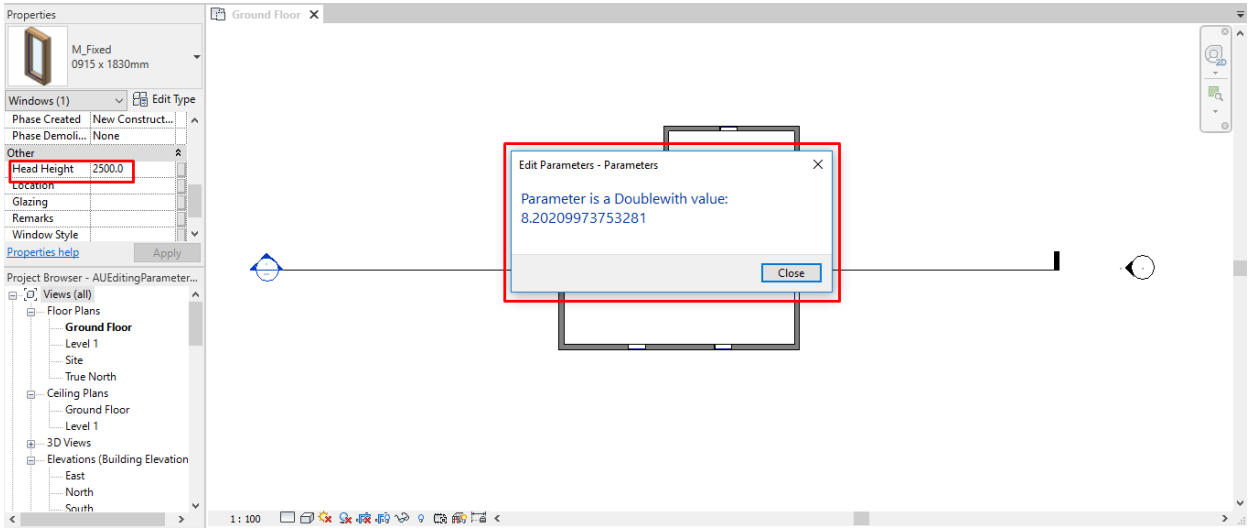


Figure 74 Display Parameter Value

Internal Units

The first thing you may notice about the values displayed in the new command, is the value does not correspond to that of the Head Height parameter as shown below.

This is because Revit's internal units are feet. By accessing the values directly through the API, we are retrieving the internal values before they have been converted to the display units in Revit.

To display the units as that shown in the Revit document, we can convert the units when retrieved. We can do this with the UnitUtils class found in the Revit API. This class provides access to methods that allow for converting units to and from internal units.

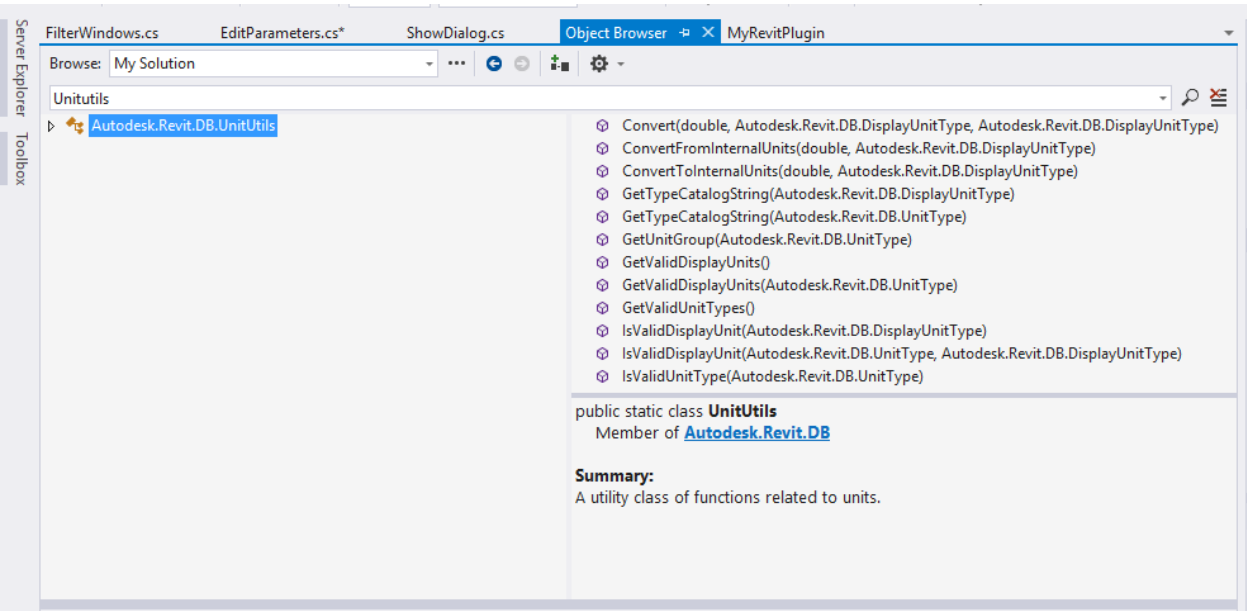


Figure 75 UnitUtils Class

Let's use the ConvertFromInternalUnits to convert the feet value into the project file units which is millimetres. Start by creating a new double variable named *newvalue*. Assign to this a call to the ConvertFromInternalUnits method in the UnitUtils class. For the parameters we need the value to convert, which will be the variable *value* in this case, and for the second parameter we need the units to convert to. This is a type of DisplayUnitType enumeration. As we will be converting to millimetres, select the enumeration value *DUT_Millimeters*. Then update the Task Dialog with the new variable.

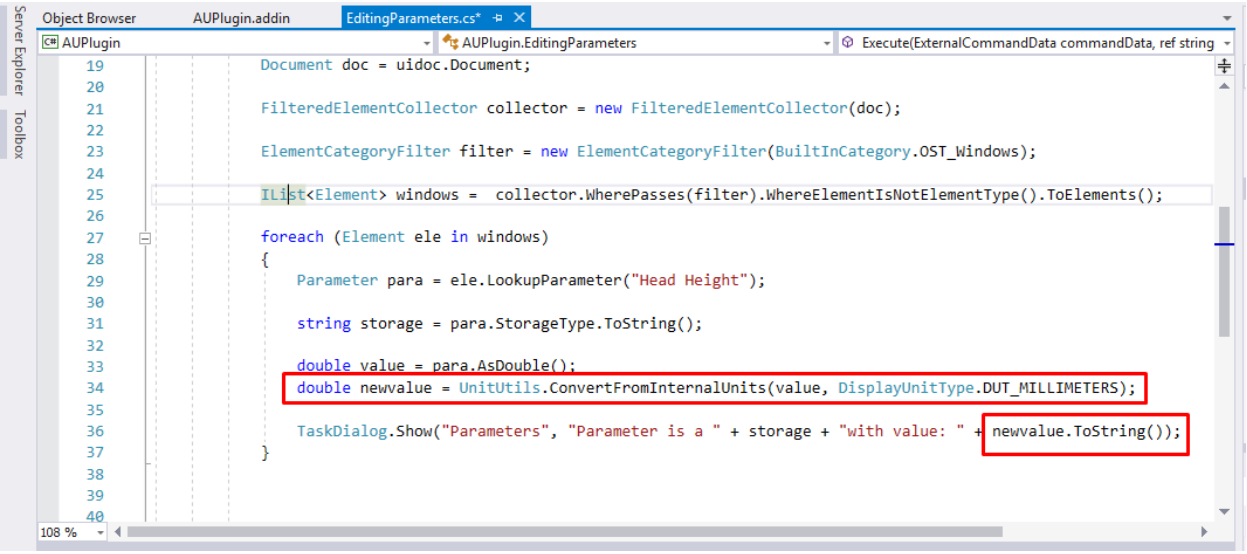


Figure 76 Convert Values from Internal

Now when displaying the parameter value, we will see the value in millimeter units as per the Revit document.

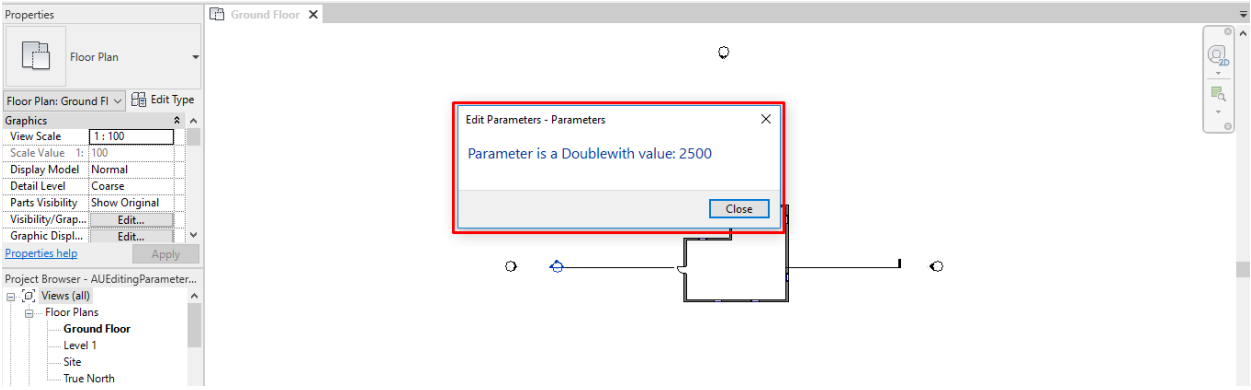


Figure 77 Parameter Values as mm

We now know how to retrieve a parameter and view its value, let's have a look at setting the parameter value by understanding Transactions.

Transactions

Any changes that are made to the revit model need to be encapsulated inside of an active transaction otherwise an exception will be thrown.

The transaction class, once instantiated, contains three different methods that control the transaction, these are start, commit and rollback:

- Transaction.Start(): The Start method is used to start a transaction that has been created. After the transaction is started, changes to the model can be made.
- Transaction.Commit(): The Commit method will commit any changes made to the model after the Start method.
- Transaction.Rollback(): The Rollback method can be used to rollback, or reverse, any changes made in a transaction.

Once an active transaction is committed, the changes made inside the transaction become part of the model. It is important to note that only one transaction can be active at any time and to enclose a transaction within a using statement to ensure the transaction does not unintentionally stay active.

A Using statement will dispose of the object that is passed to the statement as a parameter, after it's scope end. This ensures that the Transaction does not remain open after it is used.

So, let's set up a new Transaction. **Start by creating a Using statement after the Task Dialog is created and create a transaction object for the parameter. For the parameters of the new Transaction, we need to use the Document currently stored under *doc* and then a name for the transaction, in this case let's use *Parameters Updated*. This name will appear in history of Revit operations.**

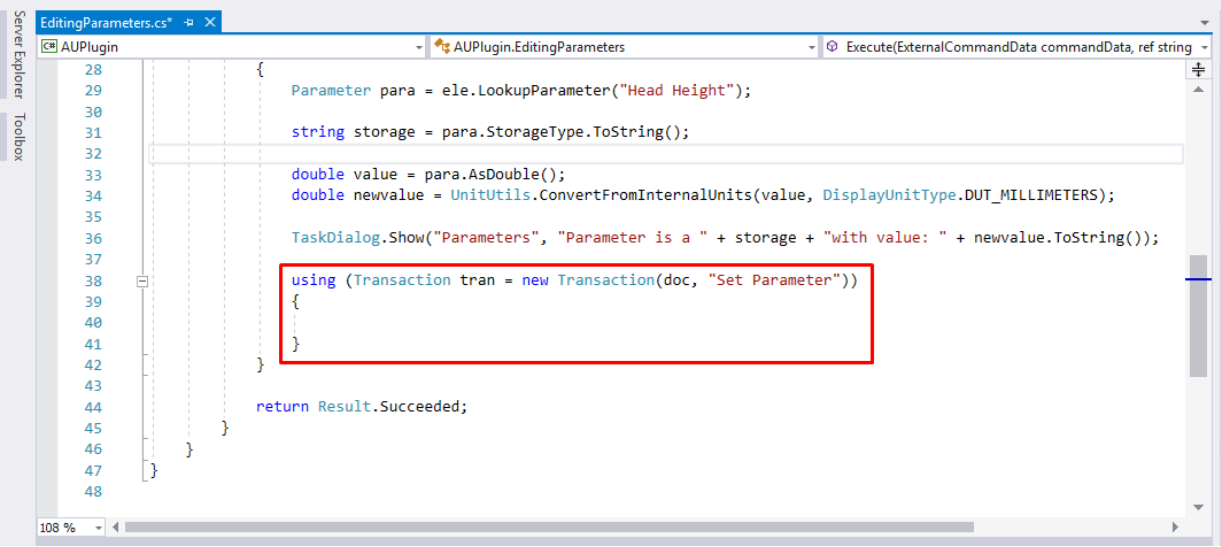


Figure 79 Start Using Statement and Create Transaction Object

Setting Parameter Values

The last step in this exercise is to update the parameter we are retrieving from each window by using the Transaction we have just created. Before we do though, create the value to which we will set the Head Height parameter to.

Before the start of the Using statement, start by creating a variable named *setvalue* which will be a double. Then let's assign to this variable a call to the ConvertToInternalUnits from the UnitUtils class. This will allow us to convert from millimetres, the unit type of the project, from the internal units, feet. So for the parameters, let's use 2100 and *DUT_Millimeters* for the DisplayUnitType.

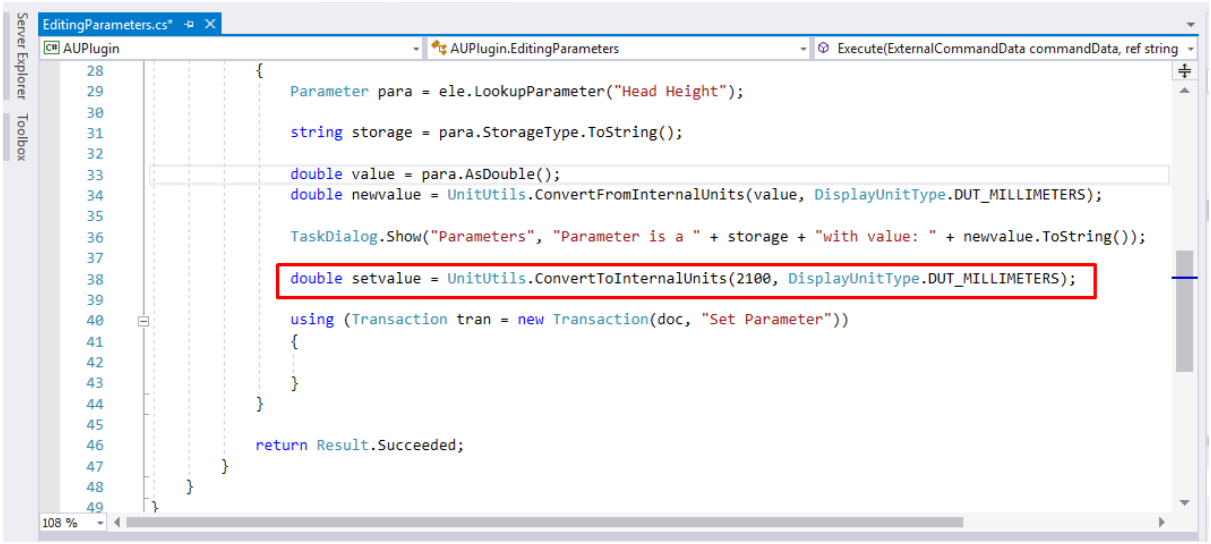


Figure 80 Create Double for Parameter Value

Now we can use the value to set our parameter. **Inside of the Using statement, begin the transaction by calling the start method from the transaction we have created, stored under the *tran* variable.**

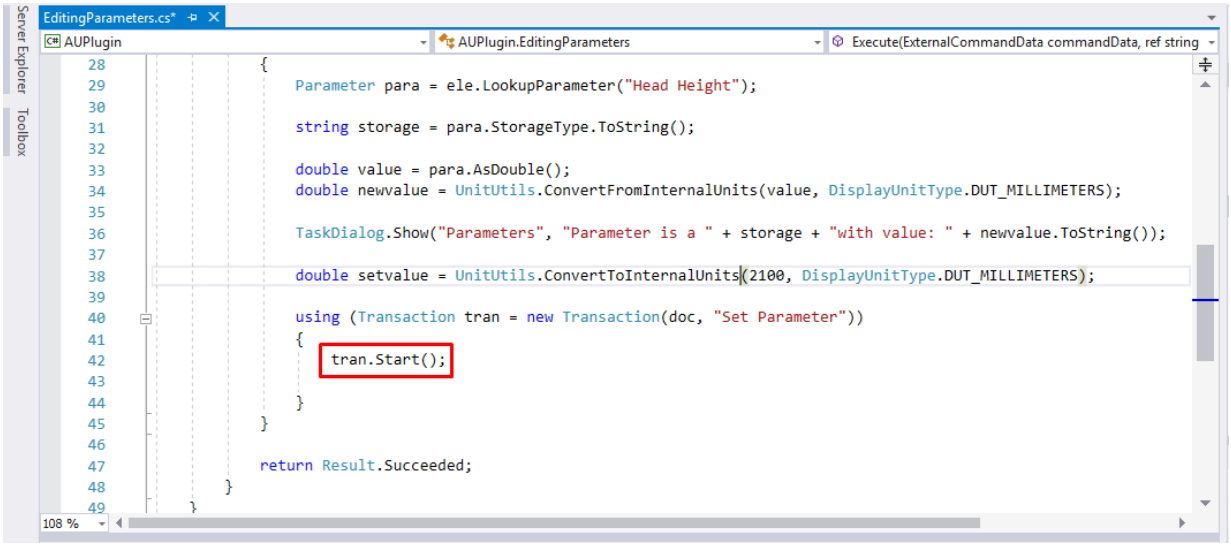


Figure 81 Start Transaction

Now with the transaction started, we can make a change to the model. **To set a new value to the parameter object, we can use the Set method from the object.** This simply takes one parameter which is the new value. Keep in mind here if we try to set the parameter with an value that is different to the parameter StorageType, it will throw an error. **As we know the Head Height StorageType is a double, let's go ahead and set it to our *setvalue* variable.**

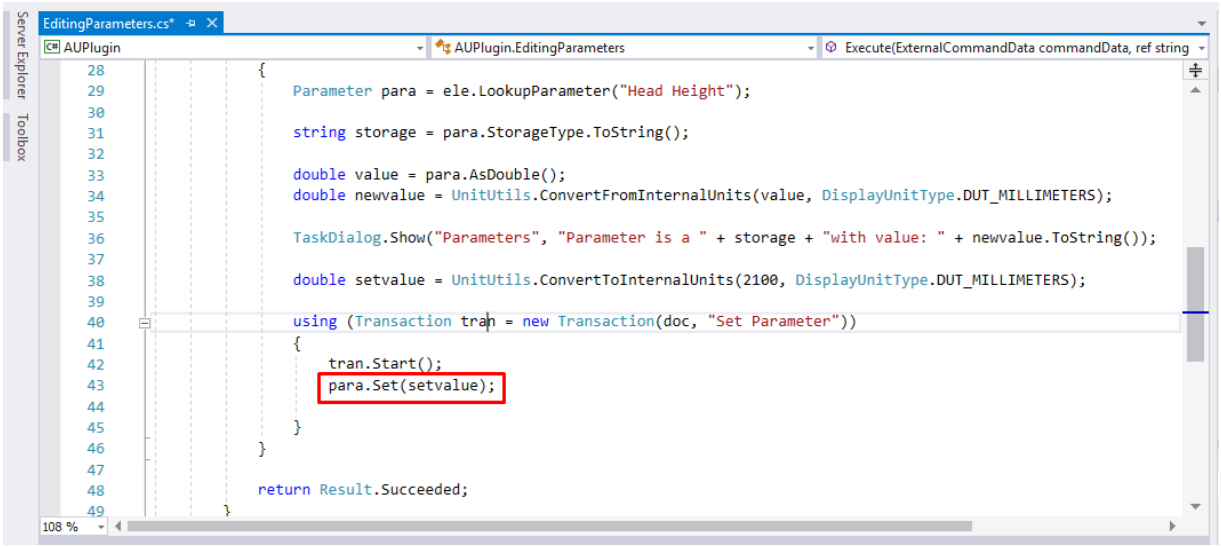


Figure 82 Set Parameter Value

With the new value set, all we need to do now is commit the transaction to the model to make the change. **To do this, make a call to the Commit method in the transaction object.**

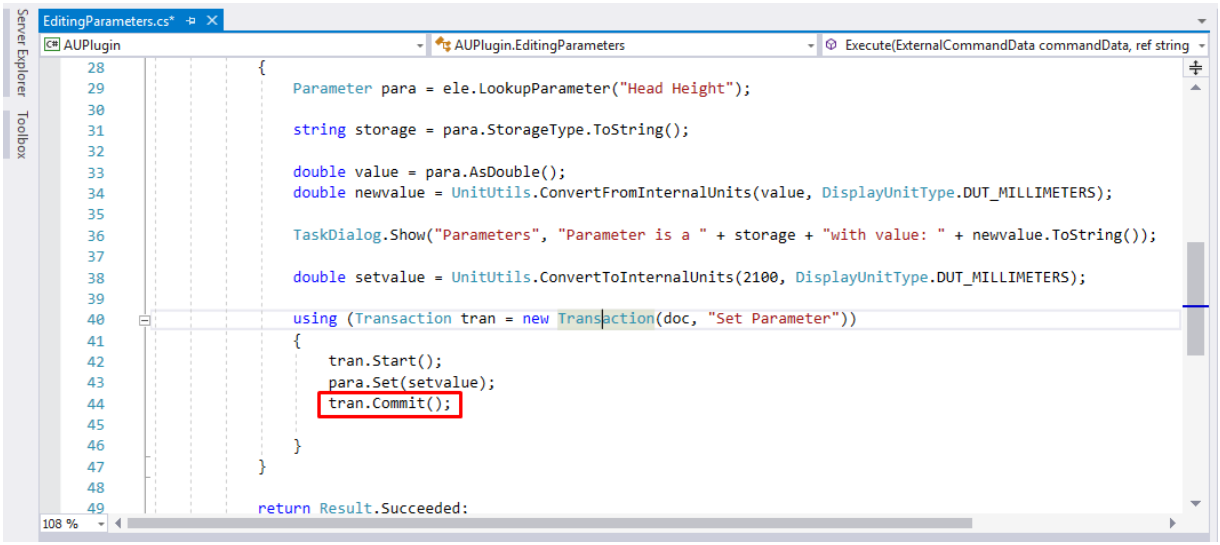


Figure 83 Commit Transaction

Our command will now collect all the windows in the project, retrieve the Head Height parameter from them and set it to a new value. **Before we test the command, let's comment out the task dialog so it doesn't show for each element.**

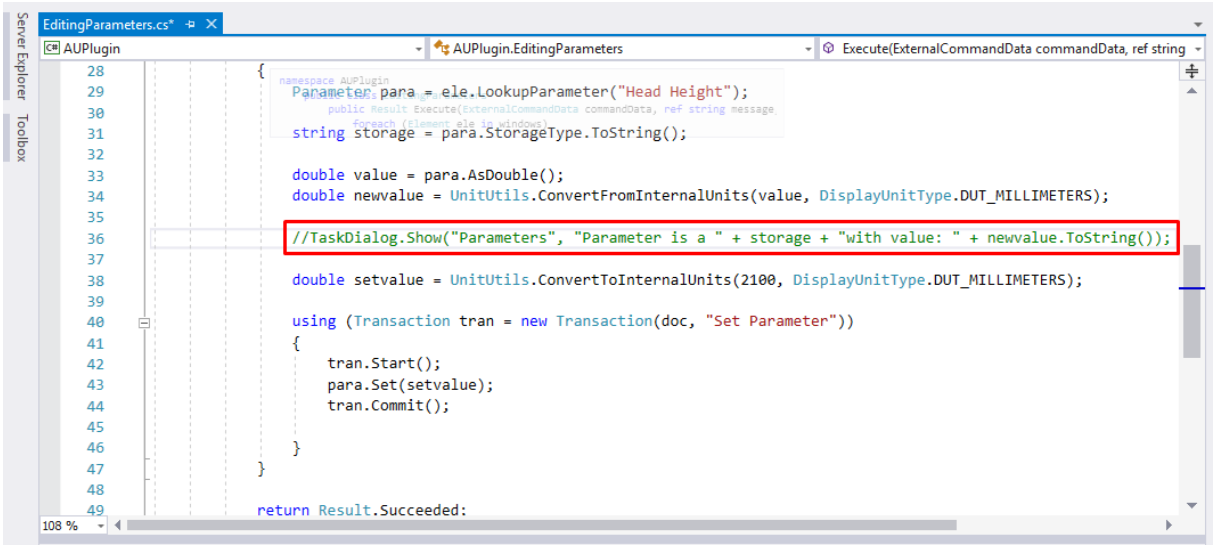


Figure 84 Comment Out Task Dialog

Now we can go ahead and test the command by hitting debug and opening the Revit exercise file for this exercise. Then, when in the project, run the new command named

***Edit Parameters*.** The result will be all the windows in the project updating with the new values as shown below.

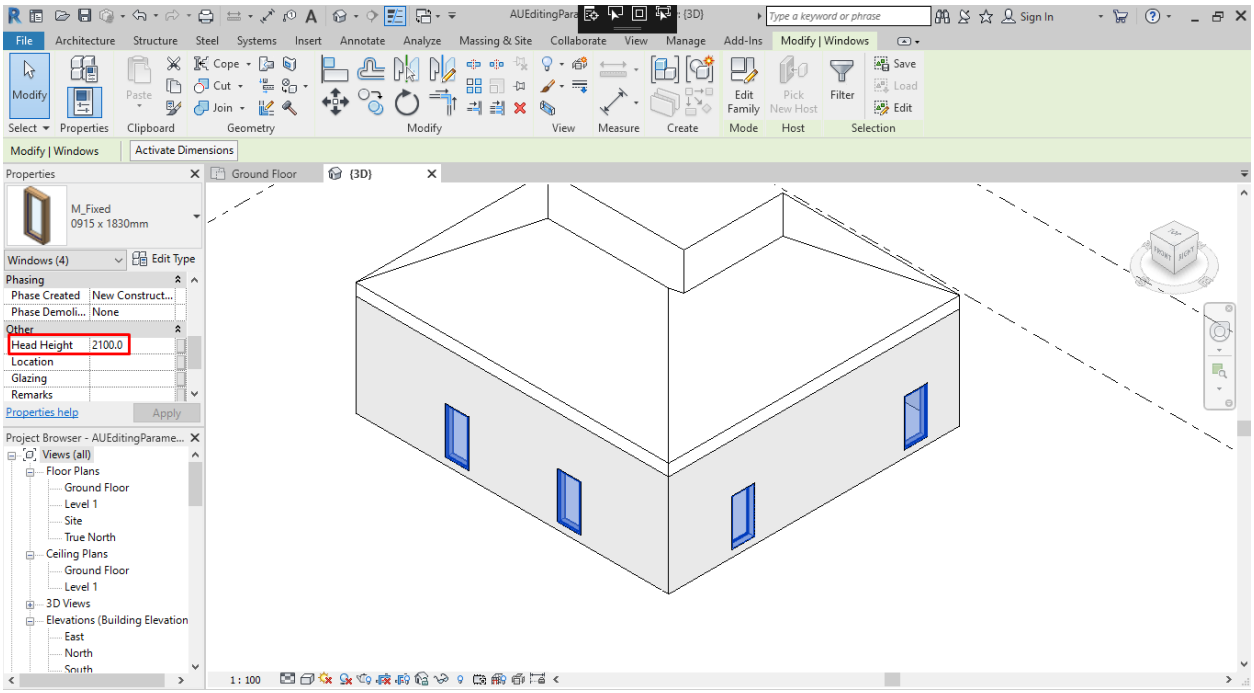


Figure 85 Edit Parameter

We can also see that our Transaction has occurred by looking at the history of commands.

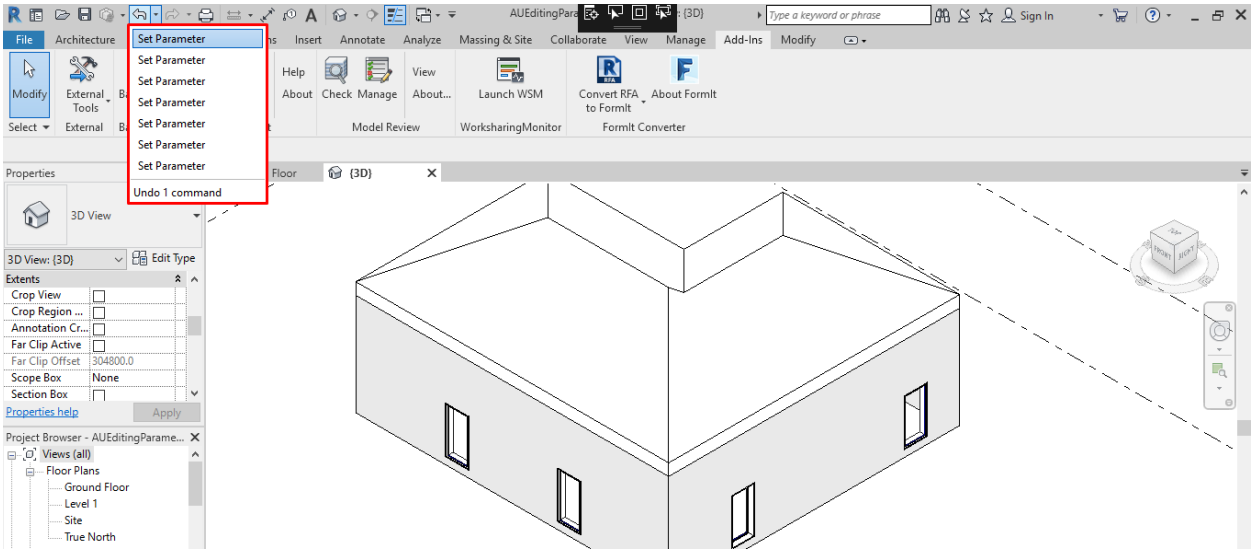


Figure 86 Command History

We have successfully created 3 different commands in our Revit plugin that access different parts of the Revit API. This is only a small part of a huge library that provides many ways of tapping into Revit through custom plugins, so I encourage you to continue learning. If you are not sure where to go from here, have a read through the Resources sections below to continue learning.

Resources

Revit API Developer Guide

https://help.autodesk.com/view/RVT/2019/ENU/?guid=Revit_API_Revit_API_Developers_Guide_html

Jeremy Tammik's Revit API Blog

http://thebuildingcoder.typepad.com/

Harry Mattison's Revit API Blog

https://boostyourbim.wordpress.com/

Danny Bentley's Revit API Youtube Channel

https://www.youtube.com/watch?v=C0mNU2bEUSs&list=PLlyMZ5IcKcci1TvB4qM9S8J-RKp0DhVWO

Revit API Online Search

http://www.revitapidocs.com/

Revit Lookup Installer

https://boostyourbim.wordpress.com/2018/05/17/revit-lookup-2019-installer/