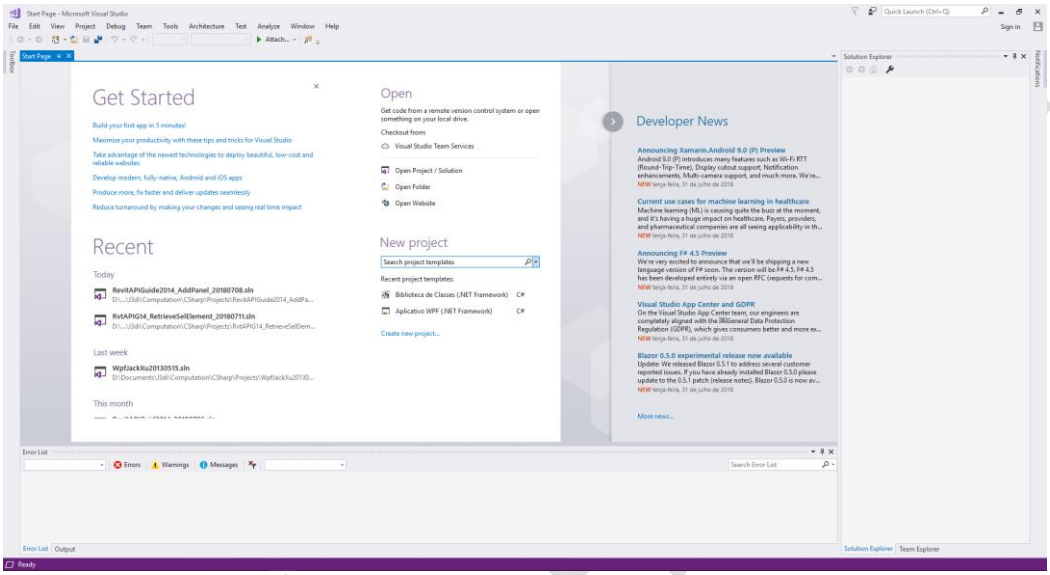
BASICS OF REVIT ADD-IN PROGRAMMING

Visual Studio

Platform for the development of computational services and applications.



Microsoft's Visual Studio platform consists of an integrated development environment (IDE) where software development is possible. It is specific to the .NET Framework, where it is possible to use the Visual Basic (VB), C, C ++, C# (C Sharp) and F# (F Sharp) languages, in addition to other solutions, such as web development, using ASP.NET platform, such as websites, web applications, web services and mobile applications. The languages that are most used in Visual Studio are VB.NET, Visual Basic.Net, and C#, whose pronunciation is C Sharp.

There is a version of Visual Studio that can be installed for individual use, the Community version, which can be downloaded for use associated with a Microsoft account.

WPF - Windows Presentation Foundation

A very important part of an application is its interface. The interface is responsible for capturing user input and, after certain processing, presenting the results properly. Windows replaced its

standard Windows Form interface, which was based on raster images, a dot matrix. The WPF, Windows Presentation Foundation, is based on a vector presentation, resulting in better defined and presented images, being lighter computationally, within other benefits, such as no distortions.

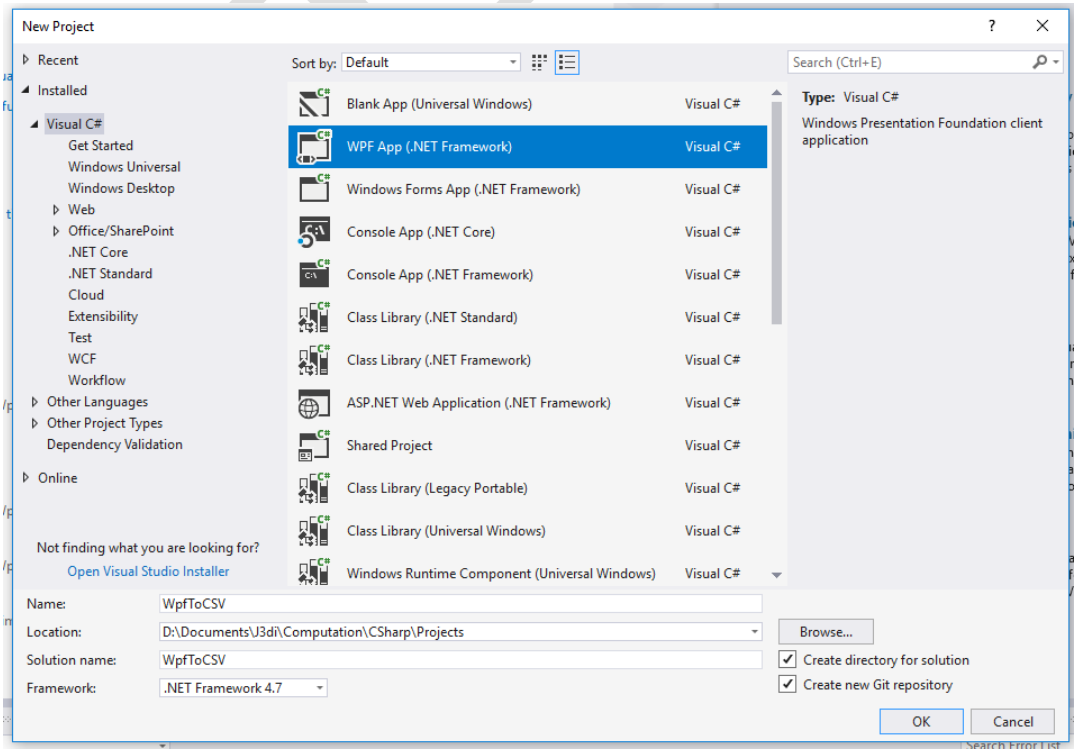
The way to define an interface is through the use of XAML, pronounced zammel in English, which means eXtensible Application Markup Language. XAML is a declarative language based on XML and has native support for editing in Visual Studio. The interface can be defined by dragging the components to the standard window, or by textual editing of XAML.

Example - WPF Project to Generate CSV

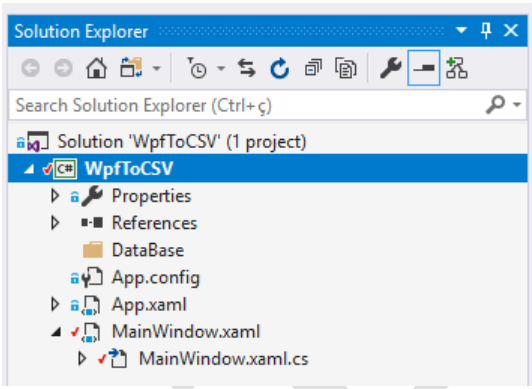
Through this example, we will better understand the use of this technology. We will open Visual Studio and start a new project.

- Menu File – New - Project

The design options are presented. We must choose WPF App and choose the name “WpfToCSV”.



In Solution Explorer we can see the files created, where we can also add folders for better organization. The “MainWindow.xaml” file contains the application interface. The “MainWindow.cs” file is the program's logic file, which in this case will be written in C #. Double-clicking on these files allows open to editing.



The focus of this work is not WPF, therefore, it presents the complete code without going into details about its syntax. It is interesting to start with the interface and then promote the logic, using the interface components. Of course, this is indicated for coding when the project is ready, something that is usually done with UML.

XAML code:

```
<Window x:Class="WpfToCSV.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfToCSV"
        mc:Ignorable="d"
        Title="Creating CSV" Height="170" Width="330">
    <StackPanel>
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="Auto"/>
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto"/>
```

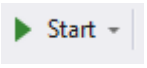
```
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
            </Grid.RowDefinitions>
            <TextBlock Text="Nível" FontWeight="Bold" Margin="5" Grid.Column="1"
Grid.Row="0"/>
            <TextBlock Text="Material" FontWeight="Bold" Margin="5" Grid.Column="2"
Grid.Row="0"/>
            <!---->
        <TextBlock Text="Janela:" FontWeight="Bold" Margin="5" Grid.Column="0"
Grid.Row="1"/>
        <TextBlock Text="Porta:" FontWeight="Bold" Margin="5" Grid.Column="0"
Grid.Row="2"/>
        <!---->
            <TextBox Name="txtJanelaNivel" Width="100" Margin="5" Grid.Column="1"
Grid.Row="1"/>
            <TextBox Name="txtJanelaMaterial" Width="100" Margin="5" Grid.Column="2"
Grid.Row="1"/>
            <!---->
        <TextBox Name="txtPortaNivel" Width="100" Margin="5" Grid.Column="1"
Grid.Row="2"/>
        <TextBox Name="txtPortaMaterial" Width="100" Margin="5" Grid.Column="2"
Grid.Row="2"/>
    </Grid>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
            <Button Name="btnLimpar" Width="51" Margin="5"
Click="BtnLimpar_Click">_Limpar</Button>
            <Button Name="btnCSV" Width="51" Margin="5"
Click="BtnCSV_Click">_CSV</Button>
            <Button Name="btnExit" Width="51" Margin="5,5,37,5"
Click="BtnExit_Click">Sai_r</Button>
        </StackPanel>
    </StackPanel>
</Window>
```

C# code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.IO;
using Microsoft.Win32;

namespace WpfToCSV {
    /// <summary>
    /// Interaction logic for MainWindow.xaml
```

```
    /// </summary>
    public partial class MainWindow : Window {

        // Global variables:
        string arquivoNome;
        public string ArquivoName { get { return arquivoNome;} set { arquivoNome =
value; } }

        public MainWindow() {
            InitializeComponent();
            ArquivoName = "";
        }

        private void BtnExit_Click(object sender, RoutedEventArgs e) {
            this.Close();
        }

        private void BtnLimpar_Click(object sender, RoutedEventArgs e) {
            txtJanelaMaterial.Text = "";
            txtJanelaNivel.Text = "";
            txtPortaMaterial.Text = "";
            txtPortaNivel.Text = "";
            // Tip: to get value as number var = Convert.ToDouble(txtField.Text)
        }

        private void BtnCSV_Click(object sender, RoutedEventArgs e) {
            try {
                string CombinedPath =
System.IO.Path.Combine(Directory.GetCurrentDirectory(), "..\\..\\DataBase");
                SaveFileDialog saveFileDialog = new SaveFileDialog {
                    Title = "Save CSV",

                    InitialDirectory = System.IO.Path.GetFullPath(CombinedPath),
                    Filter = "Rtf documents|*.rtf|Txt files (*.txt)|*.txt|Csv files
(*.csv)|*.csv|All files (*.*)|*.*",
                    FilterIndex = 3,
                    RestoreDirectory = true
                };
                saveFileDialog.ShowDialog();
                if (saveFileDialog.FileName == "") {
                    MessageBox.Show("Invalid File ", "Save as ", MessageBoxButton.OK);
                } else {
                    ArquivoName = saveFileDialog.FileName;
                    using (StreamWriter writer = new StreamWriter(ArquivoName)) {
                        writer.WriteLine("Element;Nivel;Material");
                        writer.WriteLine("Door:;{0};{1}", txtPortaNivel.Text,
txtPortaMaterial.Text);
                        writer.WriteLine("Window:;{0};{1}", txtJanelaNivel.Text,
txtJanelaMaterial.Text);
                    }
                        MessageBox.Show("File Saved Successfully!", "Save!",
MessageBoxButton.OK);
                    }

                } catch (Exception ex) {
                MessageBox.Show(ex.ToString());
            }
        }
    }
}
```

The program must be compiled so that an executable is generated, in case no serious error is found. The button for compilation, at the top under the menu, is one of the ways to compile the file



. We must observe if we did not get an error in the status bar

.

If everything goes without error, the window created as an interface is displayed and the program can be tested.

Revit API

Development of plug-ins with Revit's API, Application Programming Interface, using the C # language, from .Net, in Visual Studio.

We will use, as a resource, “Revit 2019” and “Visual Studio 2017” to develop our examples. These two platforms can be downloaded in student version, for educational use.

Visual Studio must be optimized to work with C #, computational language, and use English as a language, to better follow the examples.

Not all defined steps are indispensable for the functioning of all examples, but their definition is important for good programming practice or even indispensable for specific examples.

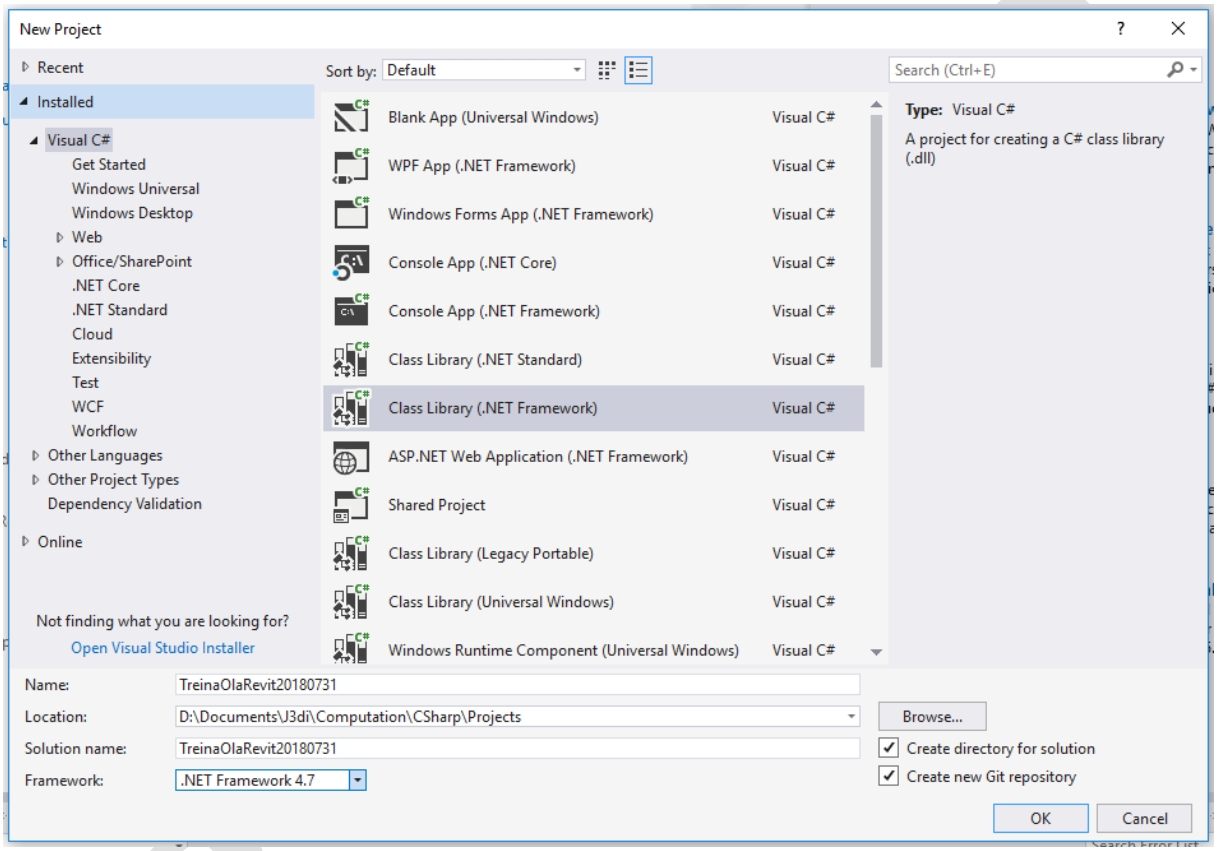
We will use practical examples to understand the concepts needed to use the Revit API, with the C # language, on the Visual Studio platform.

You can find specific help in the SDK, software development kit, of the API, available with the installation of the Revit package, the SDK installer, which is only available in the installation files, found in “C: \ Autodesk \ Revit_2019_G1_Win_64bit_dlm \ Utilities \ SDK \ RevitSDK.exe ”. The SDK provides help, as well as examples and support files for development using the Revit API.

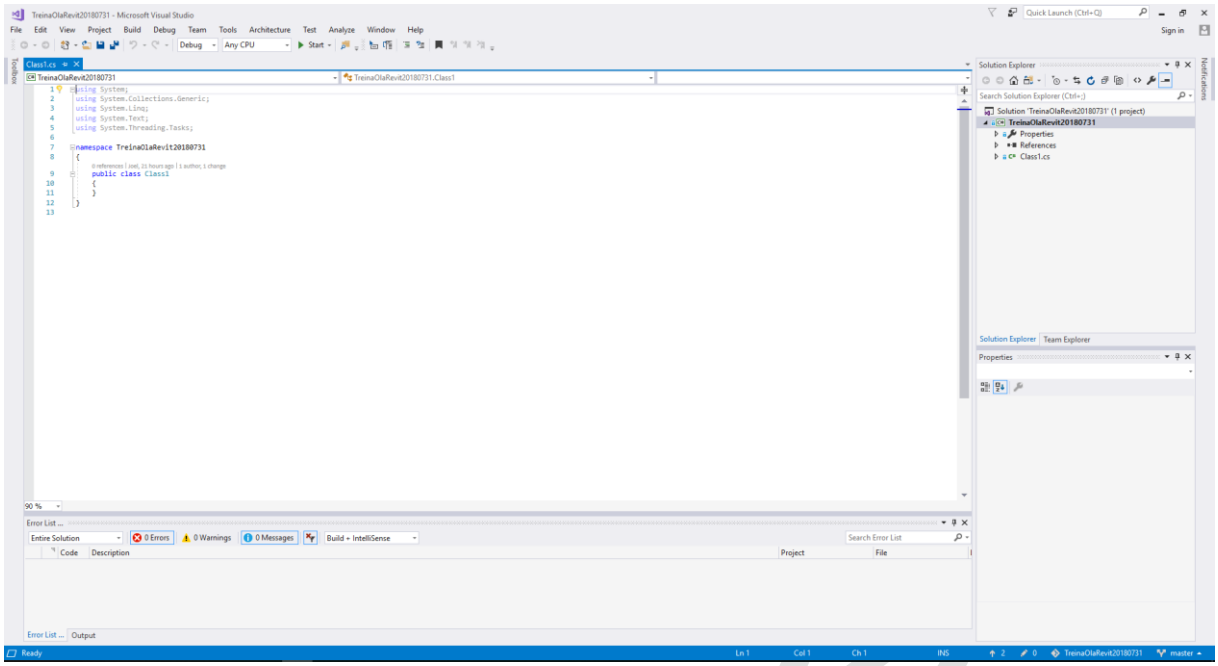
Example - "Hello Revit!" Project

There are several types of projects that can be created with Visual Studio. We will use Class Library to create a plug-in for Revit.

- Menu File - New - Project



A window for project definitions makes it possible to choose the standards to be used. We will choose "Visual C #" for language, "Class Library" to generate a dll, dynamic link library. The project's name will be “TreinaOlaRevit20180731”. We return to the main window, and the created project is displayed.

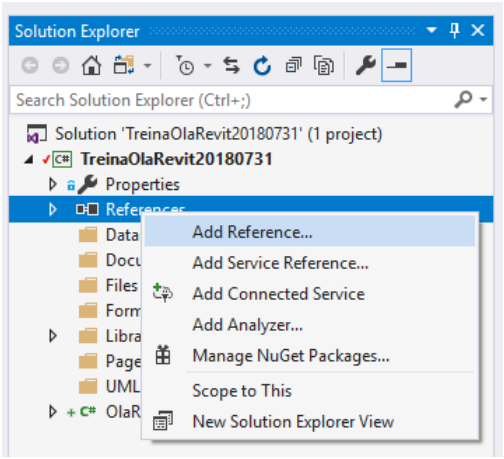


The “Class1.cs” file contains our C # code. We will rename this file to “OlaRevit.cs”. To do this, we must right-click on the file and choose the option "Rename". Visual Studio should ask if we want to update references to this name, and we should choose "Yes".

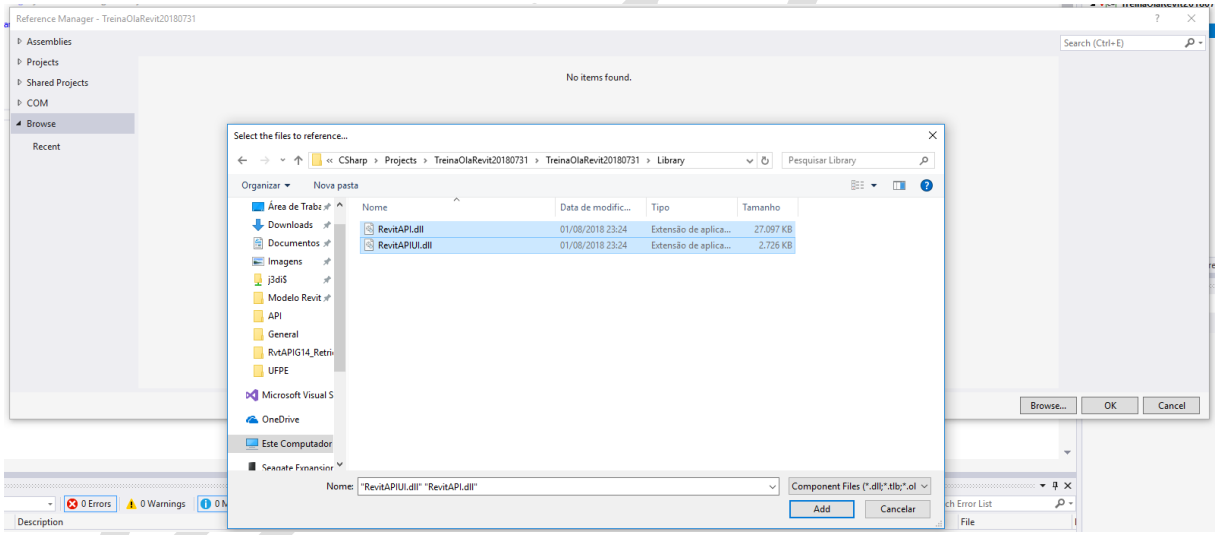
The wheel must not be reinvented in programming. We will make use of code libraries, which contain the necessary code for reuse and assembly of our logic. The Revit API is contained in a dll, dynamic link library, which are libraries with the description of the code needed to interact with Revit. We need to add a reference to the libraries that we will use.

First, we will add the two dlls needed to use the Revit API, "RevitAPI.dll" and "RevitAPIUI.dll". They are in the directory where our Revit is installed, possibly “C: \ Program Files \ Autodesk \ Revit 2019”.

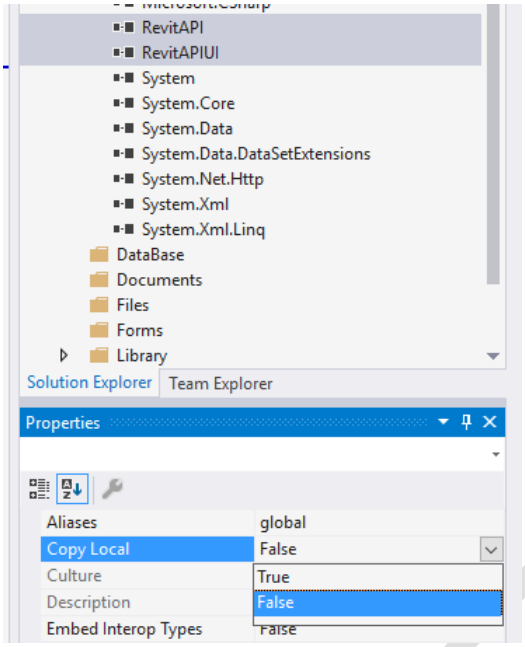
We right-click on References from the Solution Explorer window. We select “Add Reference ...”



Use the "Browser" option to browse to the location where the dlls are and add. A good option may be to copy them to the project's “Library” folder, because changes to the Revit installation can modify or lose the original location of these API libraries.



Change the “Copy Local” parameter in Solution Explorer to false. This will prevent a copy of the library from being copied to the folder whenever the program is copied.



Now we need to make some modifications to our C # code, file "* OlaRevit.cs", so that it can meet the requirements of the API.

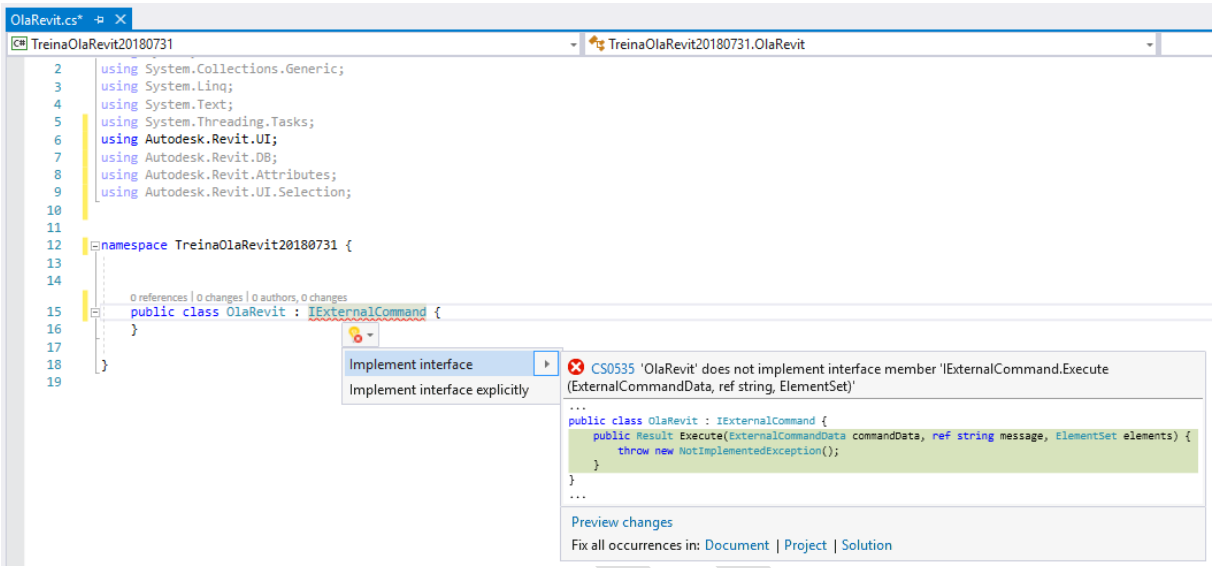
We added the libraries that the code will use:

```
using Autodesk.Revit.ApplicationServices;

using Autodesk.Revit.UI;

using Autodesk.Revit.DB;

using Autodesk.Revit.Attributes;

using Autodesk.Revit.UI.Selection;
```

We added the directive that instructs about the transaction. Note that for updates, the Automatic option is no longer supported:

```
[Transaction(TransactionMode.Manual)]
```

We need to implement the IExternalCommand interface, in which case we will make use of a very interesting feature of Visual Studio. Add the code ": IExternalCommand" right after our class definition. Visual Studio will allow us to define the implementation of this interface, just using the option that will appear under the definition, a lamp icon with the arrow that we will select, and that appears when we leave the selection on the newly added code, as shown in the image follow.



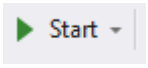
A method, "Execute", will be added, and, once it returns "Result", we need to output this to our code. We can comment on the code that throws the exception of not implementing the code with "//". We will use the output within a "try ... catch" structure, to handle exceptions.

This code of ours will only display a message in Revit using a normal text window, which can be done with the code "TaskDialog.Show (" Revit "," Hello Revit! ");".

Our complete code is:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;
```

```
namespace TreinaOlaRevit20180731 { // NameSpace.

    [Transaction(TransactionMode.Manual)]
    public class OlaRevit : IExternalCommand { // Class that will rise the dll for use
in Revit.
        public Result Execute(ExternalCommandData commandData, ref string message,
ElementSet elements) { // Input method that implements the required API interface.
            try {
                TaskDialog.Show("Revit", " Hello Revit!"); //
Dialog box that will run if all goes well.
                return Result.Succeeded; // Return that everything went well.
            } catch (Exception ex) { // Exception handling.
                message = ex.Message;
                TaskDialog.Show("Error!",message);
                return Result.Failed; // Error return.
                // throw; // Code to be commented or removed.
            }
            //throw new NotImplementedException(); // Code to be commented or removed.
        }
    }

}
```

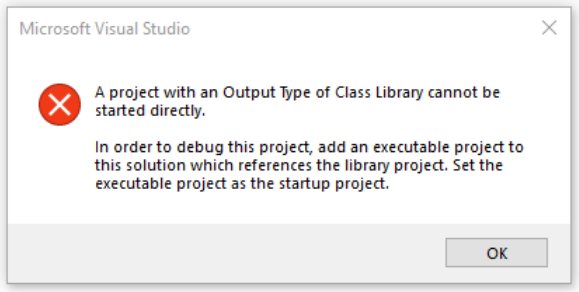
We must then compile the file 

.

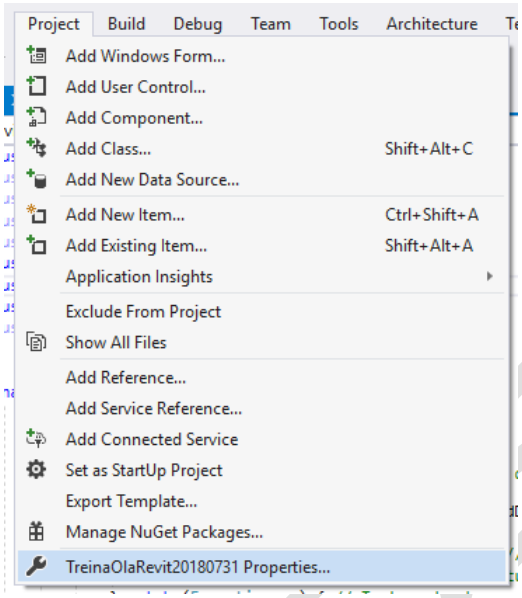
And see if we didn't get an error in the status bar 

. A warning screen may appear, since we are not creating a program, and we have not assigned it to an application, which can be configured to use Revit, but we will not do it here yet.

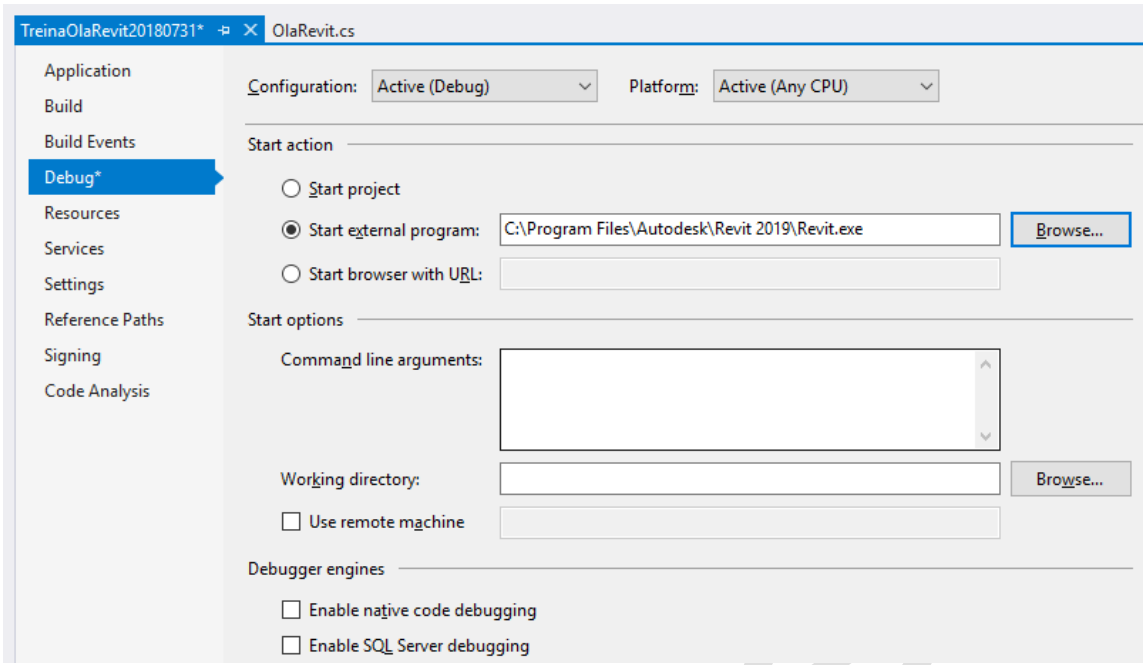
When we execute the project in this way, we will have the warning that the output cannot be executed directly, since it is a dll, and that we would need an external program for that. To immediately test the library, we can then configure Revit as the program to test the output.



The exit program must be configured through the menu "Project", submenu "Properties ...", which will take us to the configuration screen.



We must then the “Debug” tab, and then choose “Start external program:”. The "Browser" button allows us to select the program that will be executed when we compile the project, in this case, Revit. Navigate to the folder where Revit was installed and select the "Revit.exe" executable. A likely path will be “C: \ Program Files \ Autodesk \ Revit 2019”.



Whenever we compile the project, Revit will be started, in case we don't have any errors to debug. The library, the dll file, will be created in the "\ bin \ Debug \" folder of our project. This is the dll that consists of our plug-in and that will be pointed out so that Revit can make use of our code, something that is done with a manifest.

Preparation of Manifesto that Enables the Plugin

When Revit starts, it will check if there is a file, which we call a manifest, in your default plug-in folder. If there is a file in this folder, Revit will interpret it. This is the way to allow the use of plug-ins in Revit. If everything goes well with the interpretation, we will be notified of its loading, and we will have to choose between not loading the dll, because this is a possibility of entry for viruses, loading only once, or always loading. When we choose to always load, Revit will not give us warnings for loading these plug-ins on the next starts. An example of a manifest is in the download area of eTlipse, website https://www.etlipse.com/, which refers to the use of AddInManager, an application that makes it possible to directly load plug-ins from the formed .dll.

We will create the file "TreinaOlaRevit20180731.addin", which will contain the necessary information for interpretation when starting Revit. This file will need to be in the Revit plug-ins folder, in general, "C: \ Users \ [XXXX] \ AppData \ Roaming \ Autodesk \ Revit \ Addins \ 2020 \", where

"[XXXX]" is the name of the user. The AppData folder can be found most easily via the "% AppData%" shortcut.

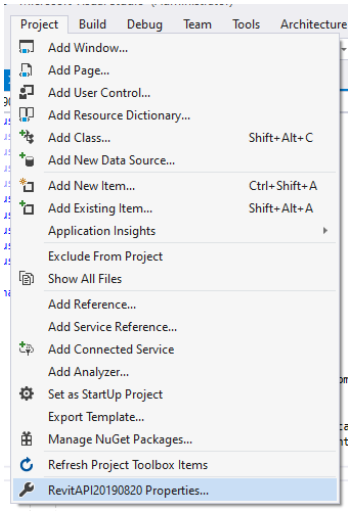
Our file will be as shown below. Parentheses are used here only for comment, but obviously they should not be contained in our manifest.

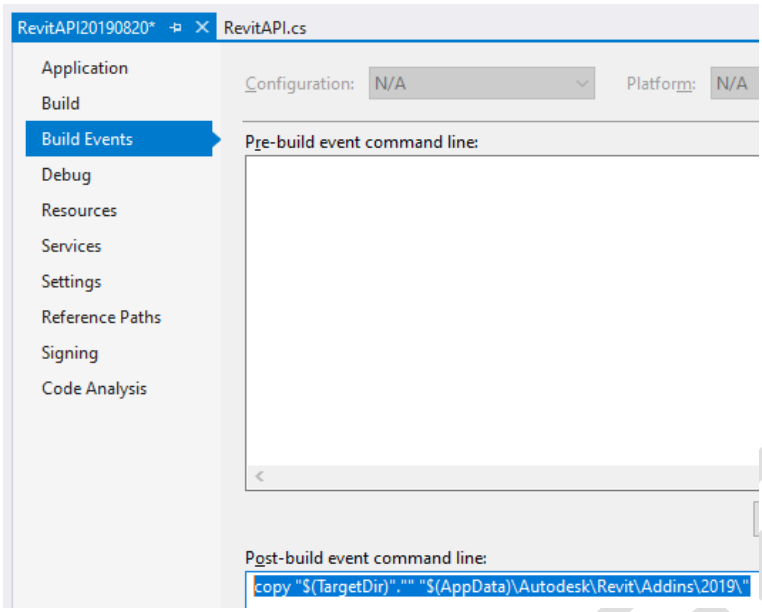
```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
  <AddIn Type="Command">
    <Name>SampleApplication</Name>
    <Assembly>C:\Temp\RevitPlugins\ TreinaOlaRevit20180731.dll</Assembly> (File and
Location where we will copy our dll, result of the file compiled)
    <AddInId>BC05297E-11B0-4CCC-A0F4-35AFC0DD12E7</AddInId> (Unique Plugin Id code,
which can be generated in Visual Studio)
    <FullClassName> TreinaOlaRevit20180731.OlaRevit</FullClassName> (
NameSpace and name of the class we created)
    <VendorId>ETlipse</VendorId>
    <VendorDescription>Edson Andrade, Rogerio Lima, Joel Diniz</VendorDescription>
  </AddIn>
</RevitAddIns>
```

The plug-in Id must be unique, and can be generated in Visual Studio itself, under Tools> Create GUID> 5. This is a number generated with a special algorithm to make it unlikely to coincide.

It is possible to automate the copy of the Manifest to the appropriate Revit folder, which understands the plug-ins to be executed. We must access, in Visual Studio, the Project menu, Properties submenu. In the project properties window, in the Built Events tab, Post-build box, we put the directive below, which will depend on the Revit in use, in the example 2019.

copy "$ (TargetDir)". "" "$ (AppData) \ Autodesk \ Revit \ Addins \ 2019 \"

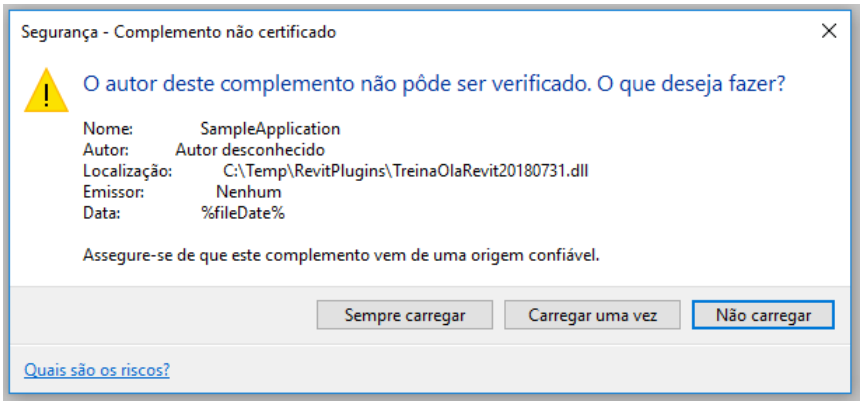




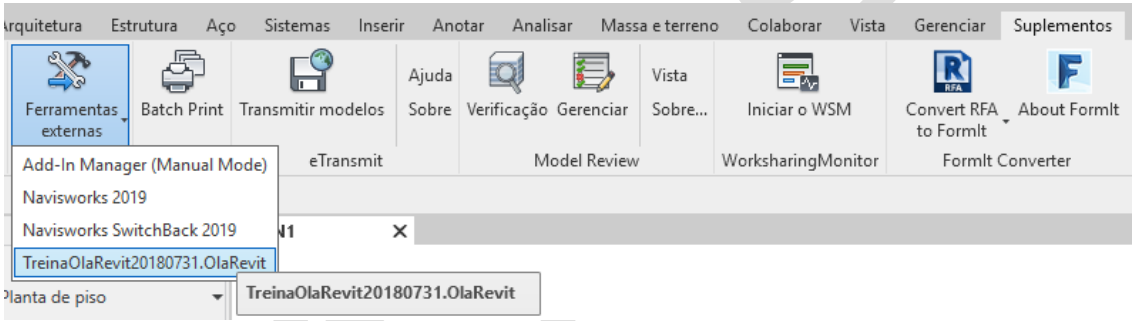
As mentioned, Revit will add files processed in the mentioned plugins folder as a plugin. These files basically consist of the addin, which needs to be correctly configured pointing to the dll, and the dll itself that will contain the plugin code. If you choose to always open the plugin, on your first access, these files will remain in the default folder, and the question about their processing will no longer be asked, which will happen automatically. In the test phase, it is therefore interesting to choose temporary loading. If you need to uninstall the plugin that you chose to always load, just delete or move the files from that folder.

Running the plug-in

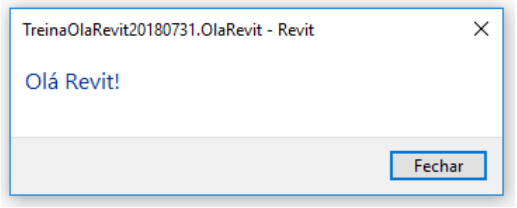
We launch Revit, and we will get the warning we described. For testing it may be interesting not to choose to always accept to run the plug-in.



The plug-in we created will be available in Add-ons> External tools. This will be the default, but it is possible to create new items and menus.



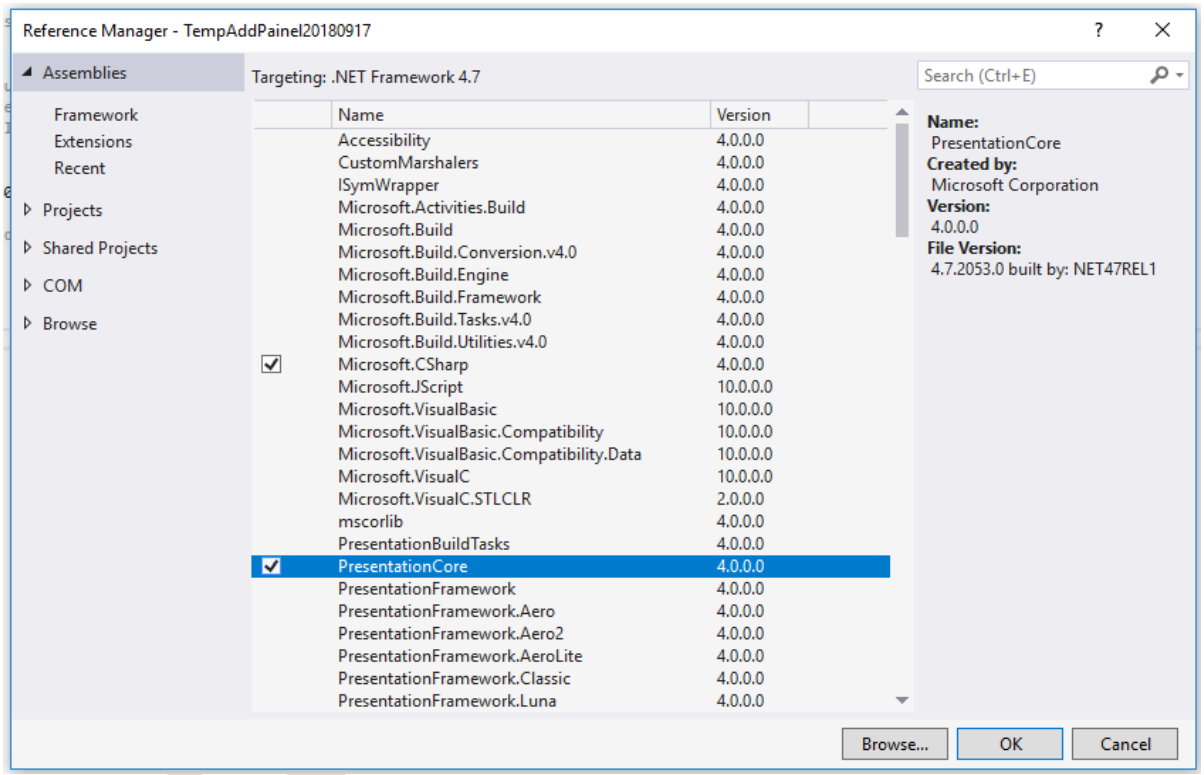
As intended, the execution, with the click on the item, will take us to the dialog screen with text that we programmed.

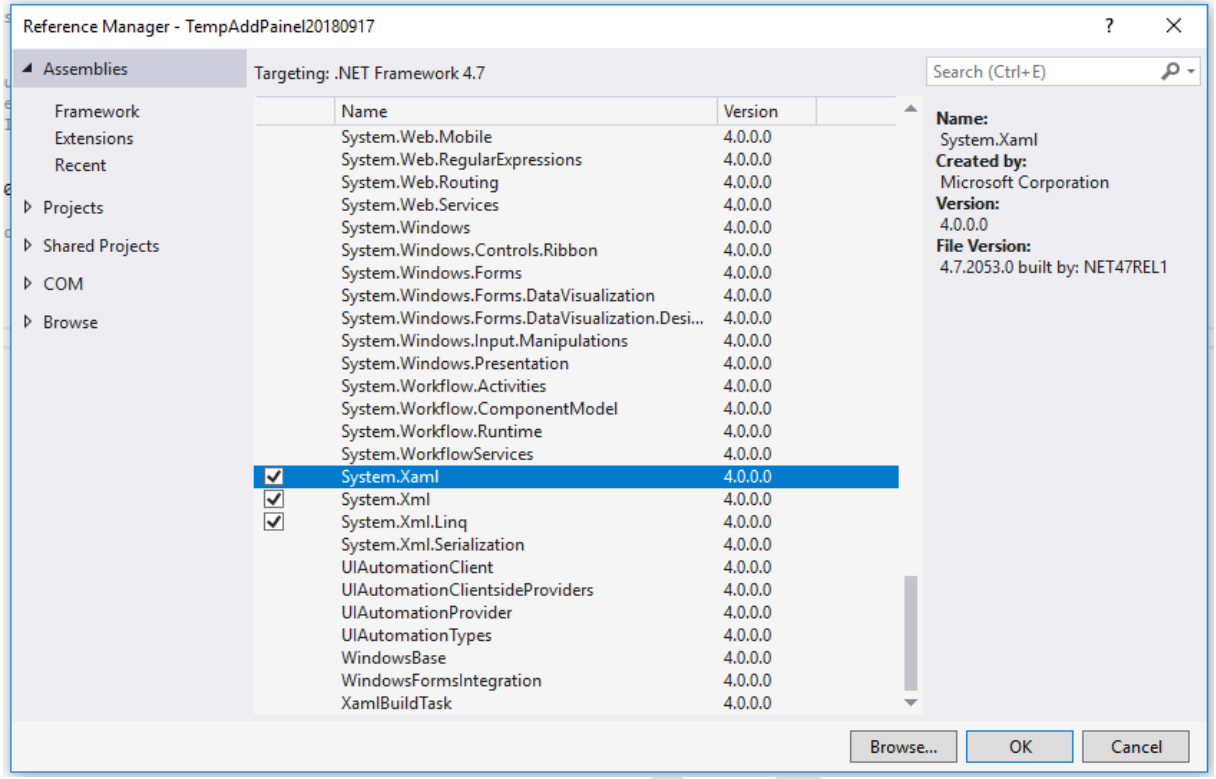


Customize Ribbon Panel

To create a plugin, it is possible to customize the Revit Ribbon panel and configure icons that direct the user to the custom plug-ins.

We created a project in Visual Studio, called “AddPanel”, as in the first example. This time we will need to add more references. We must add a reference to “PresentationCore”. The same must be done to add “System.Xaml”.





We renamed “Class1.cs” to “CsAddPanel.cs”, from Solution Explorer, right-click on the file, then rename option. We opened the “CsAddPanel.cs” file to edit it as shown below. Unlike the first application, which was Command-based, it is Application-based, and contains two abstract methods, OnStartup () and OnShutdown (). It is necessary to define an icon image and use its path in the code, as well as use the path of the dll, according to the following code. The created button must point to the dll that is to be executed when it is activated in Revit. The manifest file must be created in a similar way to the one made previously, this time with type "Application".

The complete code is:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.Attributes;
using Autodesk.Revit.UI.Selection;
using System.Windows.Media.Imaging;
```

```
namespace RevitAPIGuide2014_AddPanel_20180708{

    [Transaction(TransactionMode.Manual)]
    public class CsAddPanel : Autodesk.Revit.UI.IExternalApplication {


        Result IExternalApplication.OnShutdown(UIControlledApplication application) {

            return Result.Succeeded;
        }

        Result IExternalApplication.OnStartup(UIControlledApplication application) {

            // Add new ribbon panel
            RibbonPanel ribbonPanel = application.CreateRibbonPanel("NewRibbonPanel");

            // Create a push button in the ribbon panel "NewRibbonPanel"
            // the add-in application "Helloworld" will be triggered when button is
pushed

            PushButton pushButton = ribbonPanel.AddItem(new
PushButtonData("Helloworld", "Helloworld",
@"C:\Temp\RevitPlugins\RevitAPIGuid2014_20180702.dll",
"RevitAPIGuid2014_20180702.HelloWorld")) as PushButton;

            // Set the large image shown on button
            Uri uriImage = new Uri(@"D:\Temp\Temp_32x32.png");
            BitmapImage largeImage = new BitmapImage(uriImage);
            pushButton.LargeImage = largeImage;

            return Result.Succeeded;
        }

    }

}
```

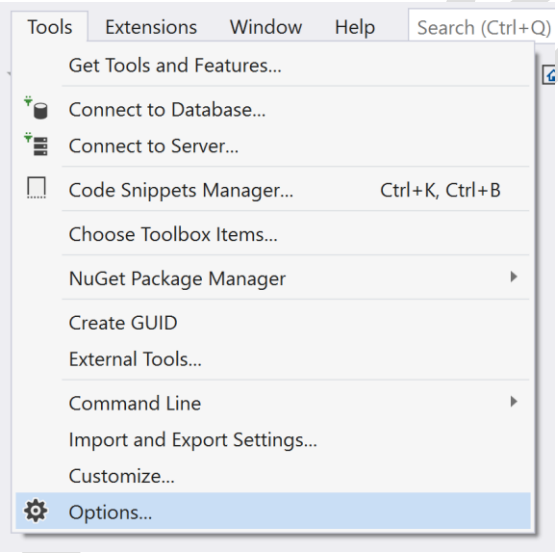
The manifest file will be:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<RevitAddIns>
  <AddIn Type="Application">
    <Name>SampleApplication</Name>
    <Assembly>C:\Temp\RevitPlugins\RevitAPIGuide2014_AddPanel_20180708.dll</Assembly>
    <AddInId>BC05297E-11B0-4CCC-A0F4-35AFC0DD12E7</AddInId>
    <FullClassName>RevitAPIGuide2014_AddPanel_20180708.CsAddPanel</FullClassName>
    <VendorId>ETlipse</VendorId>
    <VendorDescription>Edson Andrade, Rogerio Lima, Joel Diniz</VendorDescription>
  </AddIn>
</RevitAddIns>
```

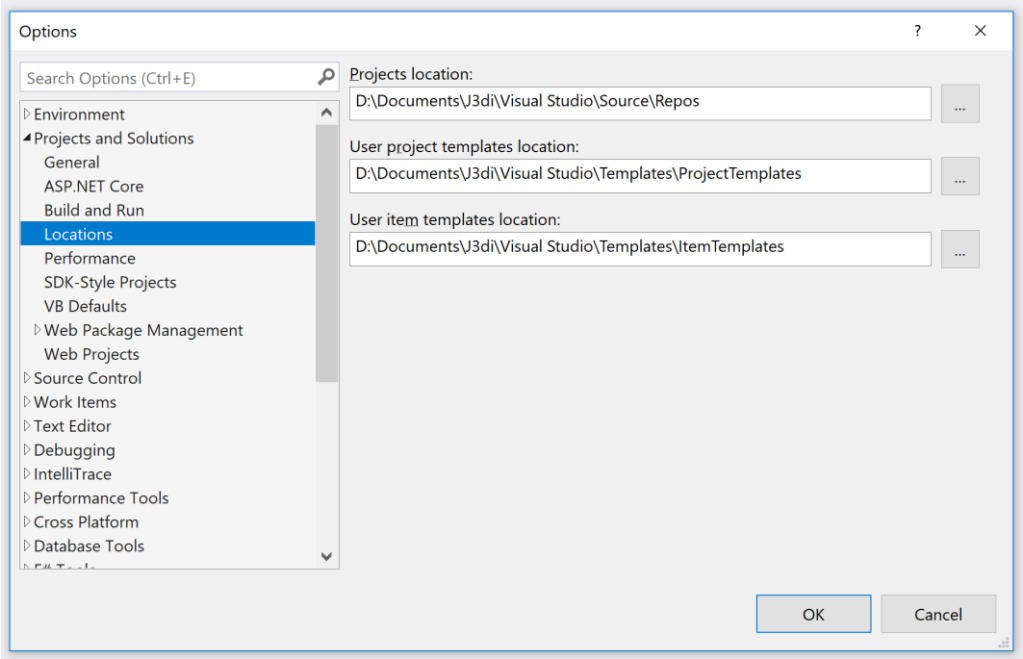
Some configurations are necessary to make use of the WPF interface with the Revit API, some very specific and not so usual. ETLipse provides a specific template for this need, facilitating the use of this important resource on the .Net platform.

The eTLipse template can be downloaded from the eTLipse page to be used in Visual Studio and to facilitate the use and configuration of WPF applications using the Revit API. The template file consists of a compressed file in .zip format that must be placed in the appropriate directory. It is important to check the default directory used by the version in use of Visual Studio.

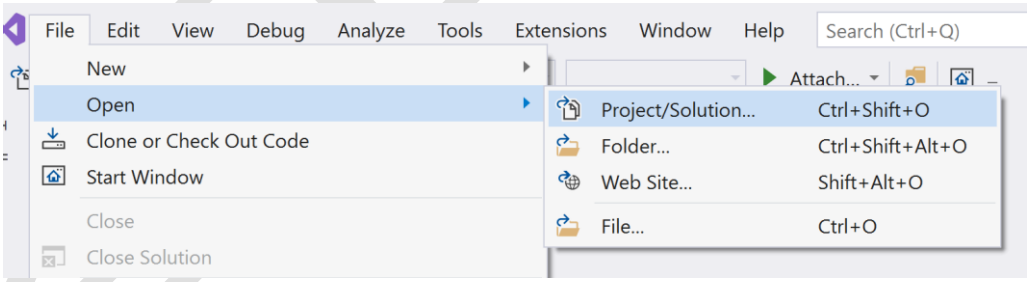
The configuration of the path used to load the templates is registered in the configuration menu. Use the "Options ..." option in the "Tools" menu.



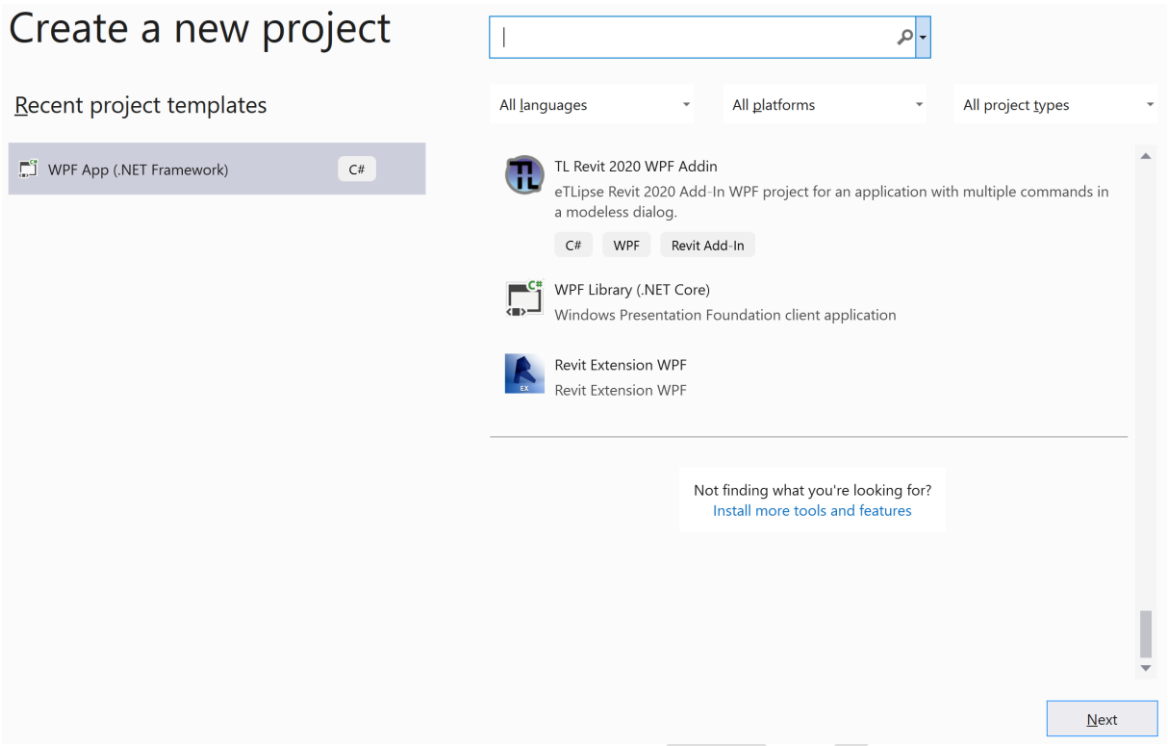
In the options window, we must choose the item “Projects and Solutions”, and, as a subtopic of this option, we will have the item “Locations”. The path options will be displayed in “Projects Location” in this same window, where we should observe the item “User Project templates location”.



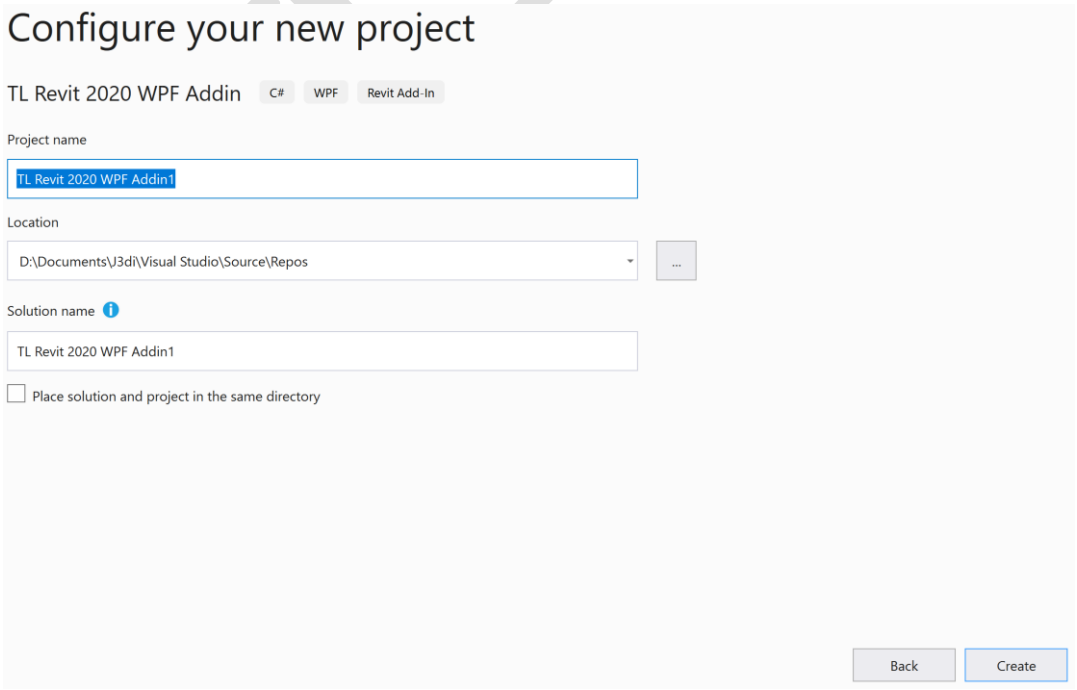
The configuration file must be placed inside this page to be loaded as a template by Visual Studio. Once the copy is made to the correct folder, we will be able to create a new project based on the chosen template. We can start a new project from the "Project" item in the "New" submenu of the "File" menu.



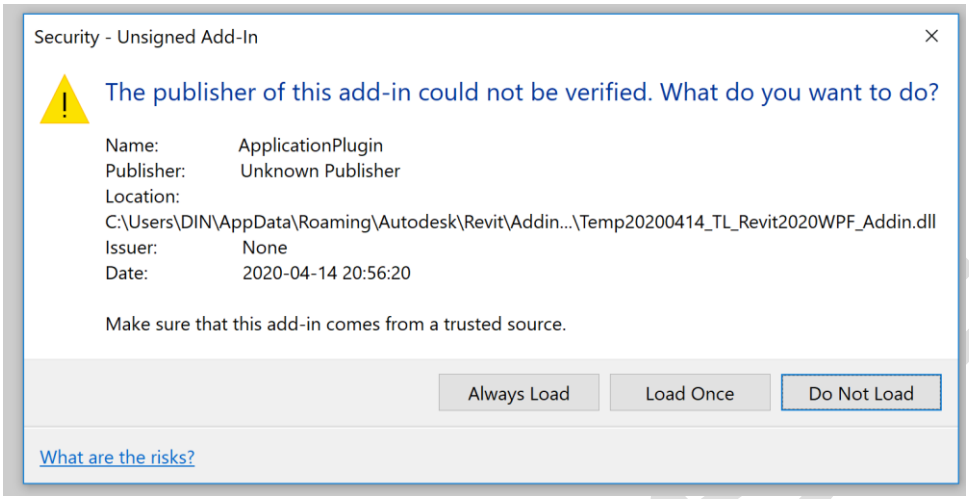
We can see that Visual Studio will present the eTLipse template as one of the options for a new project, as shown in the following figure. We need to select the displayed option "TL Revit 2020 WPF Addin" and click on "Next".



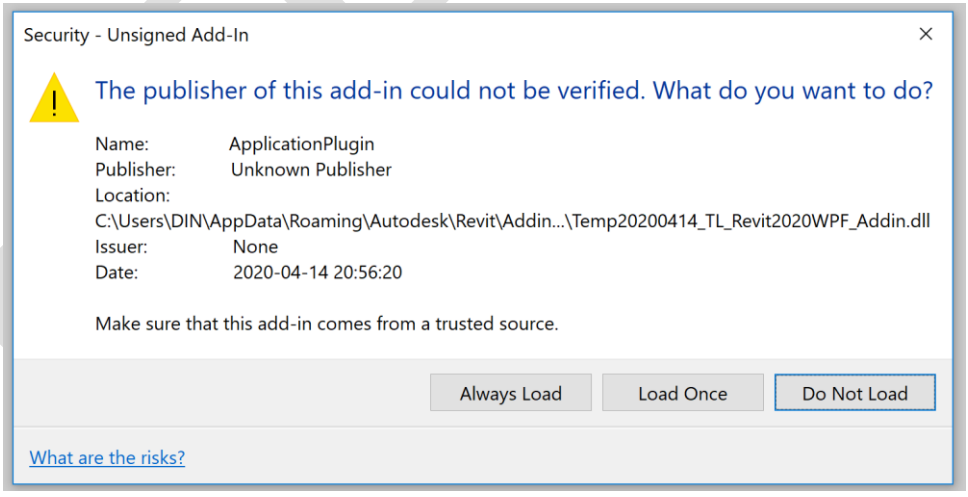
On the screen that follows we must choose the name of the project and the solution that we will build, as well as the path where we will store our solution. Having chosen the options, we click on "Create" to create our solution.

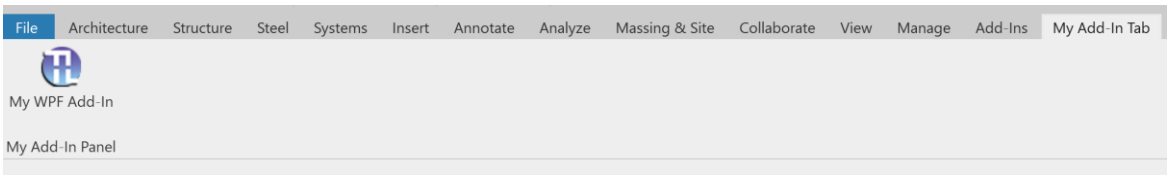


The project is created and will be open for editing. This template provides an example from which it is possible to make the necessary customizations for the solution you want to create. The code is widely commented so that it can be easily edited.



Once you have built your application on the template, you can then compile your solution. The project will open Revit and launch its solution. It is important to note that, for the protection of Revit, the initial screen will be displayed to authorize the use of the application, which may only be for once, forever, or may not allow loading. Finally, you can open a project in Revit and the created solution will be in the menu that will have the name configured in the solution.





The projected icon will be available to launch your solution. The projected window will be launched when you click on your application icon.