24. April 2015: Unity Editor Extension Dynamic-Link Library

Table of Contents

1. What it is about

In this workflow tutorial you will learn how to

- create a custom Editor with tooltips
- add a tooltip with description to intellisense
- hide public members from intellisense
- document the code
- create a dynamic-link library for the Unity Engine & Editor
- automatically create a documentation

⇑ table of contents ⇑

---

2. Why a dynamic-link library

To create a managed Dynamic-Link Library (.dll for short) has the following advantages

- public members, that are needed for the custom Editor, can be hidden from intellisense
- it's handier than dozens of scripts
- not *everyone* can **easily** read, edit and possibly steal the source code

⇑ table of contents ⇑

---

3. What you need

To complete this tutorial you'll need the following *free* tools:
Unity with MonoDevelop to create the project, code, dynamic-link library and application
DoxyGen to automatically create the documentation from code

Note: This tutorial was created with Unity 5.0.1 on a Windows 8.1 PC.

⇑ table of contents ⇑

---

4. Download the Unity project

To download the project will help you to better follow and understand this tutorial.

Download abstractexample.unitypackage

Download the file, create a new project in Unity (menu bar -> File -> New Project...), call it e.g. *"Unity Editor Extension .dll"*.
Import the Unity package in your new project (menu bar -> Assets -> Import Package -> Custom Package... -> *.unitypackage), then click the Import button in the window where the files are listed that are imported.

Double click / open the scripts *AbstractExample.cs* (in the *Scripts* folder of the project window) and *AbstractExampleEditor.cs* (in the *Editor* folder of the project window).
Then open the scene *AbstractExampleScene* (in the main folder (Assets) of the project window).

In general you can also open MonoDevelop by using Unity's menu bar -> Assets -> Sync MonoDevelop Project.

Note: "Abstract" in this case has nothing to do with the C# keyword *abstract*. It only means that this is a theoretical example! I chose an abstract example to keep it as simple as possible.

⇑ table of contents ⇑

---

5. Custom Editor with tooltips

A custom editor displays custom properties of your Unity standard script (called *AbstractExample.cs* in this example) in the Unity Inspector window.

To create a custom editor you need to extend your standard Unity script (that normally derives from the MonoBehaviour class) in MonoDevelop with a CustomEditor script (that derives from the Editor class and is called *AbstractExampleEditor.cs* in this example).

First you need to place this code inside the Editor script to tell the Editor script that it extends the standard script:

```
[CustomEditor( typeof( AbstractExample ))]
```

To show the public member variable

```
public Transform t_Abstract;
```

of the standard Unity script *AbstractExample.cs* in the Inspector window with a bold heading and tooltips you first need to connect this public member variable with the editor script.
This is done by the following lines of code in the Editor script *AbstractExampleEditor.cs*:

```
private SerializedProperty sp_t_abstract;

private void OnEnable( )
{
  sp_t_abstract = serializedObject.FindProperty( "t_Abstract" );
}
```

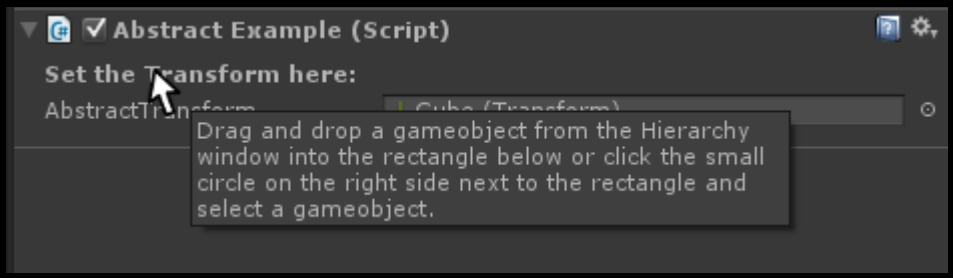
If you now change the reference to *t_Abstract* this will also automatically change the content of the variable *sp_t_abstract*.

Display the variable in Unity's inspector window by adding the following line of code to the editor script *AbstractExampleEditor.cs*:

```
EditorGUILayout.PropertyField( sp_t_abstract, new GUIContent( "AbstractTransform", "Som..." ));
```

The first string of *GUIContent* names the variable and the second string is the text shown by the tooltip when the name is hovered by the mouse cursor.
The bold heading works the same way.

In the Inspector window hover the heading and the variable name to see the tooltips.



⇑ table of contents ⇑

6. Connect the Editor script with the standard script

In much cases the standard script needs feedback from the Editor script. For example if you need to change something when (like in this example) the reference (or value) of a variable was changed.

Note: When you create a build application of a Unity project the UnityEditor namespace and also the Editor script is cut away and is not included into the build application. So you need to place all code you possibly want to execute at the runtime of the build application (normally) inside Unity standard scripts.

To create a connection you need the following code inside the Editor script *AbstractExampleEditor.cs*:

```
private AbstractExample s_target;

private void OnEnable( )
{
  s_target = ( AbstractExample )target;
}
```

This creates a variable that references to the standard script *AbstractExample.cs* and with the line

```
s_target.AbstractTransformChanged( );
```

it's possible to call the public method *AbstractTransformChanged* of the class *AbstractExample* from within the class *AbstractExampleEditor*.

⇑ table of contents ⇑

---

7. Notice changes in the Inspector window

To know when a change was made from the user, to call a method in the script, you need to create a "change check" that starts before the observed variable *sp_t_abstract* in the code and ends behind it.

```
EditorGUI.BeginChangeCheck( );
EditorGUILayout.PropertyField( sp_t_abstract, new GUIContent( "AbstractTransform", "Som..." ));
if( EditorGUI.EndChangeCheck( ))
{
  serializedObject.ApplyModifiedProperties( );
  s_target.AbstractTransformChanged( );
}
```

If a change was made and *EditorGUI.BeginChangeCheck( )* is true the change in *sp_t_abstract* is applied to *t_Abstract* with the first line of code within the *if* condition and then the method *AbstractTransformChanged* is called and several components are attached to the gameobject *t_Abstract* references to.

⇑ table of contents ⇑

8. Add tooltips with description to intellisense

Add tooltips to the intellisense of public (or even private) members with this code:

```
/// <summary>Count the amount of components of a certain type that are attached to a gameo...
/// <para>T is the type of the component to count.</para>
/// </summary>
/// <param name="targetGameObject">The gameobject its certain type of compon...</param>
/// <returns>The amount of certain components attached to the gameobject.</returns>
/// <example>To call this method from anywhere use e.g.:
/// <code>
/// int numberOfBoxColliders;
/// numberOfBoxColliders = AbstractExample.GetComponentCount&lt;BoxCollider&gt;( m... );
/// </code>
/// </example>
public static int GetComponentCount<T>( GameObject targetGameObject )
where T : Component
{
  Component[] componentTypeToCount;
  componentTypeToCount = ( Component[] )targetGameObject.GetComponents( typeof( T ));
  return componentTypeToCount.Length;
}
```

The /// comment slashes tell MonoDevelop (and also e.g. Microsoft Visual Studio) that these XML comments describe the member below in more detail.
<summary> should describe what the member is doing / good for.
<param name="parameterName"> describes a parameter of a method.
<returns> describes the content that is returned by the method.
<example>...<code> gives the user a multiline example how to use the member.

Note: Make sure you replace the pre-defined XML characters that the tooltips are displayed proper.

⇑ table of contents ⇑

---

9. Hide public members from intellisense in the standard script

Some members in the standard script need to be public that the connected Editor script can access them. But in the final product the user should only see the methods that she/he is allowed to use and that are documented.
To hide these members just add this code above the member you want to hide from intellisense like this:

```
[System.ComponentModel.Browsable( false ),
System.ComponentModel.EditorBrowsable(System.ComponentModel.EditorBrowsableState.Never)]
public Transform t_Abstract;
```

Note: To hide this variable from intellisense only shows that you can hide all kind of members. It is still accessible from outside.

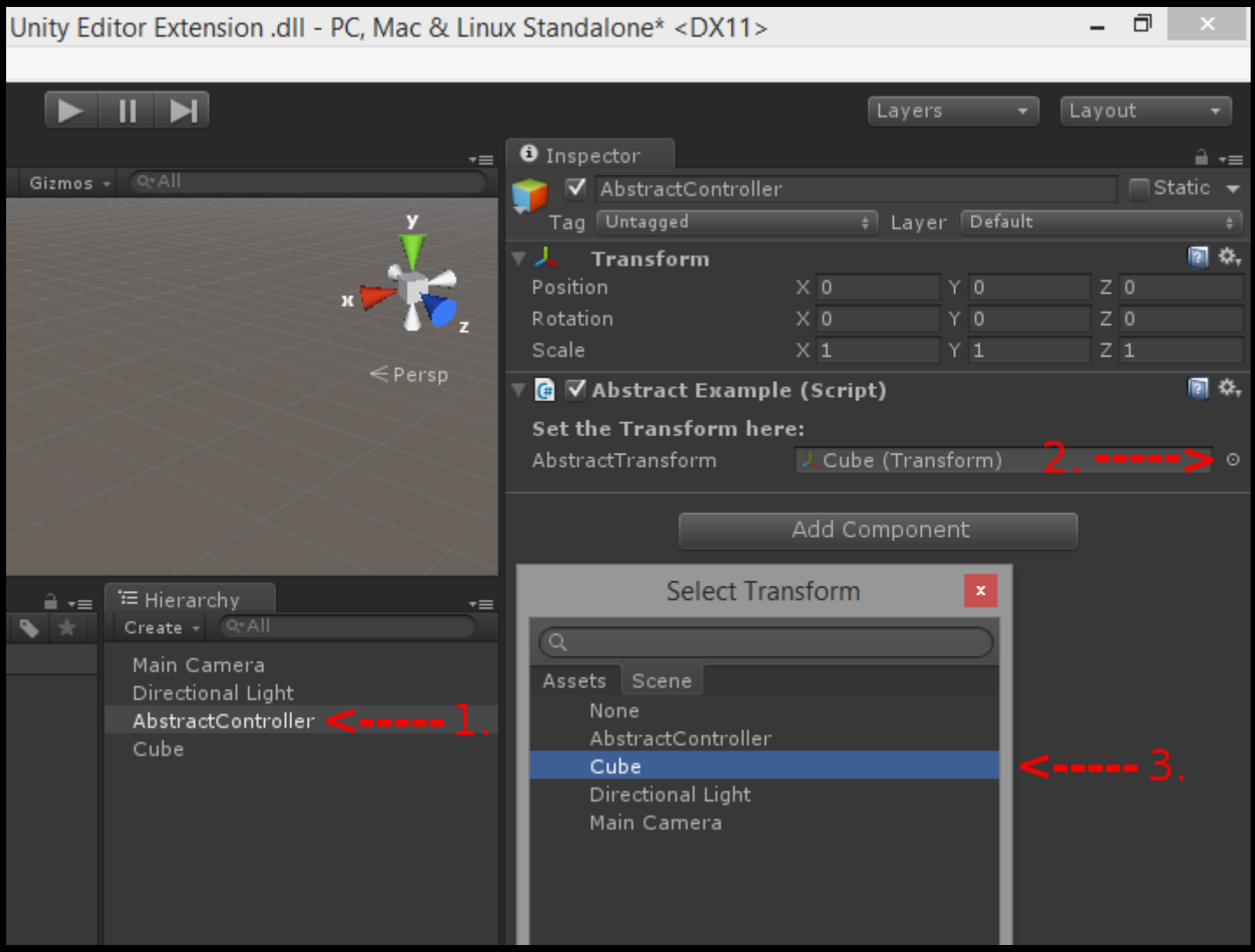
Note2: This will only work in the build .dll! (So.. let's test it after the script was build to a .dll.)

⇑ table of contents ⇑

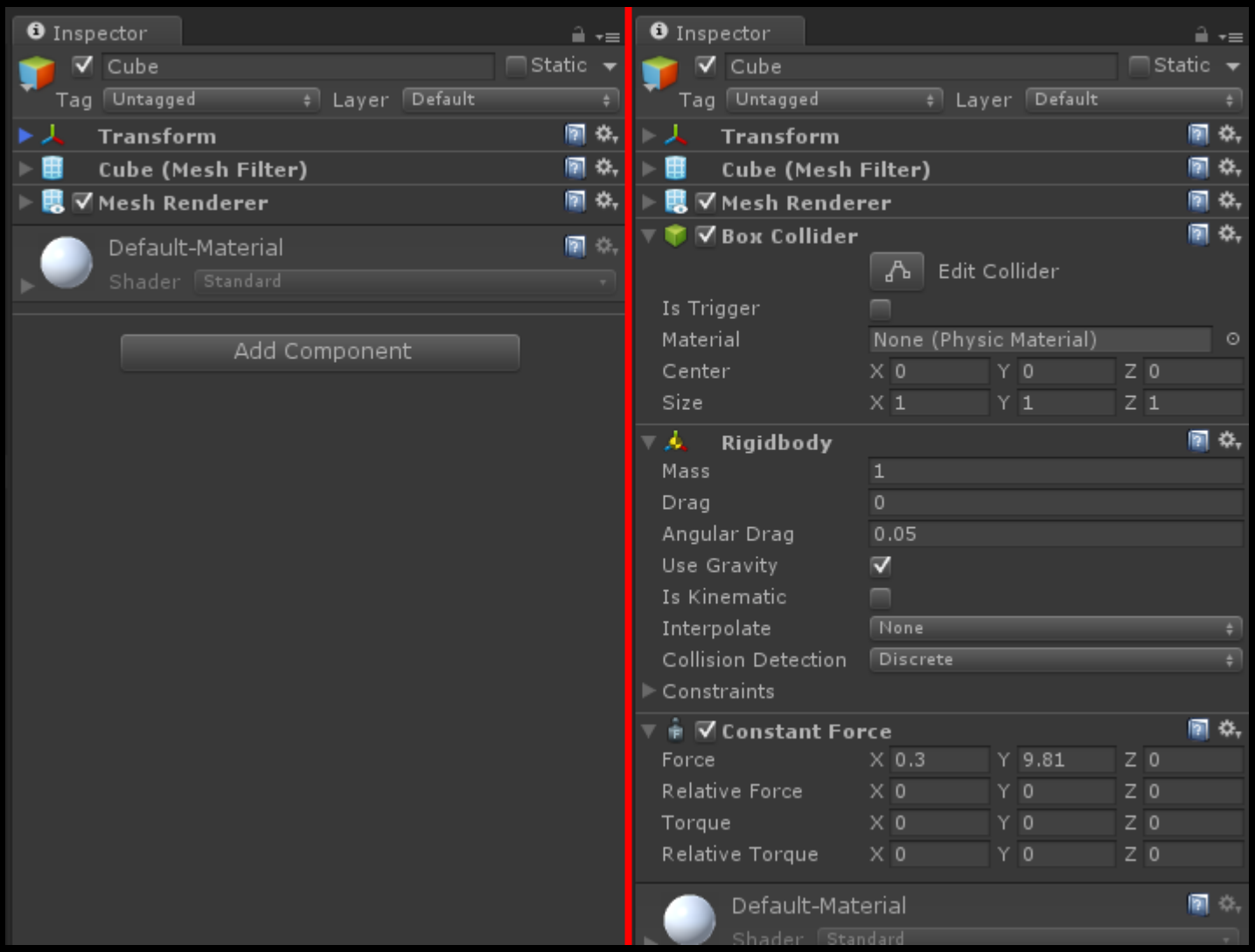
---

10. Use and Play the scene

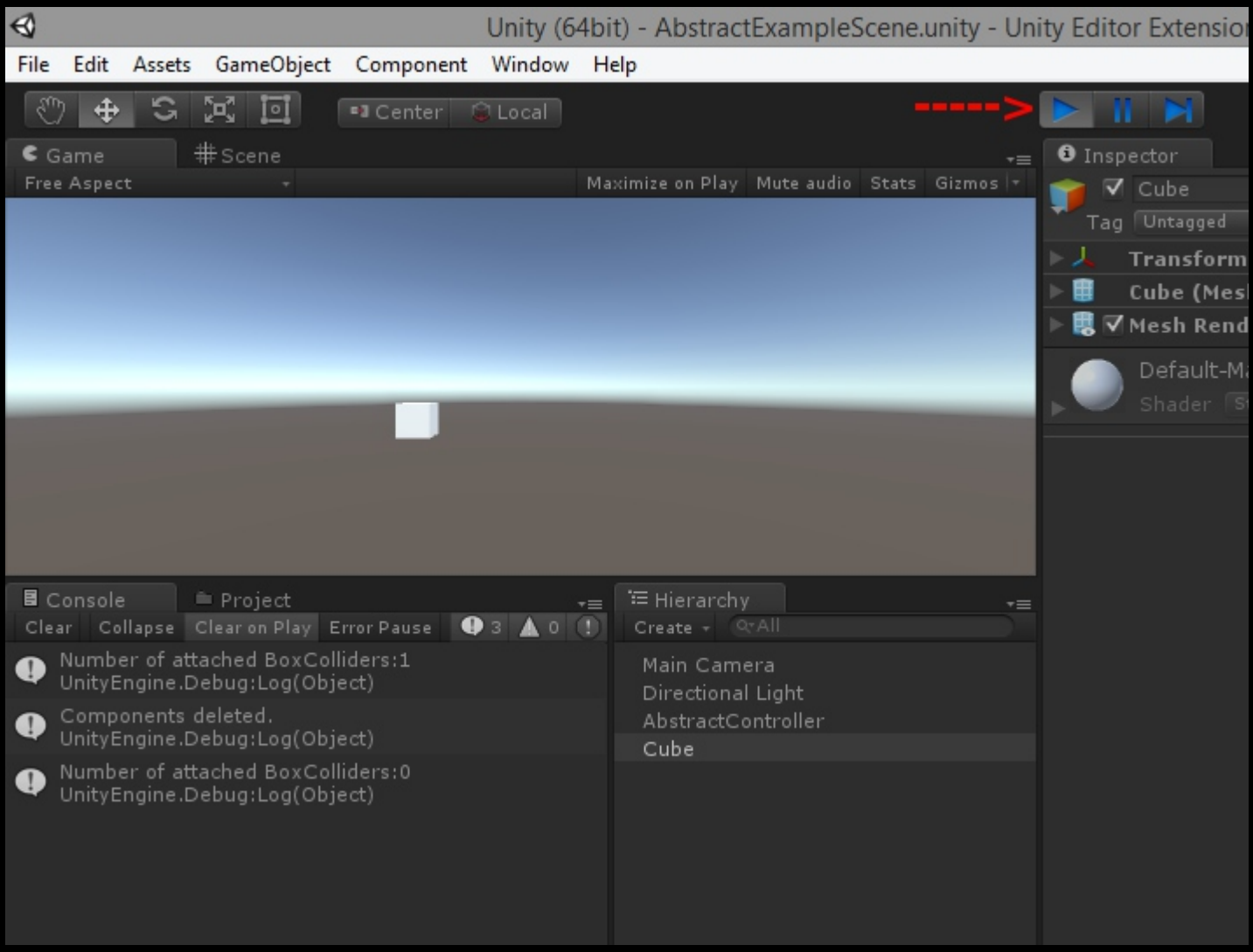
Let's test the example scene *AbstractExampleScene*.
Follow the **screenshots** and their **descriptions** below how to do this.



Select the *AbstractController* gameobject in the Hierarchy window (1.). Now you can see the custom Editor in action. To connect the Cube gameobject to the script click the small circle (2.) in the Inspector window. A *Select Transfrom* window will show up. Here select the Cube gameobject (3.).



The method *AttachColliderRigidbodyAndHoveringConstForceIfNeeded* in the standard script *AbstractExample.cs* was called when the transform reference in the Inspector window was changed.
(Left: before -- Right: after)



Click the Play button to run the scene. The Start method of the script *AbstractExample.cs* is called and generates the output in the console. The cube doesn't move.

If you would comment / change the lines 108 & 109 of code in the Start method like this

```
//DeleteColliderRigidbodyAndConstForceIfPresent( t_Abstract.gameObject );
Debug.Log ( "Components NOT deleted." );
```

the three attached components are not deleted and the Cube gameobject will move to the right on Play.

⇑ table of contents ⇑

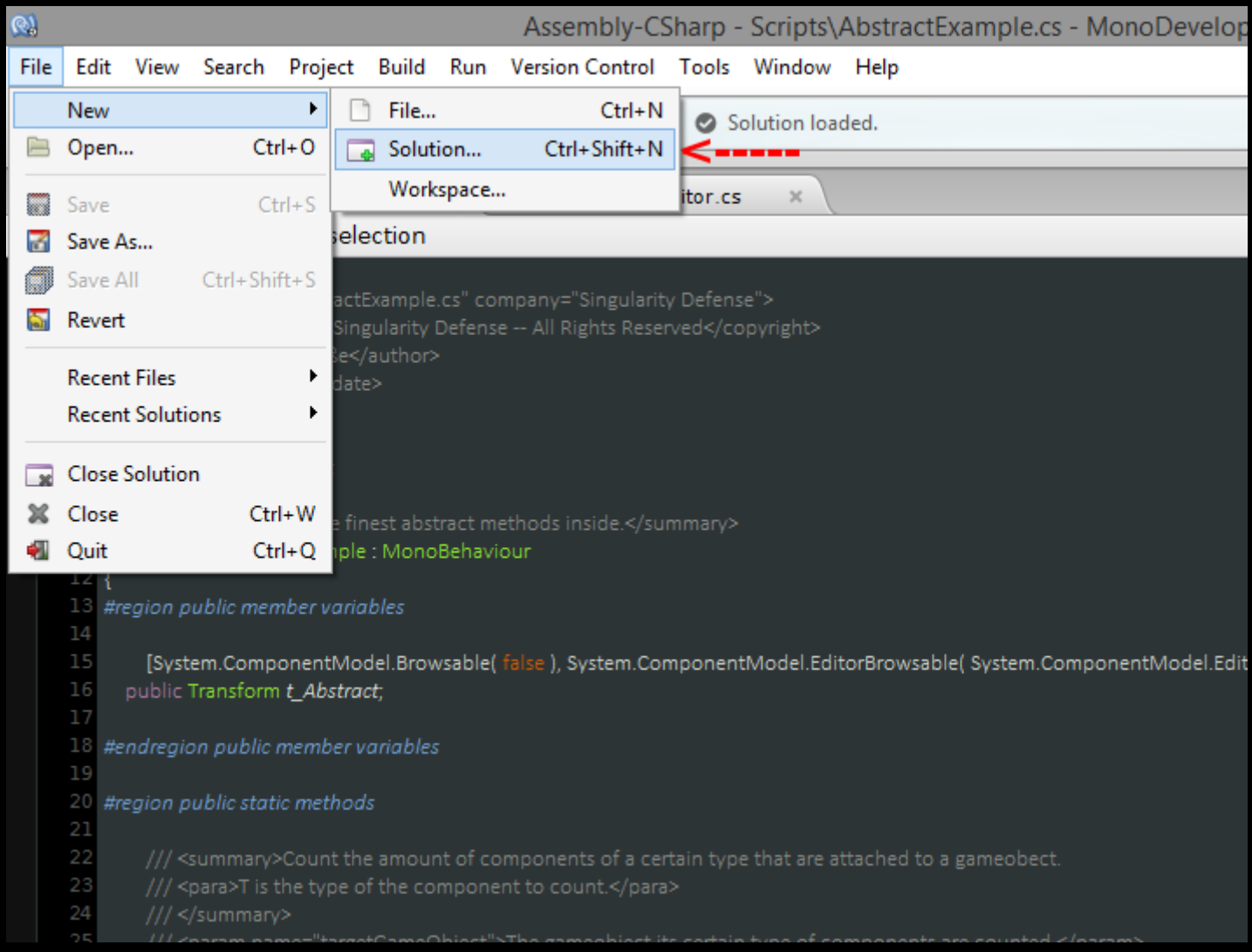
---

11. Create a dynamic-link library for the Unity Engine & Editor

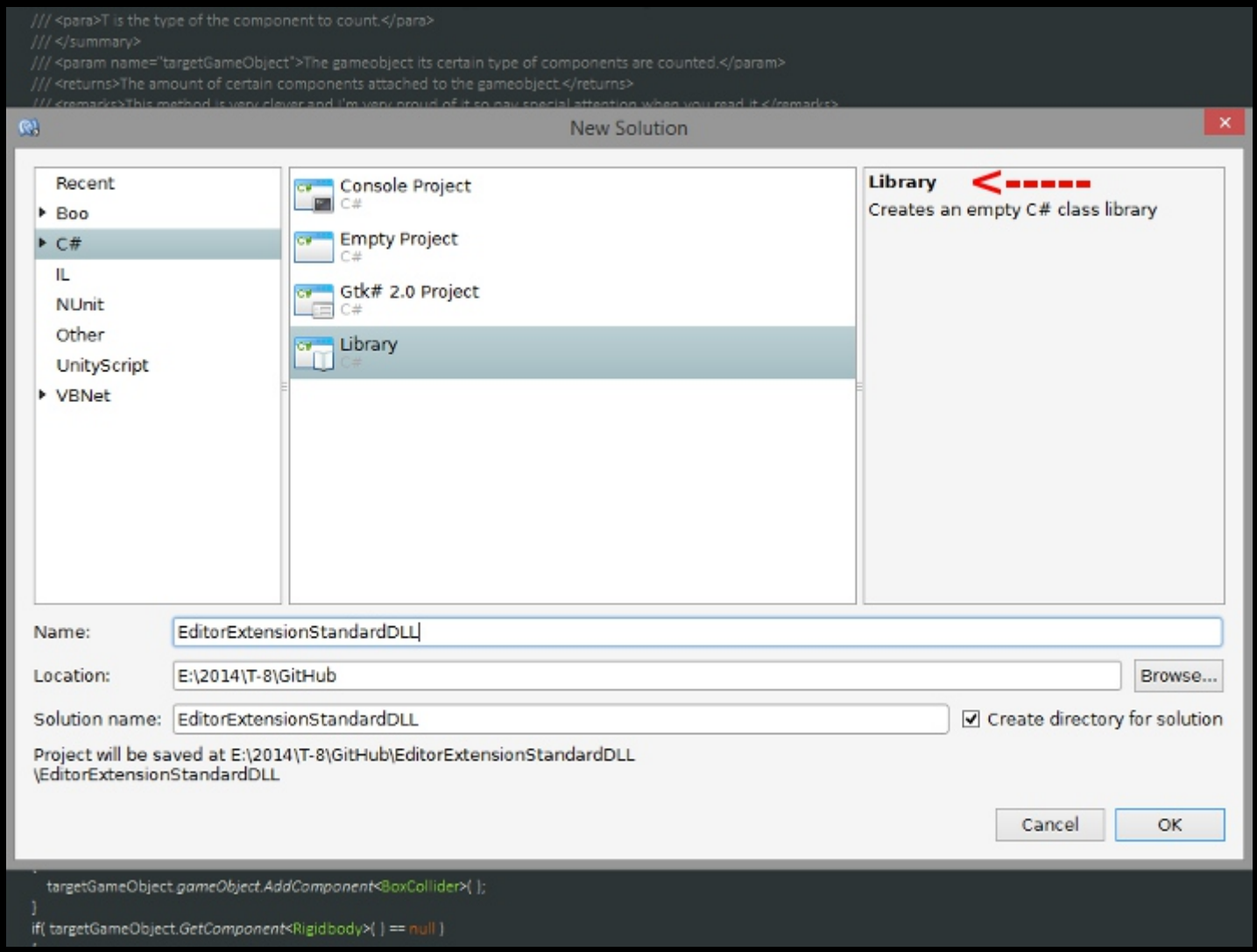
To wrap the code in a .dll with MonoDevelop it is necessary to create two .dlls. One .dll includes the standart script and the other .dll includes the Editor script that will be cut off when the project is build as application. This' the reason why two .dlls are needed.

11.1. UnityEngine namespace

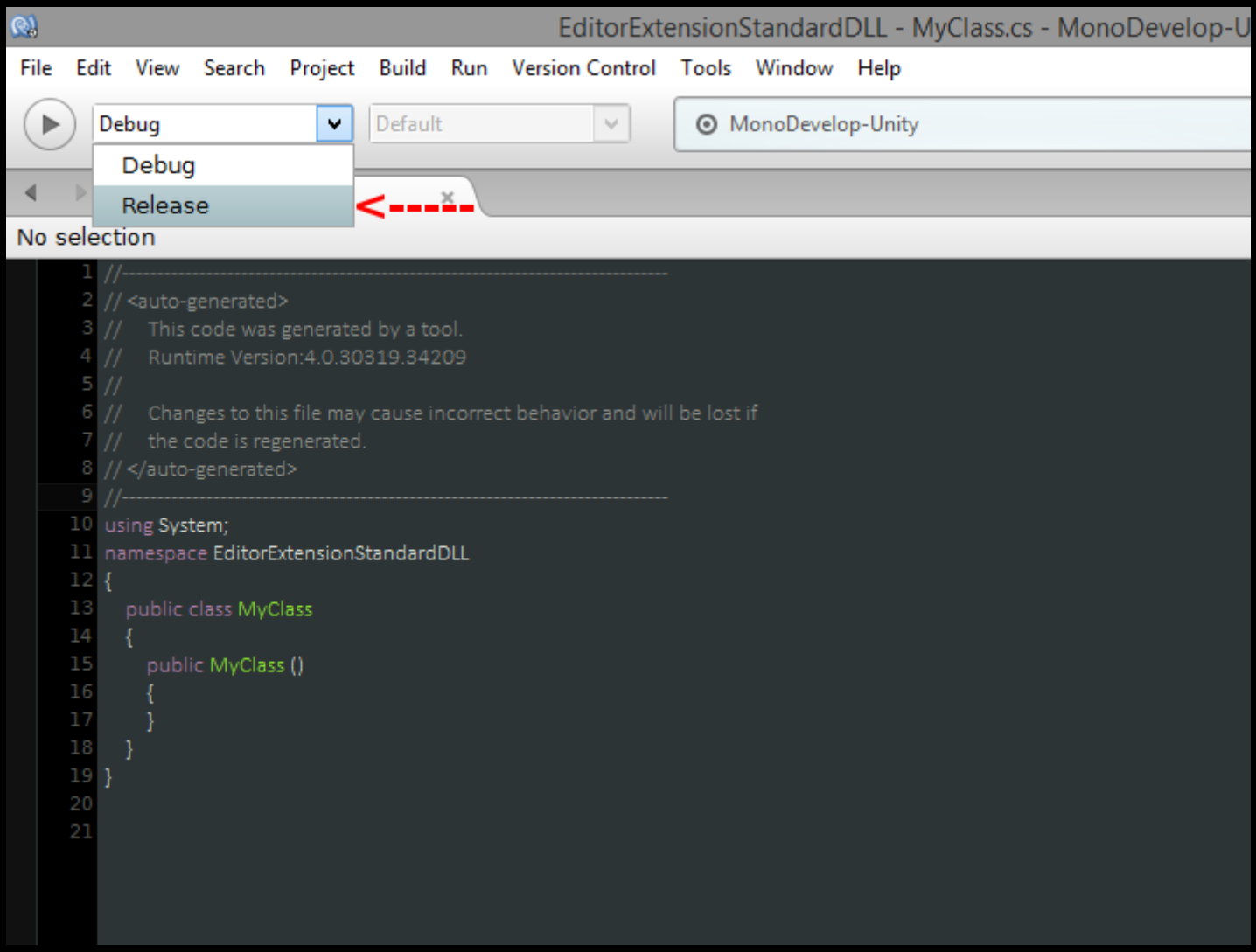
Create an new project in MonoDevelop and build the Editor script *AbstractExample.cs* as .dll.
Follow the **pictures** and the **descriptions** how to do this.



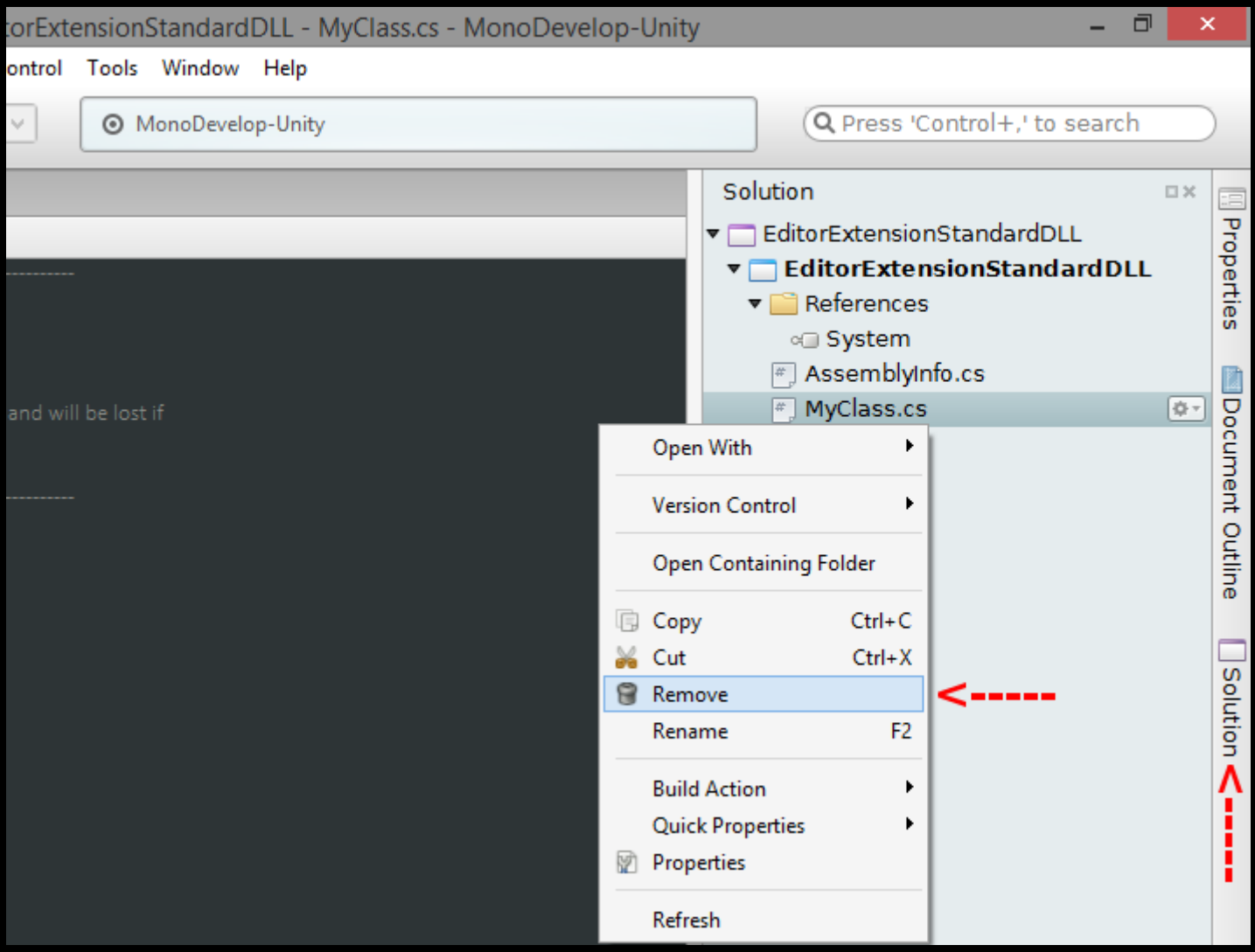
Create a new Solution.



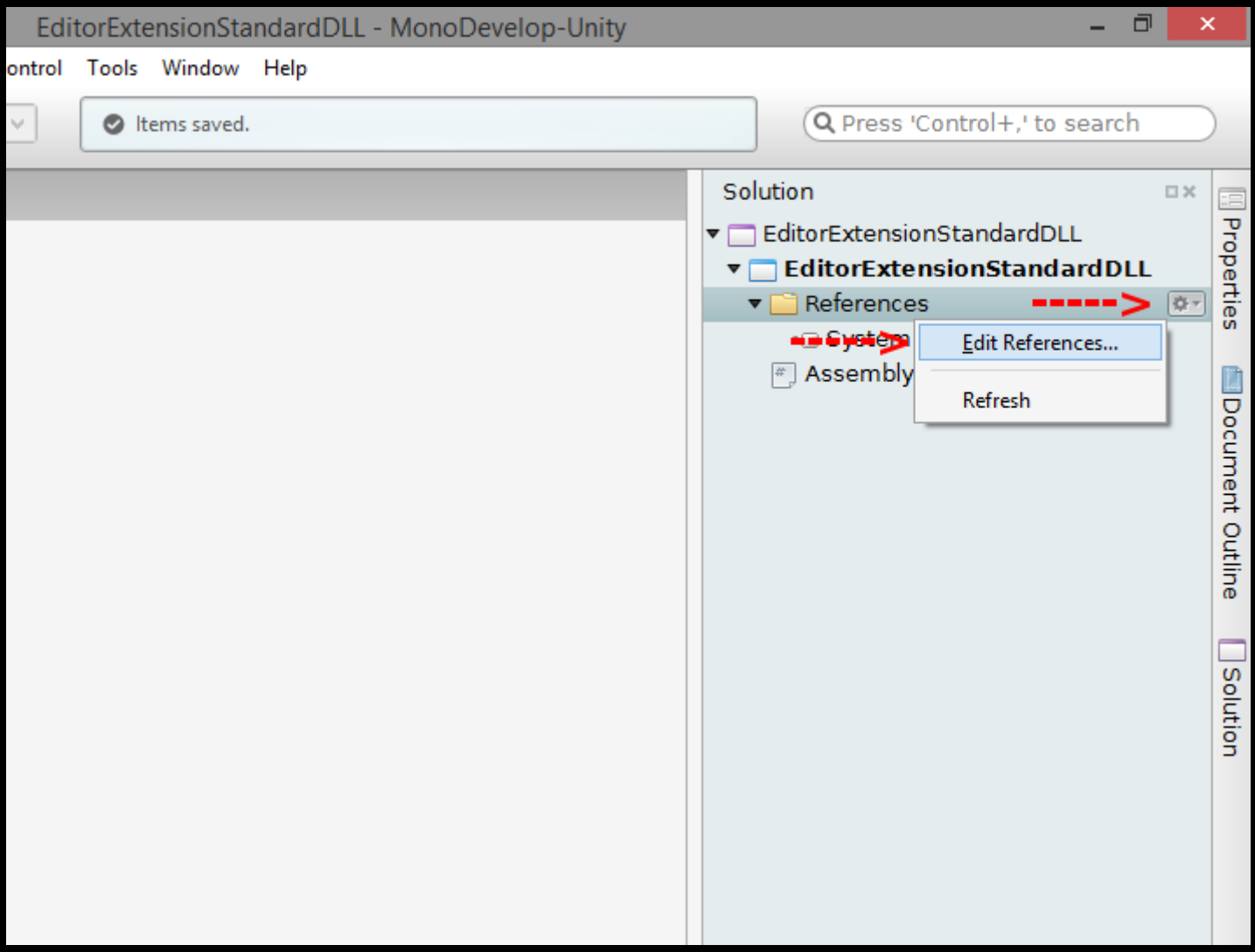
In the *New Solution* window select that you want to create a *C# -> Library*, name it *EditorExtensionStandardDLL* and choose a proper location to save it. Then click the OK button.



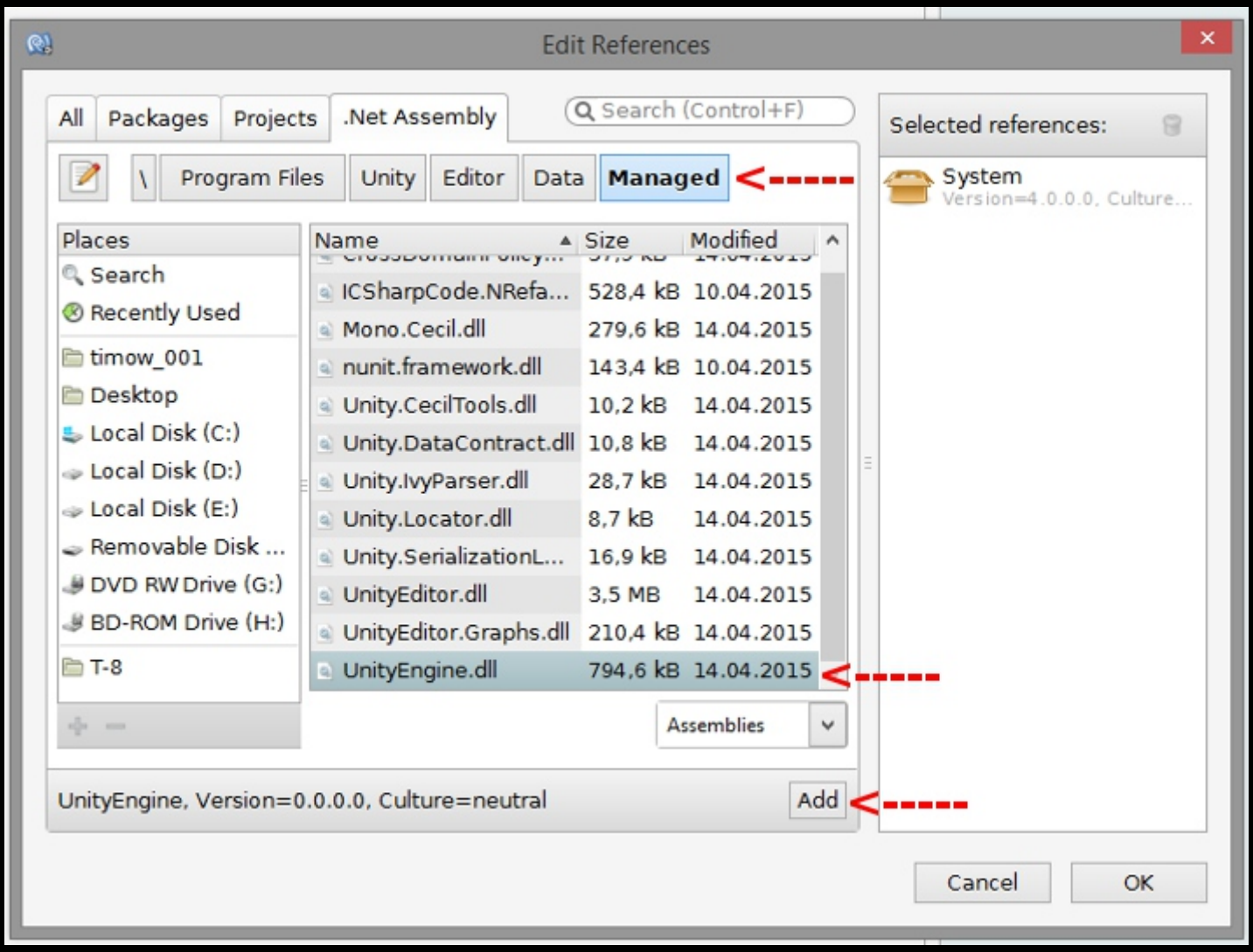
Change the build mode to *Release* in MonoDevelop because the source code is working in the Editor, so we assume it's also working as a build .dll.



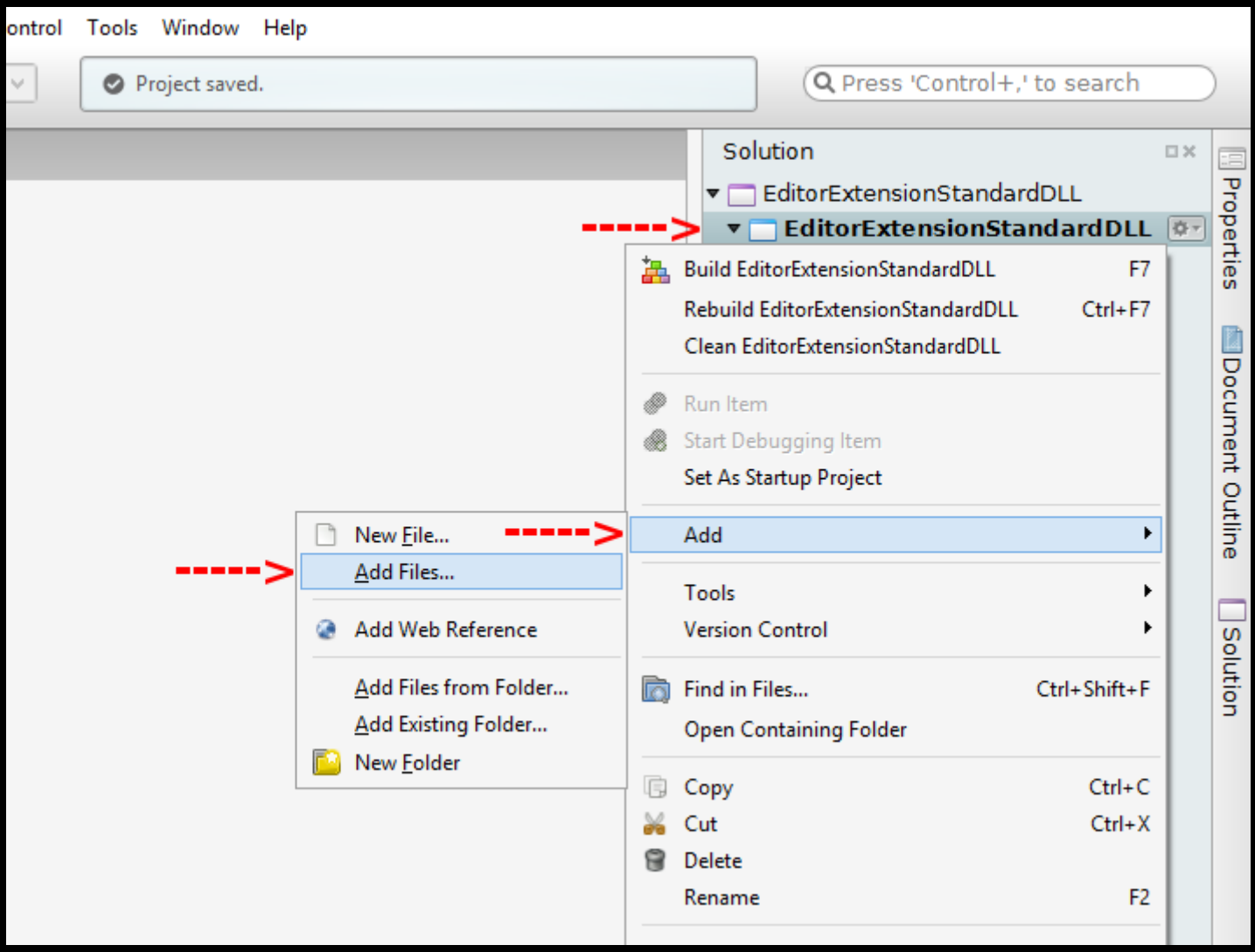
Now delete the automatically created file *MyClass.cs* in the solution window. (If you don't see the solution window tab you can let it appear with *menu bar -> View -> Pads -> Solution*.) We won't need it.



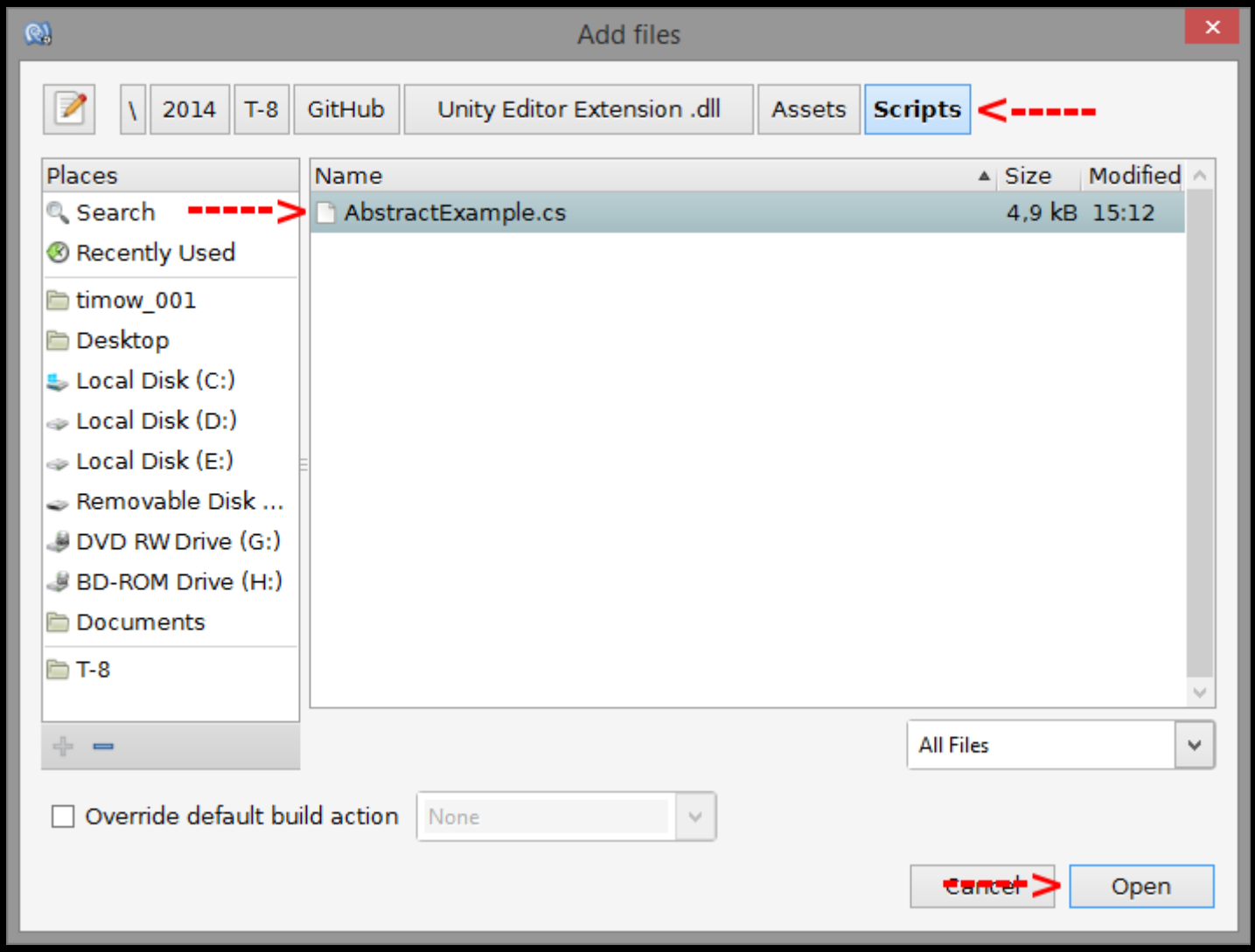
In the solution window click the small gear button and choose *Edit References...* to add references to other necessary .dlls.



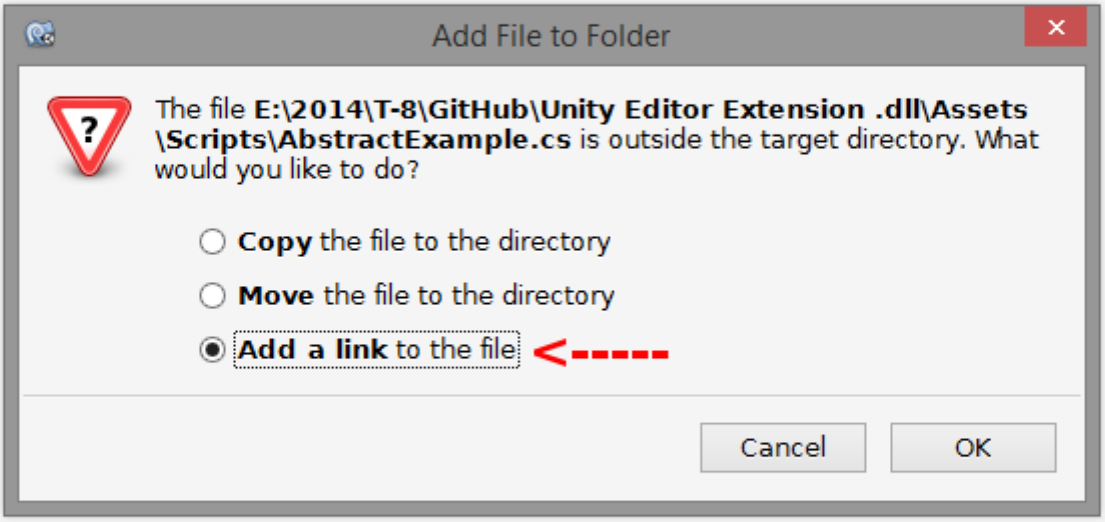
In the *Edit References* window change to the *.Net Assembly* tab and navigate to the *Managed* folder inside the Unity folder to select the *UnityEngine.dll*. Then add it to the *Selected references:* list on the right side. Click OK.



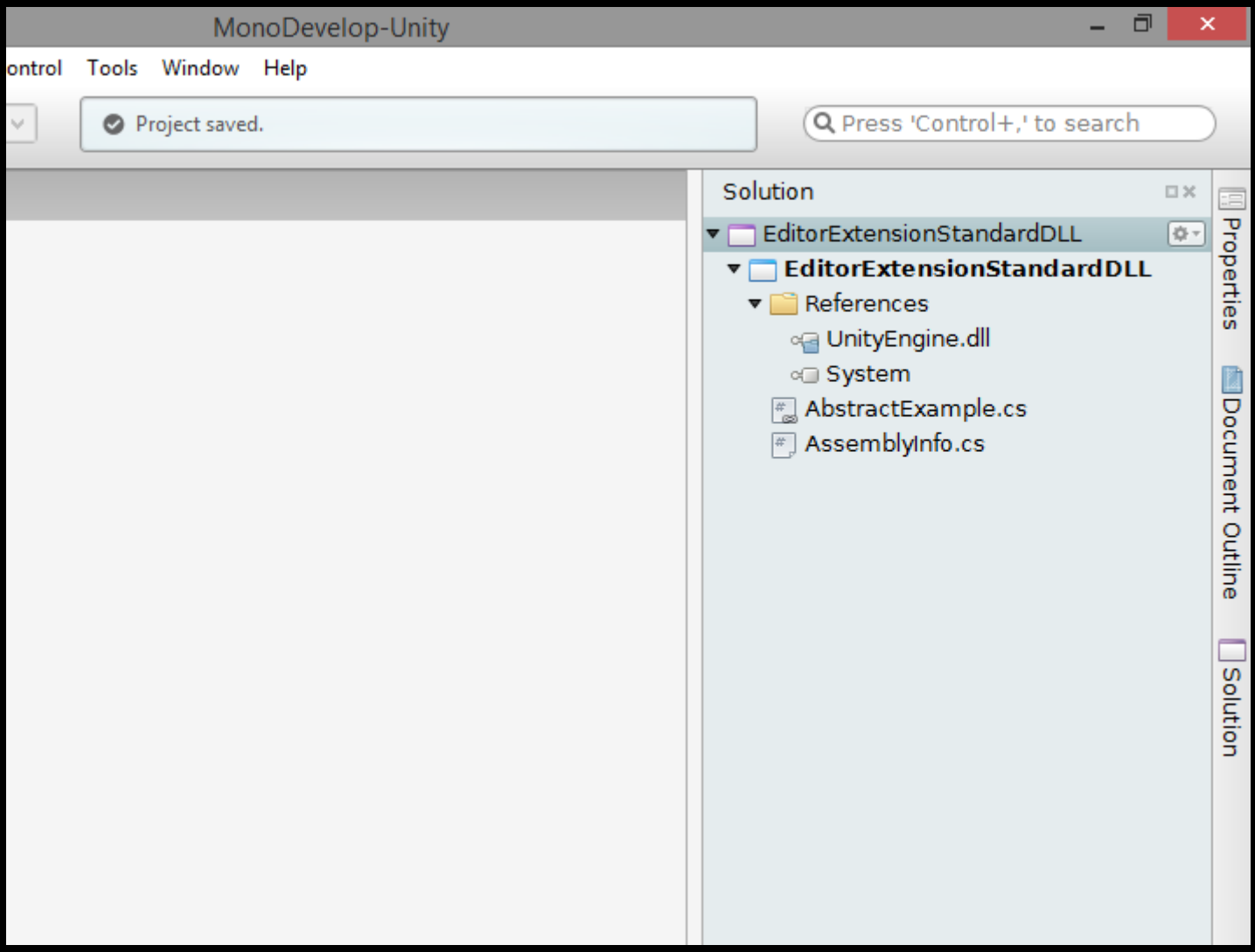
The next step is to link the source code to this solution. Click the small gear and click *Add -> Add files...*



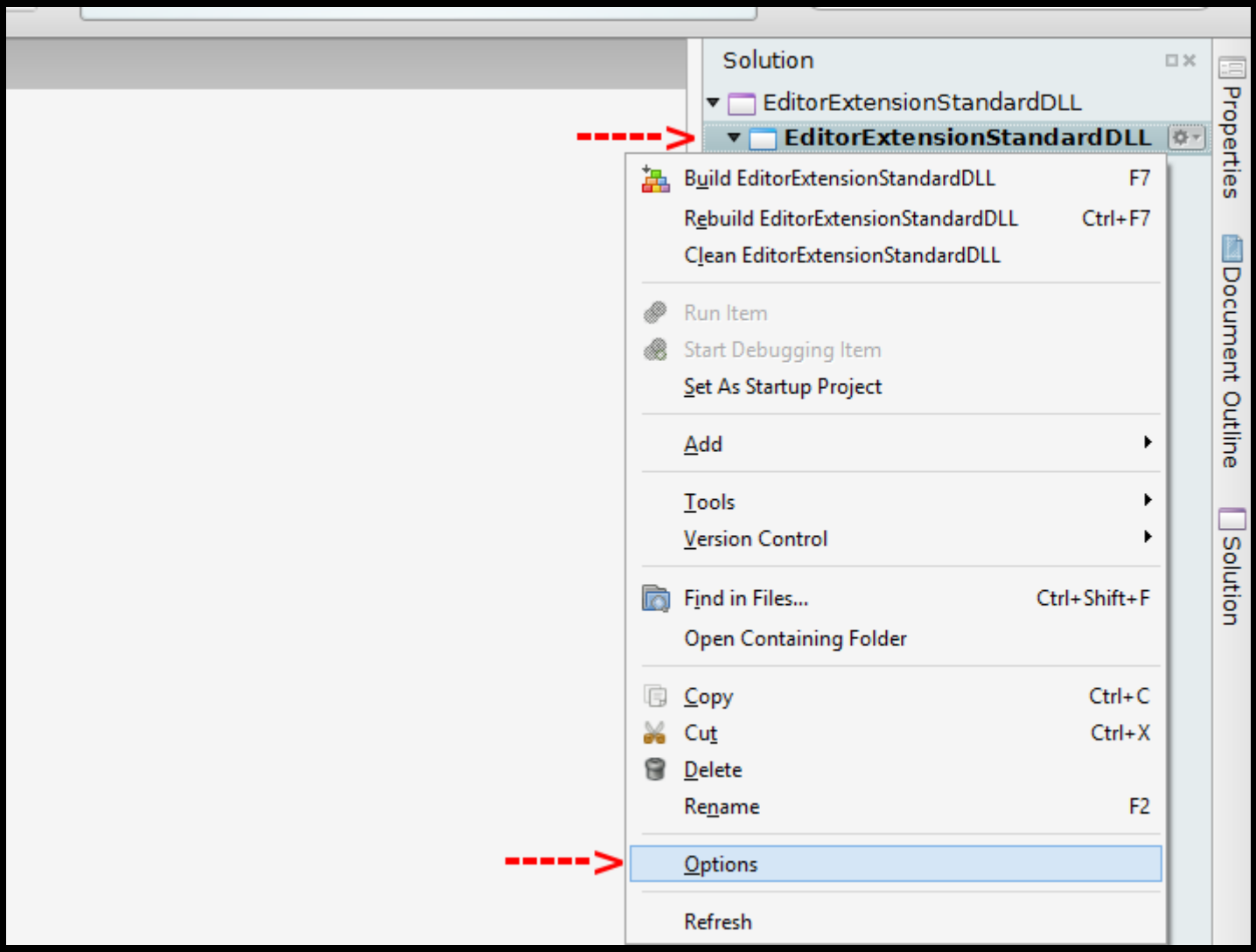
Browse to the Unity example project and select and open the file *AbstractExample.cs* in the Scripts folder.



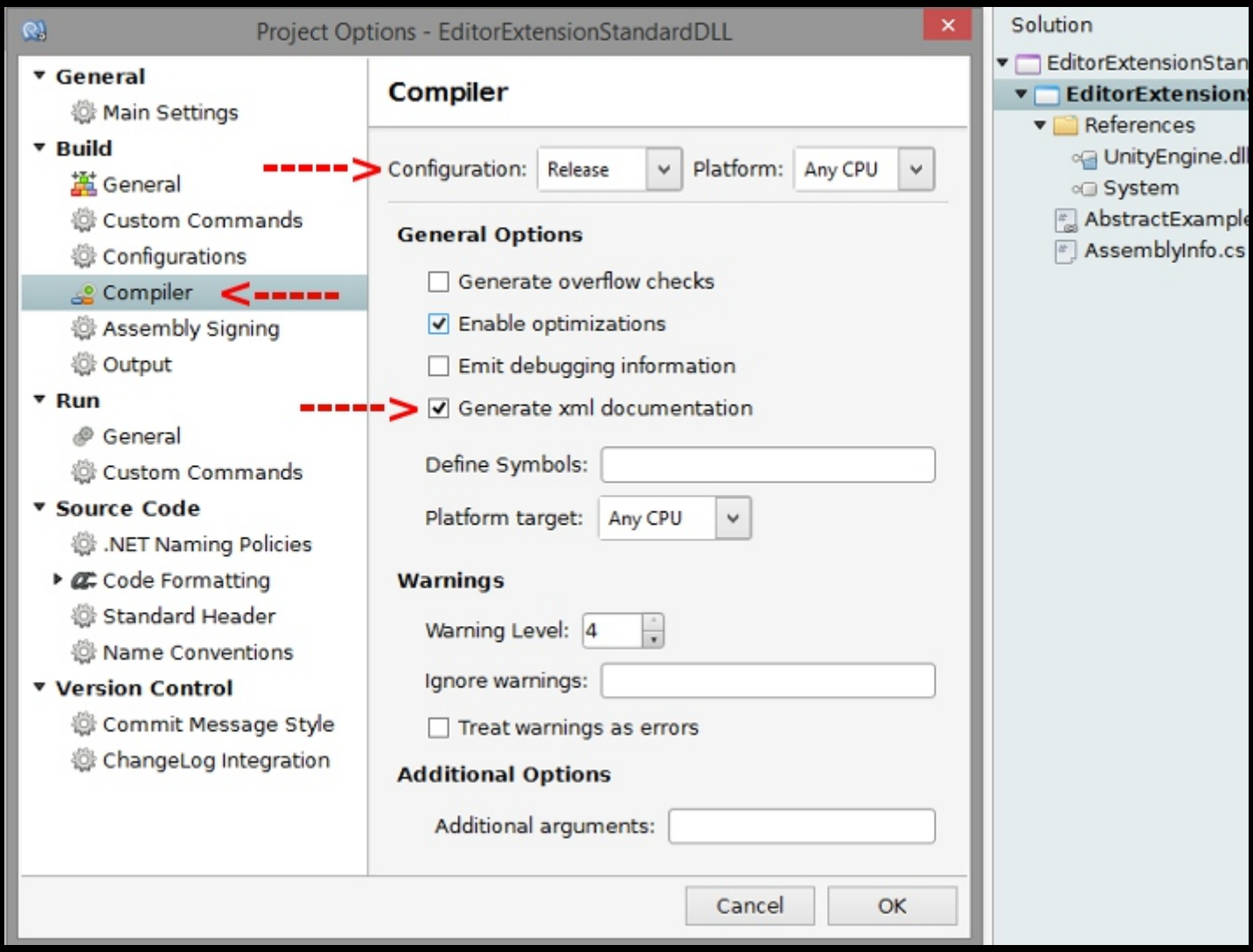
Select *Add a link* that the file is automatically updated here when you make changes in the Unity project.



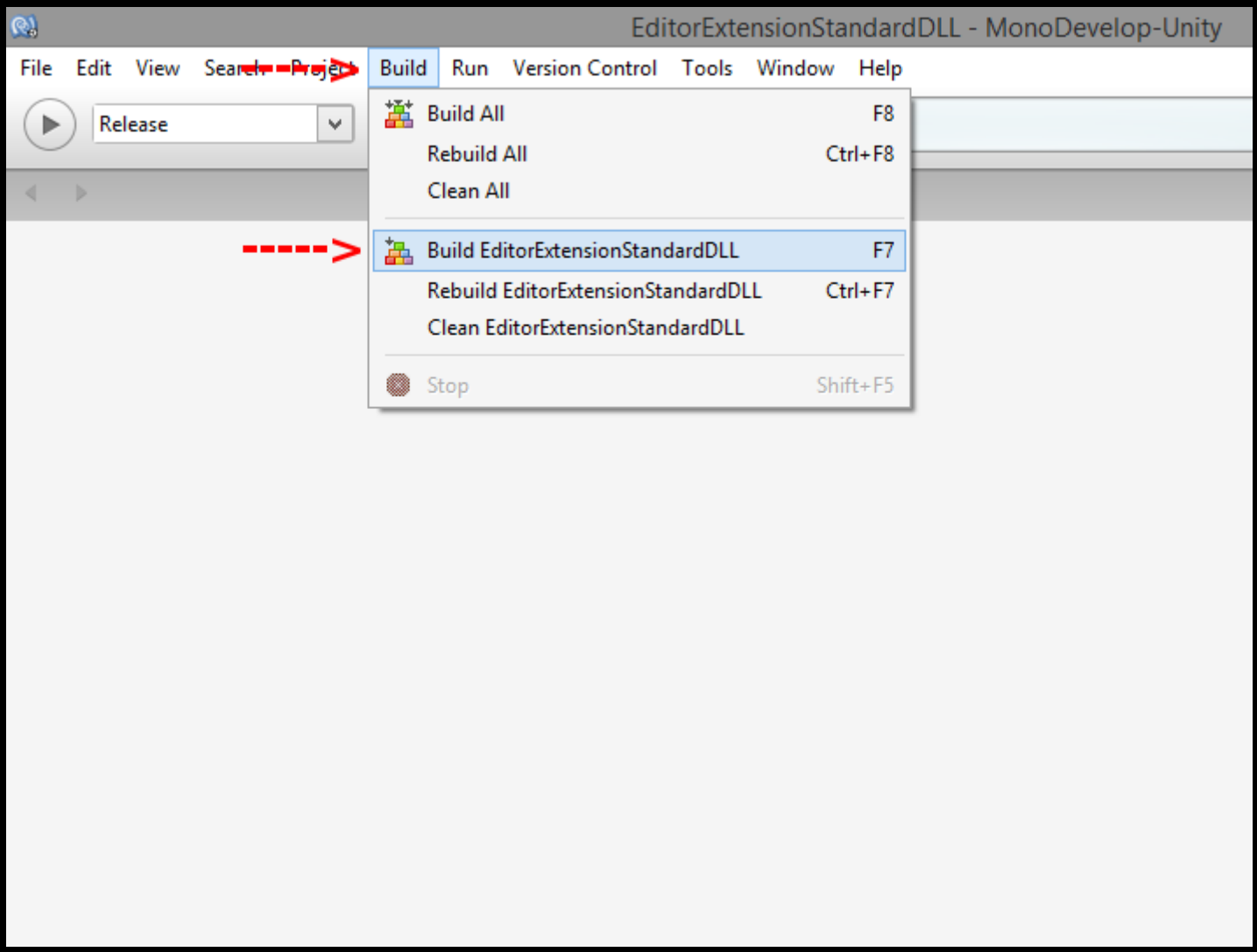
The Solution window should look like this.



To get tooltips it is necessary to change the compiler settings. Click the small gear and open the Options.



In the *Project Options* window on the left side select *Build* -> *Compiler*. Make sure the *Configuration* is set to *Release*. Enable *Generate XML documentation* to generate the XML file with the documentation. MonoDevelop will use it to load the intellisense text. Then click the OK button.



Now let's build the .dll. In the menu bar click *Build -> Build EditorExtensionStandardDLL*. Ignore the two warnings because the two members without XML comment will not show up in the intellisense at all. (Compare chapter 9.)

The build *EditorExtensionStandardDLL.dll* is saved into the MonoDevelop project folder under *bin -> Release* together with the *EditorExtensionStandardDLL.xml* file. Leave it there.. for now.

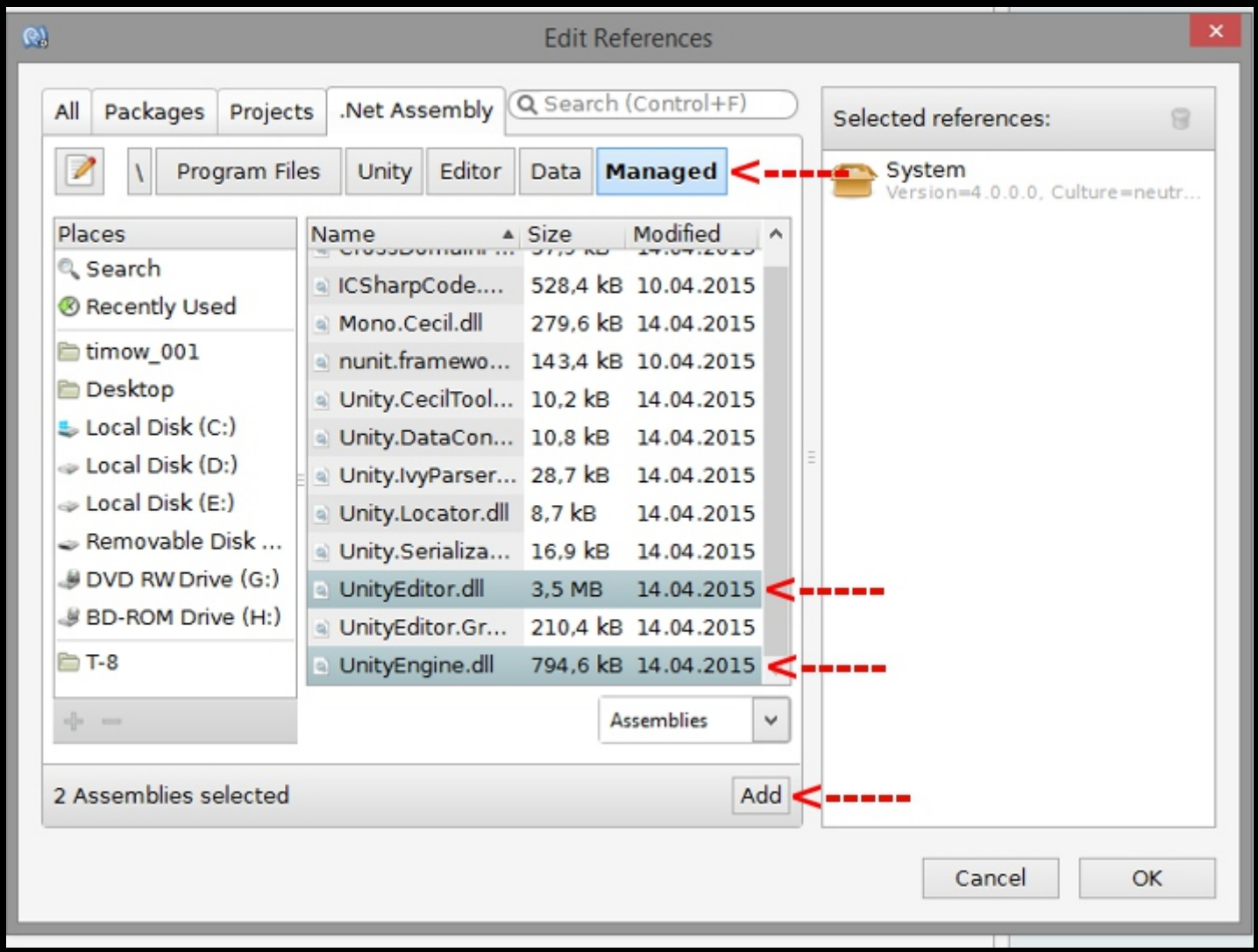
⇑ table of contents ⇑

11.2. UnityEditor namespace

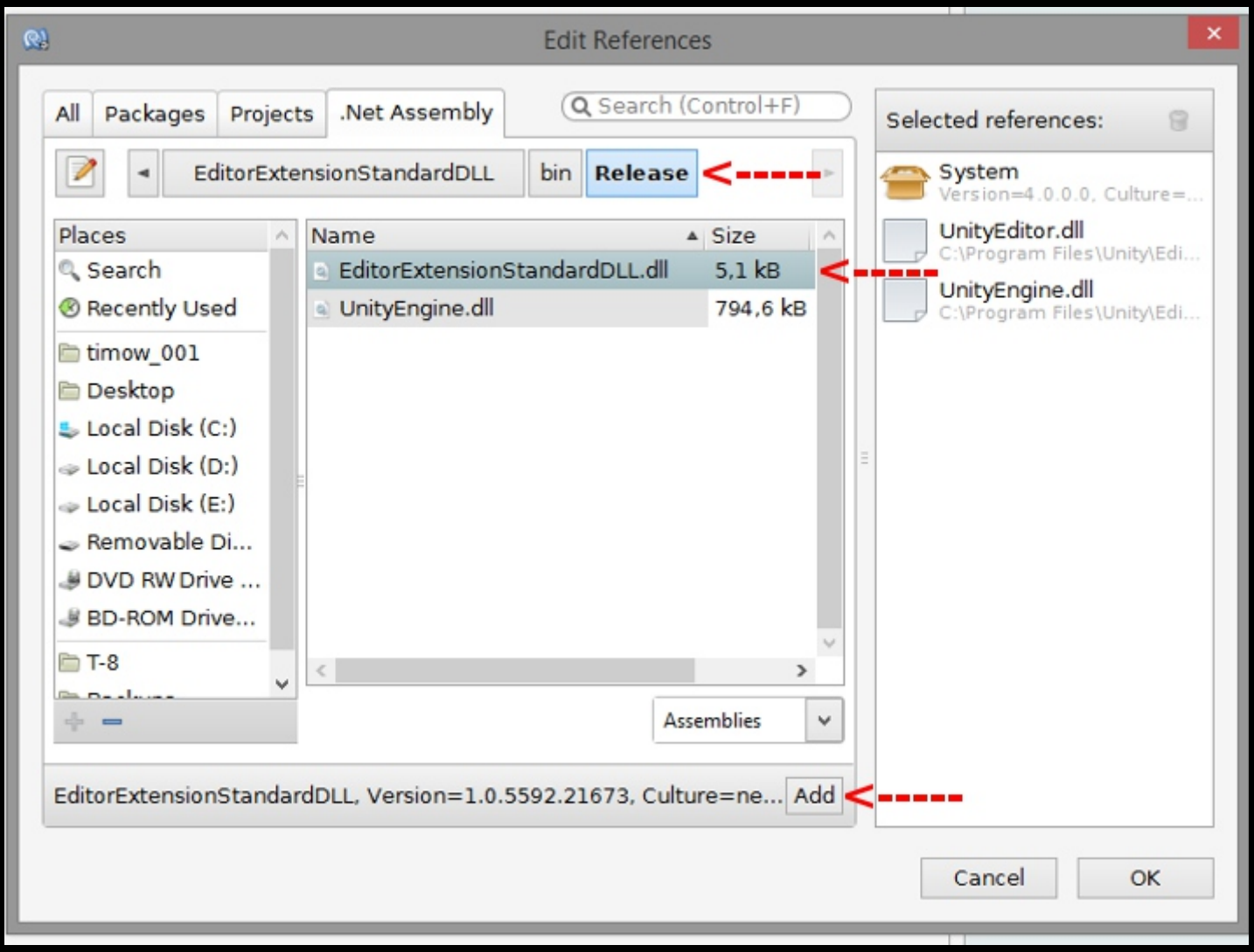
Create an new project in MonoDevelop and build the Editor script *AbstractExampleEditor.cs* as .dll. This works nearly exactly like to build the .dll of the standard script.

Choose *EditorExtensionEditorDLL* as name for this solution.

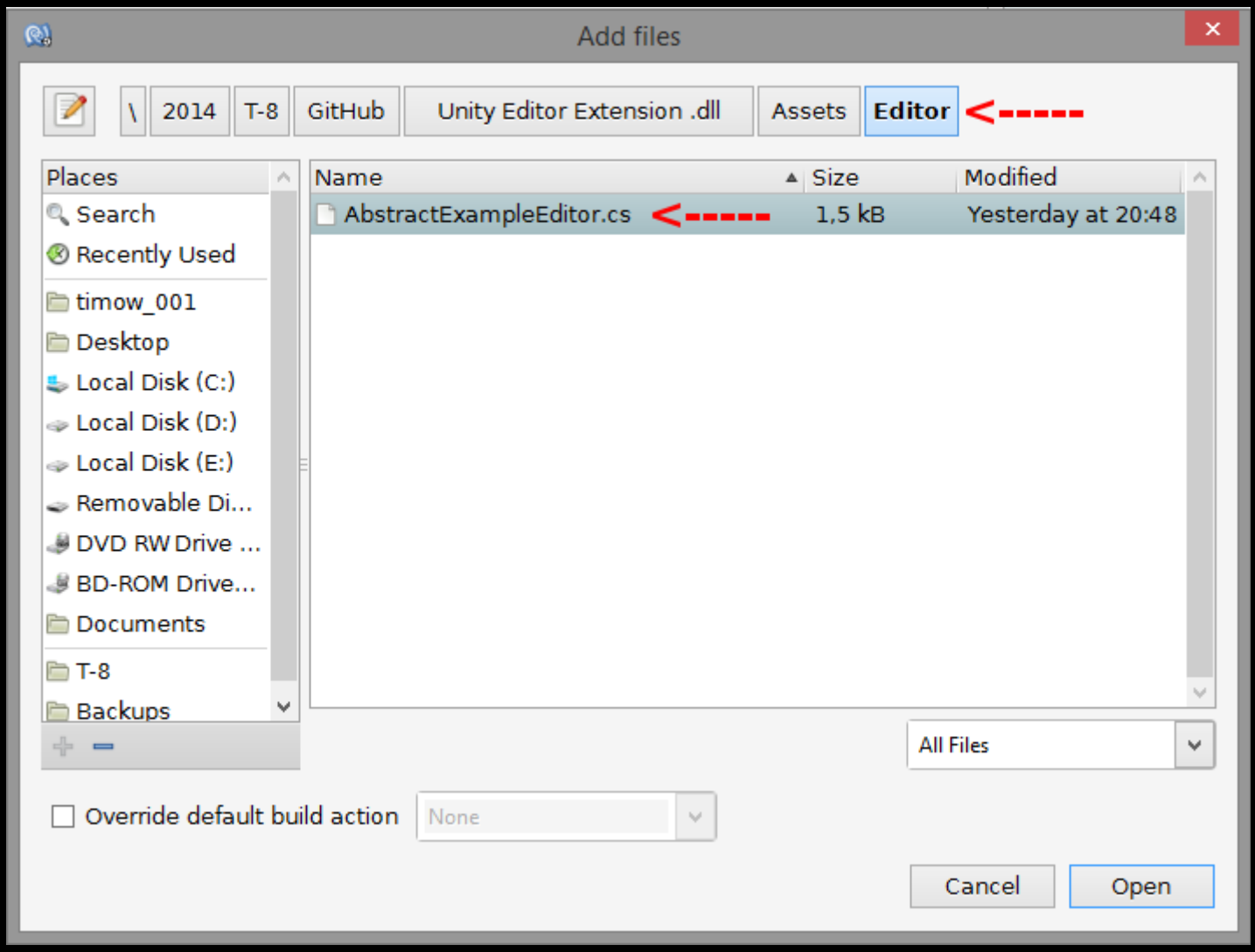
Follow the **screenshots** below, *that show only the differences to the standard script .dll creation process*, and the **descriptions** how to do it.



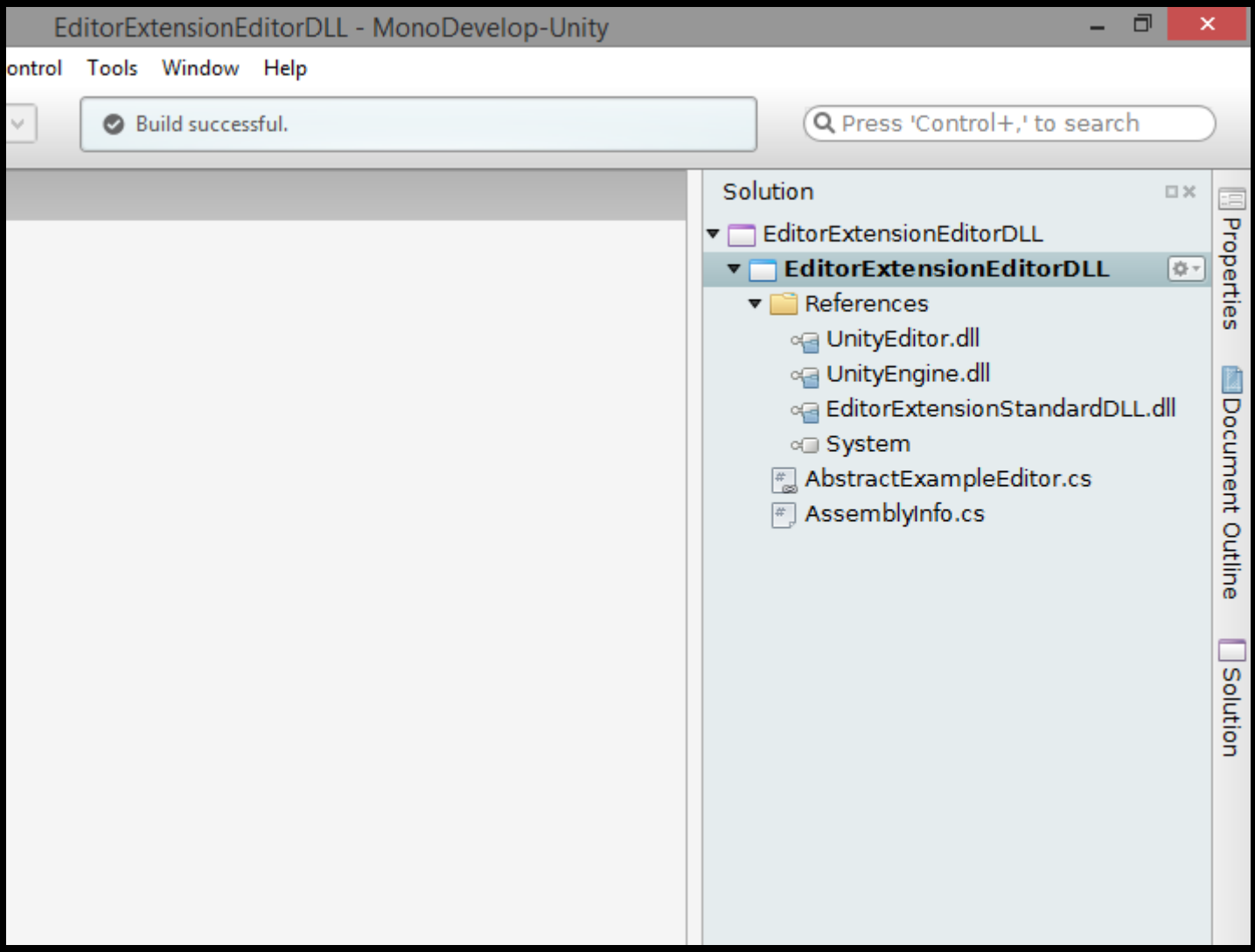
Also add a reference to the *UnityEditor.dll*.



And add a reference to the .dll of the standard script that was build in chapter 11.1..



Add a link to the file *AbstractExampleEditor.cs* instead of *AbstractExample.cs* to create a .dll of the file *AbstractExampleEditor.cs*.



The solution window should look like this now.

You don't need to enable the *Generate XML documentation* option in the *Project Options* window because there was no documentation added to the file *AbstractExampleEditor.cs*.
There shouldn't be any warning when you build the .dll.

⇑ table of contents ⇑

---

12. Test the .dlls / .xml in a new Unity project

1. In Unity create a new project (*menu bar -> File -> New Project...*) and name it e.g. *EditorExtensionDLLTest*.
2. In the Project window create a folder called "Scripts" and another called "Editor".
3. Open the Windows File Explorer and navigate to the MonoDevelop project of the standard script (*..\EditorExtensionStandardDLL\bin\Release*) and drag & drop the files *EditorExtensionStandardDLL.dll* and *EditorExtensionStandardDLL.xml* in Unity's Project window into the Scripts folder.
4. Use the Windows File Explorer to navigate to the MonoDevelop project of the Editor script (*..\EditorExtensionEditorDLL\bin\Release*) and drag & drop the file *EditorExtensionEditorDLL.dll* in Unity's Project window into the Editor folder.
5. Add a new empty gameobject to the Hierarchy window with *menu bar -> GameObject -> Create Empty.*
6. In the Project window navigate to the Scripts folder and open the *EditorExtensionStandardDLL.dll* by clicking the small triangle to the left of the .dll. A file called *AbstractExample* will becomes visible.
7. Drag & drop the AbstractExample file to the empty GameObject in the Hierarchy window.
8. Add a cube to the Hierarchy window with *menu bar -> 3D Object -> Cube*.

Now you have the same scene setup like in the Unity example project with the source code files.

Note: Unity's new Plugin Inspector automatically detects the .dll correct. It's made for all platforms because it only contains "Unity code".

⇑ table of contents ⇑

---

13. Create a build application

1. Attach the Cube gameobject to the GameObject gameobject to add the three components.
2. Check the Cube gameobject that all three components are attached by the .dll.
3. In the menu bar click File -> Build & Run.
4. Set the Platform to "PC, Mac and Linux Standalone".
   (Btw. -- Aren't Mac OS X and Linux computers "personal computers" (PCs for short) as well ?
5. Click the "Build & Run" button and select a location to save the build and name it.
6. Click "Play!". (Press ALT + F4 to quit the application.)
7. The cube doesn't move so the .dll deletes the components when the application is started.

Note: When you have a look at the log file (output_log.txt) of the build application (located in the application folder) you'll notice that the number of attached BoxColliders seems to remain one!
This happens because in the build application

```
GameObject.Destroy( targetsCollider );
```

is used, so the deletion is delayed slightly until after the current Update loop.

⇑ table of contents ⇑

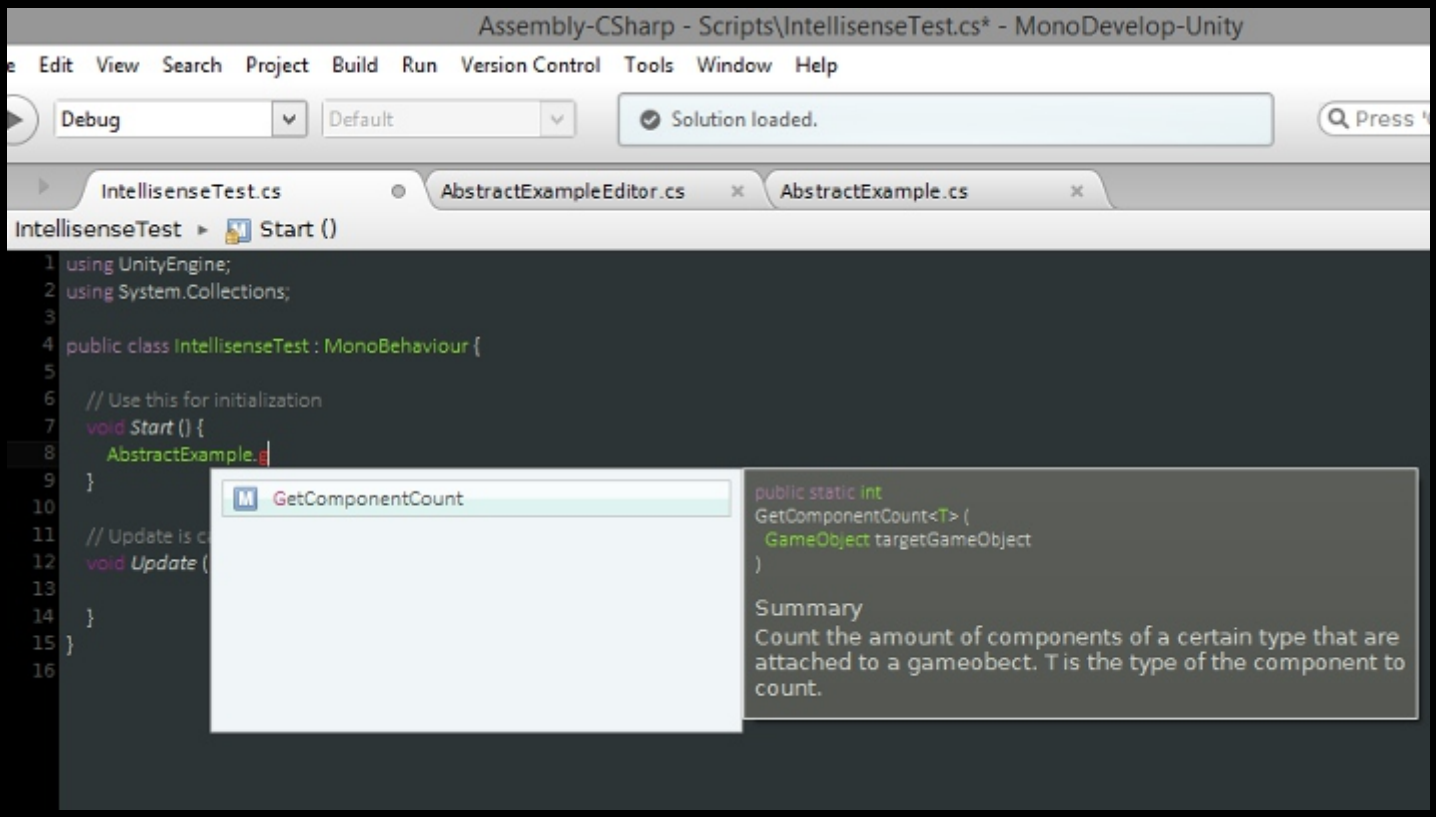
---

14. Test the intellisense tooltips

To test the intellisense tooltips in MonoDevelop just use the Unity project from chapter 13. and create new C# script (or use the one included in the *abstractexample.unitypackage*) in the *Scripts* folder and type

AbstractExample.g

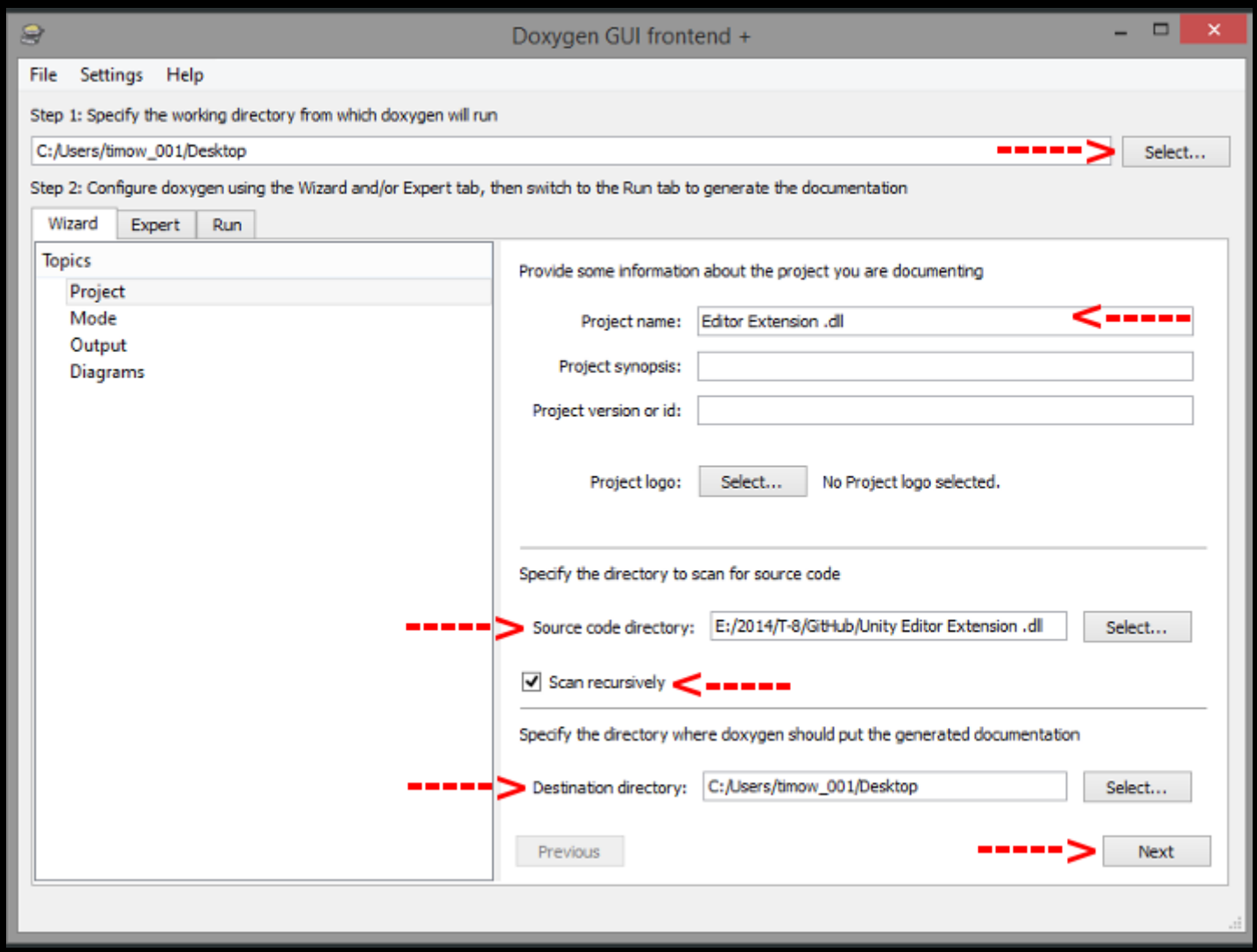
inside the method *Start* and you can see the description of the method *GetComponentCount*.



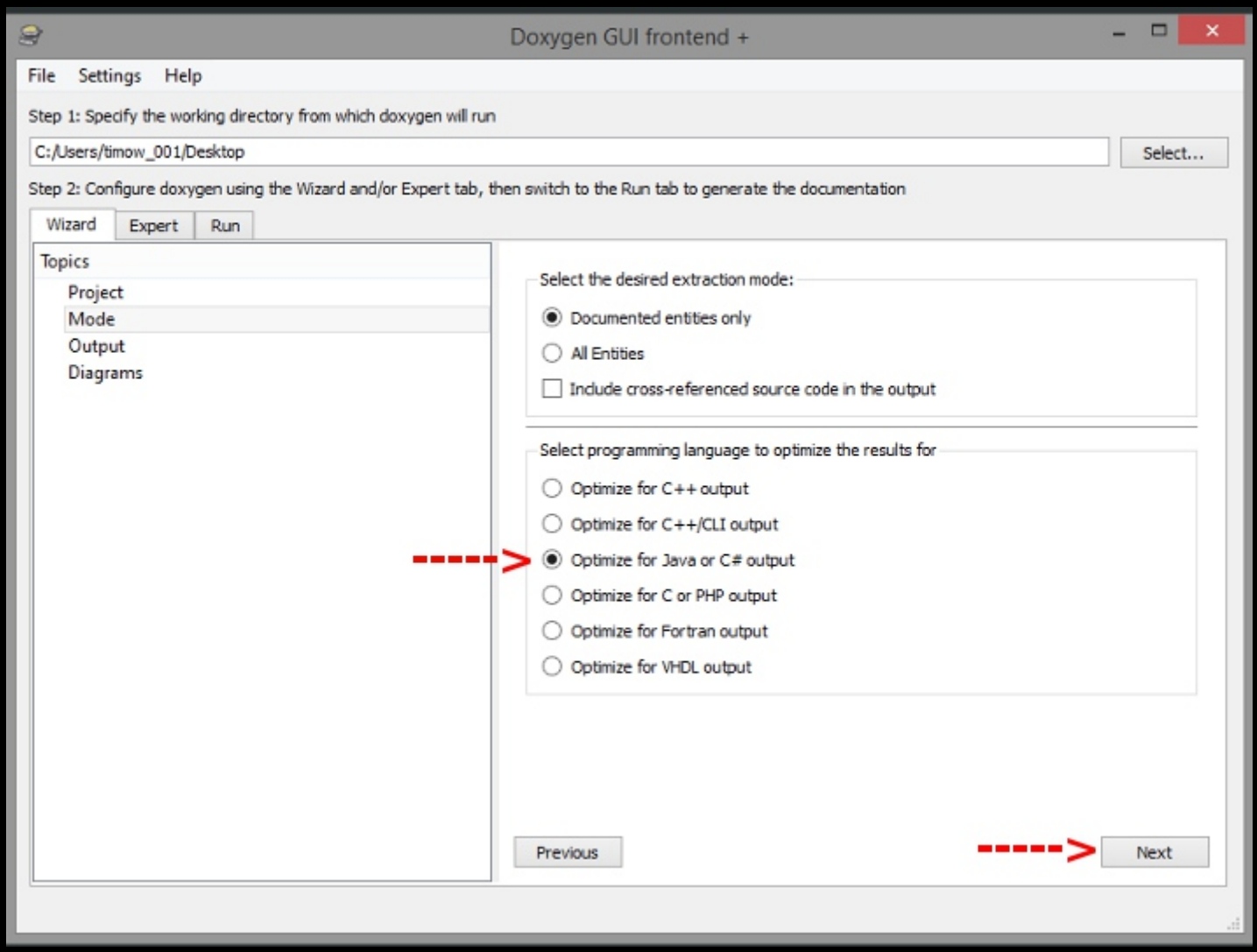
⇑ table of contents ⇑

15. Automatically generate a script documentation

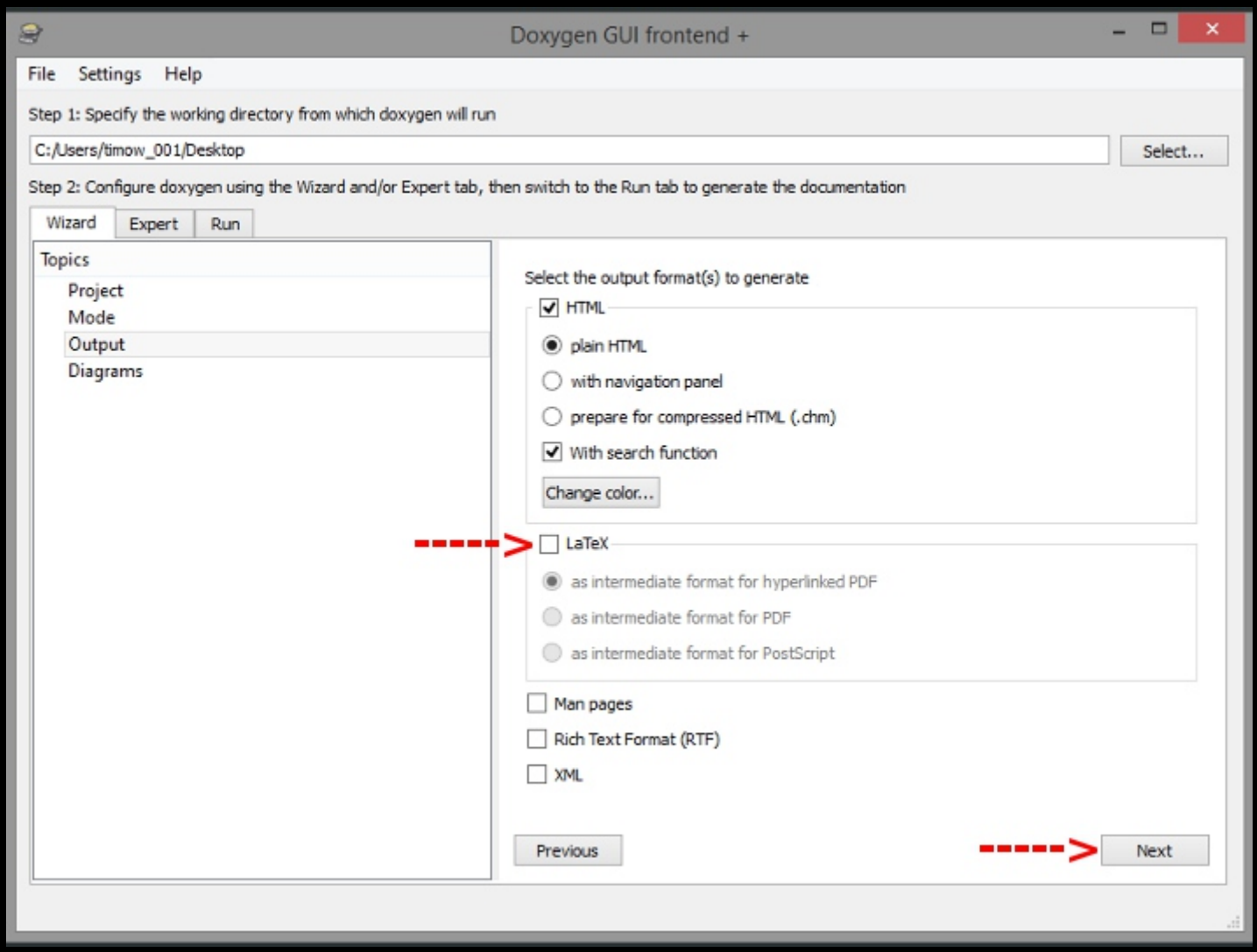
Open the *DoxyWizard* application.
Let's use the example project you have downloaded here.
Then follow the screenshots and their **descriptions** below how to use *DoxyGen*.



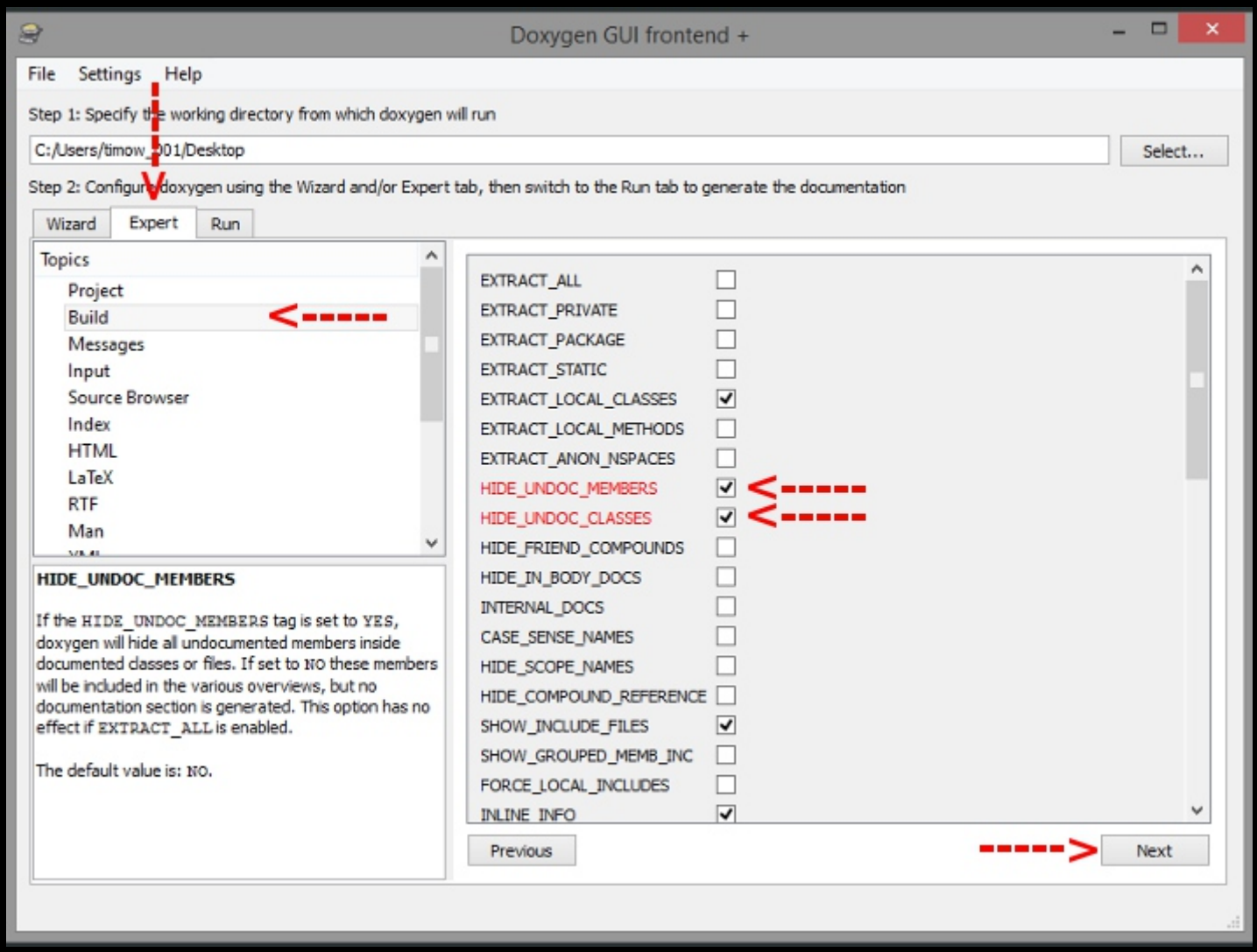
In the *Wizard* tab fill out the *Project* Topic and make sure you enable recursive scanning.



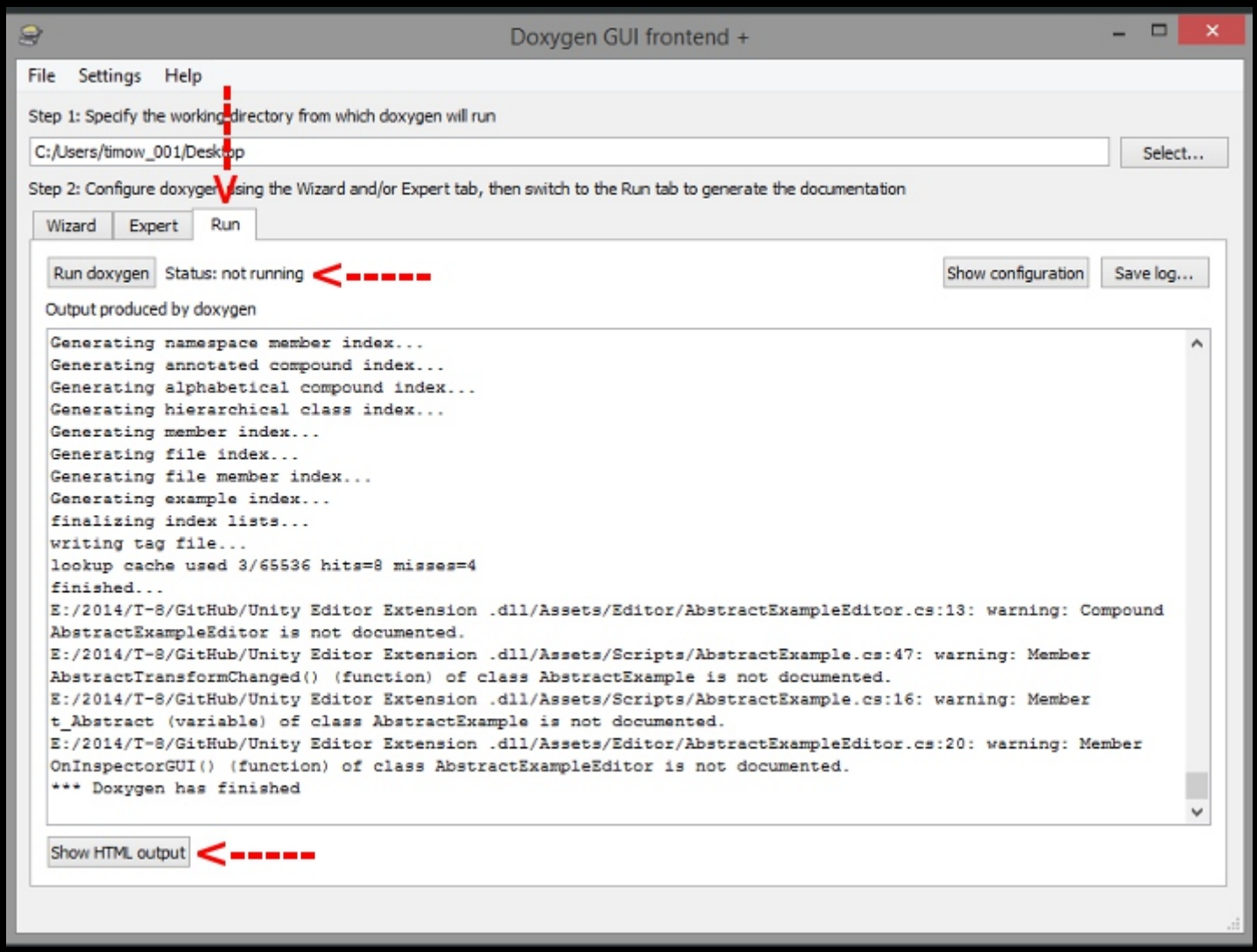
In the Mode *Topic* select *C# output*.



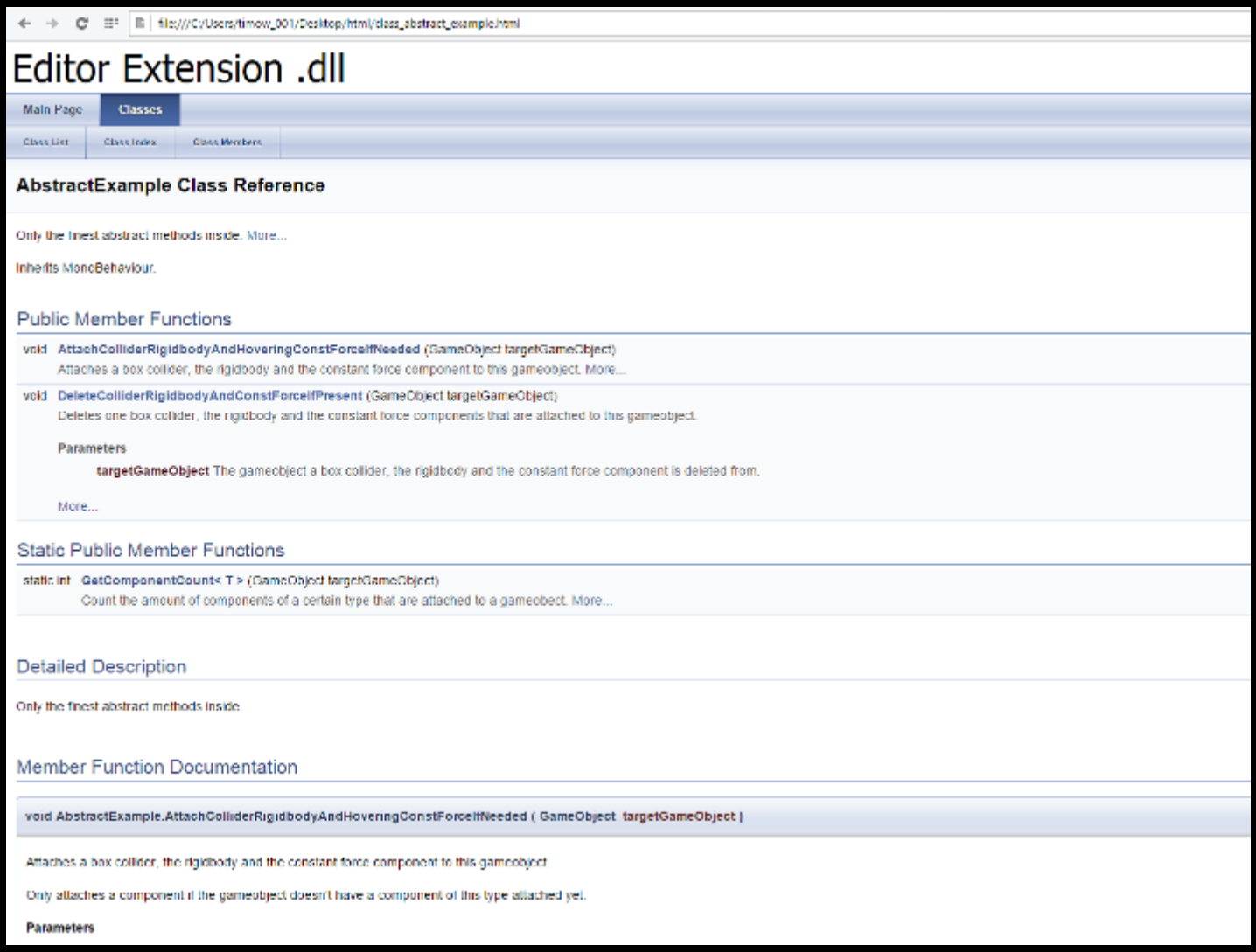
In the *Output* Topic disable *LaTeX*. Don't change anything in the *Diagrams* Topic.



Switch to the *Expert* tab and select the *Build* Topic. Enable the *HIDE_UNDOC_MEMBERS* and *HIDE_UNDOC_CLASSES* flags that all classes and members without XML comment are not included into the documentation.



Change to the *Run* tab and click the *Run doxygen* button. When *DoxyGen* has finished click the *Show HTML output* button.



The result.

Congratulations! You have finished this tutorial successfully. :)

⇑ table of contents ⇑
