

Extensibility Tools User Guide

for DesignBuilder v7.1

Overview

DesignBuilder may be extended via scripts or plugins. Each system is targeted at a different use-case and different levels of complexity. Both systems may be used at the same time, and multiple active plugins and scripts are supported.

Scripts are intended to hook into certain stages of a DesignBuilder simulation in order to modify or report on the process. Plugins are a superset of Scripts, providing extra functionality such as the ability to create a custom sub-menu in DesignBuilder's main-menu, which may allow for ad-hoc processing of DesignBuilder data.

Scripts can be written in the C# or Python programming languages. Plugins are written in the C# programming language.

The DesignBuilder API

DB.Api.Environment

For C#, DesignBuilder exposes its API to both extensibility systems by setting the `ApiEnvironment` property on the `IProperties` interface, which `IScript` and `IPlugin` inherit. The `Api.Environment` object is the root API object, which can be used to access the rest of the API.

For Python, the global variable `api_environment` is set.

During start-up, DesignBuilder instantiates a single `DB.Api.Environment` object. This singleton object is used by all active scripts and plugins, and as such, a script or plugin cannot rely on the state of the system at any given time. For example, suppose a script modifies the system at point A in a simulation. The script is then called again at point B and may expect the state of the system to be as it was at point A. However, it is possible that another script or plugin (or even DesignBuilder itself) also modified the system between point A and B. We therefore suggest that no assertions are made about the state of the system at any given point.

`DB.Api.Environment` is partitioned into several major areas, including:

- CFD operations, accessible via `DB.Api.Environment.CfdOperations`, which exposes functionality relating to DesignBuilder's CFD system
- Event subscriptions, accessible via `DB.Api.Environment.Events`, which allow for scripts and plugins to be notified of certain changes to the system
- HVAC operations, accessible via `DB.Api.Environment.HvacOperations`, which exposes functionality relating to DesignBuilder's HVAC system
- Site operations, accessible via `DB.Api.Environment.Site`, which exposes functionality relating to the active site. Functionality relating to Buildings, Blocks, Zones, Surfaces, etc can be accessed from the `Site` object
- Environment operations, accessible directly from `DB.Api.Environment`, which exposes a lot of miscellaneous functionality such as undo/redo, navigation, etc

05/07/22

Object Attributes

DesignBuilder uses object attributes to store much of the data for any given object (such as a building, block, zone, etc). An attribute is simply a key-value pair of strings that can be accessed via the `GetAttribute` and `SetAttribute` methods on any object that supports attributes.

For example, to get the occupancy density for a zone you would use `DB.Api.Zone.GetAttribute("OccupancyValue")`. To set the occupancy density to 0.1 you would use `DB.Api.Zone.SetAttribute("OccupancyValue", (0.1).ToString())`.

Generally, an object can have a lot of attributes and it is impractical to list and describe each one. Nevertheless, to get full use of the API you will need to know an attribute's name, or key, so that you can access or change its value. To help with this, you can enable DesignBuilder to show an attribute's name in a tooltip in DesignBuilder's GUI. To do this, simply enable `Tools > Program options > Interface > Interface Style > Show attribute names in tooltips` (you will need to enable "Show Tooltips in Model data" first, if it is not already enabled).

With "Show attribute names in tooltips" enabled, you can hover over an attribute in DesignBuilder and the tooltip will display that attribute's name, which you can then use with `GetAttribute` and `SetAttribute`. For example:



DesignBuilder Hooks

During a DesignBuilder simulation, DesignBuilder can pass control to a script or plugin at certain points in the process. These points are called hook points, or hooks, or call points. Both scripts and plugins have access to the same hooks and any implemented hooks are called for each active script and plugin. That is to say, if you have 2 active plugins and 3 active scripts, all of which implement the same hook, DesignBuilder will pass control to all 5 hooks in the following order: All scripts in the order they appear in the Script Manager, followed by all plugins in alphabetical order of the plugin's assembly name.

The following hooks are available, with each hook point (in a category) being called in the order they are written:

Simulation:

- **BeforeEnergyIdfGeneration (C#)/before_energy_idf_generation (Python)** is called immediately before the EnergyPlus IDF file is created
- **BeforeEnergySimulation (C#)/before_energy_simulation (Python)** is called immediately before an EnergyPlus simulation starts
- **AfterEnergySimulation (C#)/after_energy_simulation (Python)** is called immediately after an EnergyPlus simulation has finished
- **BeforeHeatingIdfGeneration (C#)/before_heating_idf_generation (Python)** is called immediately before the EnergyPlus IDF file is created for a Heating design simulation
- **BeforeHeatingSimulation (C#)/before_heating_simulation (Python)** is called immediately before a Heating design simulation starts

05/07/22

- **AfterHeatingSimulation (C#)/after_heating_simulation (Python)** is called immediately after a Heating design simulation has finished
- **BeforeCoolingIdfGeneration (C#)/before_cooling_idf_generation (Python)** is called immediately before the EnergyPlus IDF file is created for a Cooling design simulation
- **BeforeCoolingSimulation (C#)/before_cooling_simulation (Python)** is called immediately before a Cooling design simulation starts
- **AfterCoolingSimulation (C#)/after_cooling_simulation (Python)** is called immediately after a Cooling design simulation has finished

Optimisation:

- **BeforeOptimisationStudy (C#)/before_optimization (Python)** is called immediately before an optimisation study starts
- **AfterOptimisationStudy (C#)/after_optimisation (Python)** is called immediately after an optimisation study has finished
- **OnDesignVariableChanged (C#)/on_design_variable_changed (Python)** is called when an optimization design variable of type “Custom Script” is changed before an optimisation simulation. The id and value of the variable are passed to this hook point. The Id’s start at 10000 and are ordered as they appear in the Design Variables window
- Note that optimisation will run multiple simulations and for each simulation the **BeforeEnergyIdfGeneration** and **BeforeEnergySimulation** hooks are called

CFD

- **BeforeCfdSimulation (C#)/before_cfd_simulation (Python)** is called immediately before a CFD simulation starts
- **AfterCfdSimulation (C#)/after_cfd_simulation (Python)** is called immediately after a CFD simulation has finished

Daylight Simulation:

- **BeforeDaylightSimulation (C#)/before_daylight_simulation (Python)** is called immediately before a daylighting simulation starts
- **AfterDaylightSimulation (C#)/after_daylight_simulation (Python)** is called immediately after a daylighting simulation has finished

Cost and Carbon

- **BeforeCostAndCarbon (C#)/before_cost_and_carbon (Python)** is called immediately before generating a cost and carbon report
- **AfterCostAndCarbon (C#)/after_cost_and_carbon (Python)** is called immediately after generating a cost and carbon report

Misc:

- **AtCommandLine (C#)/at_command_line (Python)** is called when DesignBuilder is run from the command-line with the argument /process=ExternalCommand_<arg>, where arg is an optional string that is passed to the hook point
- **ScreenChanged (C# plugin v2 only)** is called when DesignBuilder changes screen. An enum corresponding to the new screen is passed as an argument.

05/07/22

- **ModelLoaded (C# plugin v2 only)** is called after a model (dsb file) has been opened or a new model has been created.
- **ModelUnloaded (C# plugin v2 only)** is called after a model has been unloaded (the file has been closed).

Note that before each hook point is called¹, for C#, the ActiveBuilding property on IProperties will be set to the instance of the building that the hook point is called by. For Python, the active_building global variable will be set. In the case that a building is not currently selected, ActiveBuilding/active_building will be null.

See section [Plugin API Versions](#) for more information on the available plugin versions.

Extending DesignBuilder With Scripts

What is a Script?

A script is a means for users to extend a DesignBuilder simulation by providing custom functionality via the C# or Python programming languages. A script may modify or report on DesignBuilder's internal data at each of the hook points described in section *DesignBuilder Hooks*.

Unlike a plugin, a script may only interact with DesignBuilder at the defined hook points and only exist during the lifetime of a simulation.

Scripts are saved with a model in the .dsb file, unlike plugins, which are separate from a model.

Writing a C# Script


To extend DesignBuilder with a script using the C# programming language:

1. Open DesignBuilder, open or create a model and navigate to Tools > Scripts. This will open the *Script Manager*.
2. In the Script Manager, ensure the *Enable scripts* option is checked and click on the *Script* sub-item, which will open the *Select the Script* dialog. Here, you can see a list of scripts that are supplied with DesignBuilder. You will also notice that DesignBuilder supports three types of scripts; CS-Script, EMS, and Python-Script

EMS scripts do not support the DesignBuilder API and work differently from CS-Scripts and Python-Scripts. Documentation for EMS scripts can be found [here](#).

Python-Scripts are not fully supported in DesignBuilder v6.0. Access to the API will be provided for Python scripts in v6.2.

3. In the Select the Script dialog, open the CS-Script tree folder and select any existing script. From here you may create a new script, or copy or edit the selected script by using the icons at the bottom of the dialog.

Press the  button to create a new script.

¹ Except for ModelLoaded and ModelUnloaded

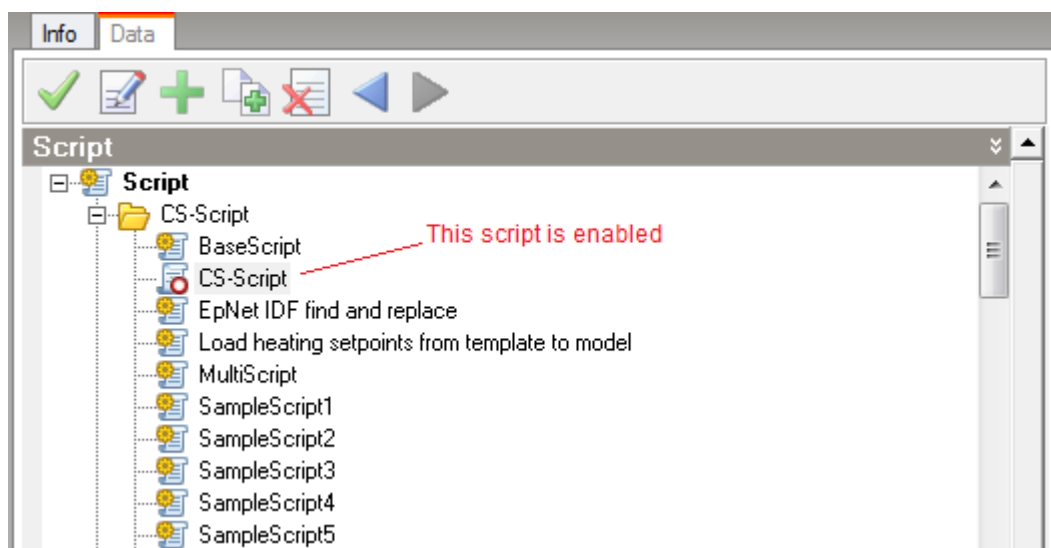
05/07/22

- The *Edit Script* window provides a basic text editor into which you can write your code. It is suggested, however, that an editor more suited to C# development is used to write your script from which you can copy the code into the Edit Script text box.

Generally, a script will consist of a single class. You may choose to scope your class to a namespace, but it is not necessary. Copy the following code into the Edit Script text box

```
public class ExampleScript
{
}
```

- Click on the *Compile script* button in the Info pane and you should see the message “Script compiled with no errors.” Press the OK button on the message box. The Compile script button is a convenient way of checking your script is syntactically correct before we run a simulation.
- In order to enable this script to run during a simulation, make sure the *Enable program* checkbox in the Edit Script window is checked. It can be found just above the Edit Script textbox under the *Script* header.
- Press the OK button on the Edit Script window. This will take us back to the Script Manager window. You should now see your script listed in the tree view of the data pane with a red circle on its icon. This indicates that the script is enabled and will be run during a simulation.




- Press the OK button on the Script Manager window. Navigate to the Simulation tab in DesignBuilder and run a new simulation. The simulation should complete as normal.

This is the basic process required to create, write, and enable a script in DesignBuilder. The script we wrote in step 4, however, doesn’t do a whole lot. In order for the script to be more useful it needs to hook in to DesignBuilder through various entry points.

05/07/22

9. Navigate back to the Script Manager and click on the script created in step 4. To edit the

script, click the *Edit highlighted item* button , which will take you back to the Edit Script window, where you can alter the script's code.

10. In order for DesignBuilder to pass control of a simulation to a script, a script must expose at least one of the hook points described in section *DesignBuilder Hooks*. Each hook point defines a different stage in a simulation. A script can expose as many hook points as it chooses.

From a C# point-of-view, a script must implement the `DB.Extensibility.Contracts.IScript` interface. This interface declares a method for each hook point that DesignBuilder can call.

Because it's likely that a script will not want to provide an implementation for every hook point, a base class (`DB.Extensibility.Contracts.ScriptBase`) is also provided, which implements default functionality for the `IScript` interface.

Add the following code to the existing script:

```
using DB.Extensibility.Contracts;

public class ExampleScript : ScriptBase, IScript
{
}
```

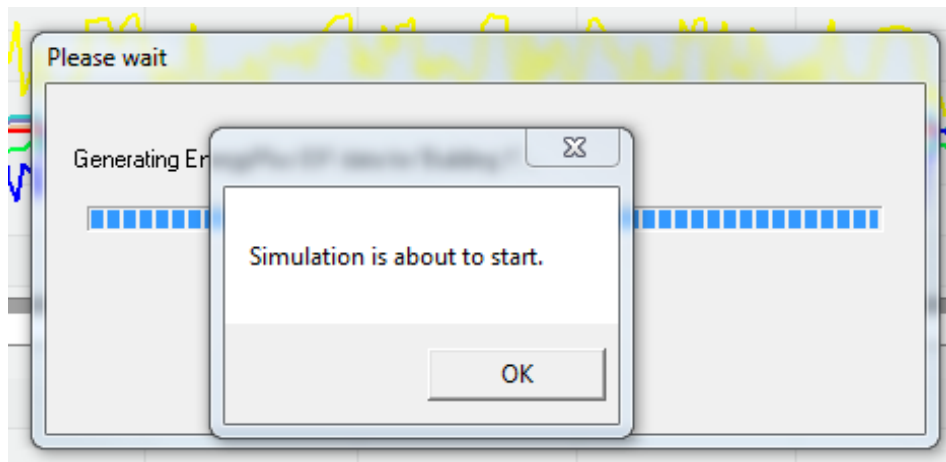
11. The script now does a bit more than the previous version, but not much. Technically, it exposes all hook points to DesignBuilder, but each is implemented with the default behaviour provided by `DB.Extensibility.Contracts.ScriptBase`. The default behaviour of `ScriptBase` is to do nothing. To remedy that, the script will have to override one of the methods. For the purposes of this example override the `BeforeSimulation` hook point as follows

```
using System.Windows.Forms;
using DB.Extensibility.Contracts;

public class ExampleScript : ScriptBase, IScript
{
    public override void BeforeEnergySimulation()
    {
        MessageBox.Show("Simulation is about to start.");
    }
}
```

12. Click the Compile script button to make sure the code is syntactically correct, then press the OK button on the Edit Script window. Now press the OK button on the Script Manager window and run a new simulation. After the Loading data and Generating IDF stages of simulation, a messages box should appear as depicted below

05/07/22



To continue with the simulation press the OK button on the message box.

13. The script is now a fully functioning script that displays a message before simulation starts. It's not hard to imagine that this script could be modified to open a more useful window that allows users to view, or modify certain options before continuing with the simulation.

For more ideas of what can be done with a script please examine the example scripts provided by default. They can be found in the Script Manager window.

Writing a Python Script

To extend DesignBuilder with a script using the Python programming language:

1. Open DesignBuilder, open or create a model and navigate to Tools > Scripts. This will open the *Script Manager*.
2. In the Script Manager, ensure the *Enable scripts* option is checked and click on the *Script* sub-item, which will open the *Select the Script* dialog. Here, you can see a list of scripts that are supplied with DesignBuilder. You will also notice that DesignBuilder supports three types of scripts; CS-Script, EMS, and Python-Script.
3. In the Select the Script dialog, open the Python-Script tree folder and select any existing script. From here you may create a new script, or copy or edit the selected script by using the icons at the bottom of the dialog.

Press the  button to create a new script.

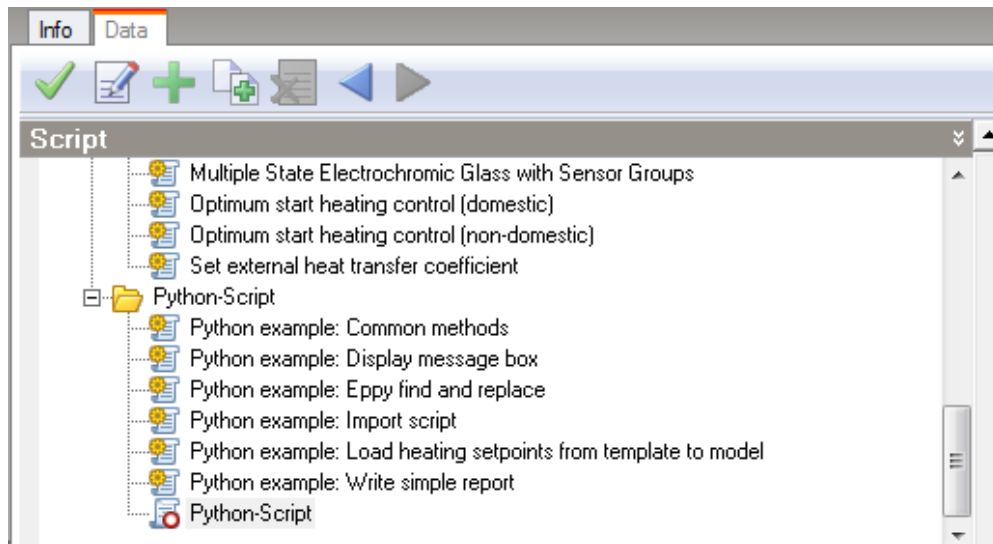
4. The *Edit Script* window provides a basic text editor into which you can write your code. It is suggested, however, that an editor more suited to Python development is used to write your script from which you can copy the code into the Edit Script text box.

In order to enable a script to run during a simulation, make sure the *Enable program* checkbox in the Edit Script window is checked. It can be found just above the Edit Script textbox under the *Script* header.

05/07/22

The *Compile script* button in the Info pane is a convenient way of checking your script is syntactically correct before we run a simulation.


5. Press the OK button on the Edit Script window. This will take us back to the Script Manager window. You should now see your script listed in the tree view of the data pane with a red circle on its icon. This indicates that the script is enabled and will be run during a simulation.



6. Press the OK button on the Script Manager window. Navigate to the Simulation tab in DesignBuilder and run a new simulation. The simulation should complete as normal.

This is the basic process required to create, write, and enable a script in DesignBuilder. Our script, however, is empty and doesn't do a whole lot. In order for the script to be more useful it needs to hook in to DesignBuilder through various entry points.

7. Navigate back to the Script Manager and click on the script created in step 4. To edit the

script, click the *Edit highlighted item* button , which will take you back to the Edit Script window, where you can alter the script's code.

8. In order for DesignBuilder to pass control of a simulation to a script, a script must expose at least one of the hook points described in section *DesignBuilder Hooks*. Each hook point defines a different stage in a simulation. A script can expose as many hook points as it chooses.

From a Python point-of-view, all that is required is for a script to define a function with the same name as one of the hook points.

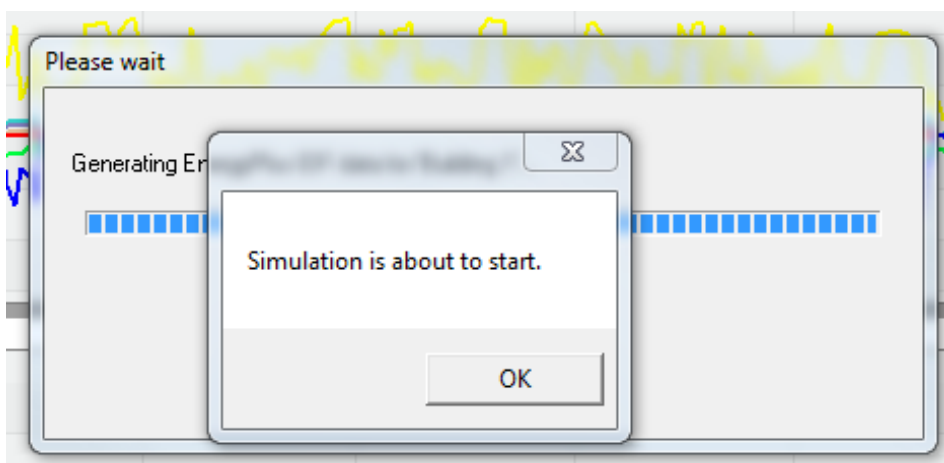
Add the following code to the existing script:

05/07/22

```
import ctypes

def before_energy_simulation():
    ctypes.windll.user32.MessageBoxW(
        0, "Simulation is about to start.", "Title", 0)
```

- Click the Compile script button to make sure the code is syntactically correct, then press the OK button on the Edit Script window. Now press the OK button on the Script Manager window and run a new simulation. After the Loading data and Generating IDF stages of simulation, a messages box should appear as depicted below



To continue with the simulation press the OK button on the message box.

- The script is now a fully functioning script that displays a message before simulation starts. It's not hard to imagine that this script could be modified to open a more useful window that allows users to view, or modify certain options before continuing with the simulation.

For more ideas of what can be done with a script please examine the example scripts provided by default. They can be found in the Script Manager window.

Importing Other Scripts Into a Script

A DesignBuilder script can import 1 or more other scripts before it is run. This is useful if you want to combine scripts that may expose different hook points as if it were a single script, or if you want to store utility classes/methods in some base script, which you can use in other scripts.

In order to import a C# script into another, simply use the directive `css_import <other script's name>` as a comment at the top of the importing script. For example, say "ScriptParent" intends to import functionality from "ScriptChild", ScriptParent would include `//css_import ScriptChild` at the top of its file (this can be before or after the using statements). For Python, use the `from/import` directives. For example, to import a method called "show_message" from "ScriptChild", ScriptParent would include `from ScriptChild import show_message`

It is important to note that **both the parent script and all child scripts must be enabled** in the Script Manager window.

05/07/22

The name of the script is the one that appears in the Script Manager window with the following alterations: Any spaces or instances of the characters '<' (less than), '>' (greater than), '\ ' (back slash), '/' (forward slash), '?' (question mark), ':' (colon), ';' (semi-colon), '"' (quotation mark), ',' (comma), '*' (asterisks) will be replaced by underscores.

For example, a script with the name "Example: My script", will need to be imported with the name "Example__My_script".

If more than one enabled script implements the same hook point, each one will be executed in alphabetical order of the class name that implements it (C#), or the filename that implements it (Python).

Extending DesignBuilder with Plugins

What is a Plugin?

A plugin provides a more complete way of extending DesignBuilder. Plugins are a superset of C# Scripts so they are able to take advantage of the same hook points that scripts can use with the addition of being able to add a menu to DesignBuilder's top-level menu (if they choose).

Unlike scripts, which are interpreted at runtime, plugins are pre-built assemblies that are loaded by DesignBuilder. This gives the developer more freedom than a script because they are able to reference other assemblies and nuget packages.

Plugins can be used:

- As a more complete scripting solution. By implementing only the hook points and not providing a menu structure, a plugin will not be visible to a user but will still be used by DesignBuilder during a simulation. This is useful if you want to reference third-party assemblies, or your own libraries in your solution
- To allow for ad-hoc calculations, reports, or modifications to DesignBuilder's data. Because plugins can create their own menu in DesignBuilder they do not rely on being called by the hook points like scripts do. Plugin operations can be called whenever a menu item is pressed
- A more complete extension of DesignBuilder. Because plugins are virtually unrestricted they may provide a complete GUI-based solution. Whilst they cannot integrate with DesignBuilder's GUI, they can launch their own windows and work in a very similar manner to DesignBuilder's own Model Data Grid View (which can be found in Tools > Model data grid view... when the Edit tab is active)

One downside of the plugin system, however, is that **plugins are not saved with the .dsb file**. If you therefore want to share a model and a plugin is vital to your work, you will also need to share the plugin assemblies. Scripts on the other hand are saved with the model.

Writing a Plugin

DesignBuilder loads plugins at start-up from .NET assemblies. Therefore, to develop a plugin you must create a new .NET class library. It is assumed that you already know how to do this as it is out of the scope of DesignBuilder's help.

The example provided here assumes a .NET class library using the C# programming language.

To extend DesignBuilder with a plugin:

05/07/22

1. Communication between DesignBuilder and a plugin is achieved through the DesignBuilder API and a plugin-specific interface. It is therefore important that your project assembly references the `DB.Api` and `DB.Extensibility.Contracts` assemblies.

The `DB.Api` assembly can be found in the “`Components/DB.Api`” directory of DesignBuilder’s installation directory. `DB.Extensibility.Contracts` can be found in the “`Components/DB.Extensibility`” directory.

2. DesignBuilder uses Microsoft’s Managed Extensibility Framework (MEF) to locate and load plugins. Your project assembly must therefore also reference `System.ComponentModel.Composition`
3. Now that your project assembly has the correct references it’s time to create a class that DesignBuilder can interact with. As mentioned, DesignBuilder communicates with a plugin via a plugin-specific interface, namely `DB.Extensibility.Contracts.IPlugin`.

Create the following new class in your project

05/07/22

```
using System;
using DB.Extensibility.Contracts;

namespace DB.Extensibility.Plugins
{
    public class ExamplePlugin : PluginBase, IPlugin
    {
        public bool HasMenu
        {
            get { throw new NotImplementedException(); }
        }

        public string MenuLayout
        {
            get { throw new NotImplementedException(); }
        }

        public bool IsMenuItemVisible(string key)
        {
            throw new NotImplementedException();
        }

        public bool IsMenuItemEnabled(string key)
        {
            throw new NotImplementedException();
        }

        public void OnMenuItemPressed(string key)
        {
            throw new NotImplementedException();
        }

        public void Create()
        {
            throw new NotImplementedException();
        }
    }
}
```

Here, ExamplePlugin inherits IPlugin and PluginBase, both of which are provided by DB.Extensibility.Contracts. IPlugin is the interface that DesignBuilder uses to communicate with a plugin and declares methods for plugin creation/initialisation, methods for the description and interaction of a plugin's menu, and methods relating to the various simulation hook points as described in section *DesignBuilder Hooks*.

PluginBase provides default implementations of the simulation hook points. Inheriting from PluginBase is not required, but it's useful if don't intend to use the simulation hook points.

Note: The choice of namespace and class names are arbitrary.

4. DesignBuilder uses MEF to locate and load plugins. To make your plugin MEF aware, add the following code:

05/07/22

```
using System;
using System.ComponentModel.Composition;
using DB.Extensibility.Contracts;

namespace DB.Extensibility.Plugins
{
    [Export(typeof(IPlugin))]
    public class ExamplePlugin : PluginBase, IPlugin
    {
        public bool HasMenu
        {
            get { throw new NotImplementedException(); }
        }

        ...
    }
}
```

If you are unfamiliar with MEF, the new code tells MEF that the ExamplePlugin class should be made available to an importer as something that implements IPlugin.

5. The plugin is now properly configured with the required references, implements the required interface, provides default implementations for each simulation hook point, and is publically available to DesignBuilder for import. For all intents and purposes it is a valid plugin that can be loaded and used by DesignBuilder.

Nevertheless, as it's currently implemented, the plugin is not very useful. To make it more useful we'll create a menu structure that will appear in DesignBuilder's top-level menu. The menu will have only 1 item, which will display a message box when pressed.

A plugin's menu structure is described by a custom description language that is returned to DesignBuilder as a string via IPlugin.MenuLayout. The description language has the following rules:

- The '*' (asterisk) character indicates a new menu item
- Zero or more '>' characters indicate the indentation level of the menu item
- The ',' (comma) character indicates the split between the menu item's name as will be displayed in DesignBuilder and the item's id (or key), which will be used in communication between DesignBuilder and the plugin. The text preceding the ',' (and after any '*' or '>' characters) is the displayed name, and the text preceding the ',' is the menu item's id

Example 1: The string `"*Test Plugin,root*>Item 1,i1*>>Sub-item A,1a"` would create a menu structure that looks like:

```
Test Plugin      (key = root)
-- Item 1        (key = i1)
  -- Sub-item A  (key = 1a)
```

05/07/22

If a user press Sub-item A, DesignBuilder would inform the plugin by passing key “1a” to `IPlugin.OnMenuItemPressed`.

Example 2: The string “*Test Plugin,root*>Item 1,i1*>Item 2,i2” would create a menu structure that looks like:

```
Test Plugin (key = root)
-- Item 1   (key = i1)
-- Item 2   (key = i2)
```

As well as `MenuLayout`, `IPlugin` declares 4 other methods that are used in the creation and usage of a plugin’s menu. They are:

- **HasMenu**, which simply returns whether the plugin has a menu or not. If this returns false, the other menu-related methods will not be called
- **IsMenuItemVisible** returns whether a given menu item is currently visible or not. This is called when the plugin is first loaded and every time a menu item has been pressed
- **IsMenuItemEnabled** returns whether a given menu item is currently enabled or not. This is called when the plugin is first loaded and every time a menu item has been pressed
- **OnMenuItemPressed** is called when a user selects one of the plugin’s menu items. The key of the menu item (as defined by `MenuLayout`) is passed to the method to indicate which item has been pressed

Note that because `IsMenuItemVisible` and `IsMenuItemEnabled` are called every time a menu item has been pressed, a plugin can include logic to dynamically change the structure of its menu by changing the visibility of menu items in response to user actions. This is described in more detail in section *Writing a Plugin With a Dynamic Menu*.

Using what we’ve learned, it’s now possible to add a menu with 1 item that displays a message box when pressed, to the example plugin. The code looks like this:

05/07/22

```
using System.ComponentModel.Composition;
using System.Text;
using System.Windows.Forms;
using DB.Extensibility.Contracts;

namespace DB.Extensibility.Plugins
{
    [Export(typeof(IPlugin))]
    public class ExamplePlugin : PluginBase, IPlugin
    {
        class MenuKeys
        {
            public const string Root = "root";
            public const string ShowMessage = "showMessage";
        }

        public bool HasMenu
        {
            get
            {
                return true;
            }
        }

        public string MenuLayout
        {
            get
            {
                StringBuilder menu = new StringBuilder();
                menu.AppendFormat(
                    "*Example Plugin,{0}", MenuKeys.Root);
                menu.AppendFormat(
                    ">Show Message,{0}", MenuKeys.ShowMessage);
                return menu.ToString();
            }
        }

        public bool IsMenuItemVisible(string key)
        {
            return true;
        }

        public bool IsMenuItemEnabled(string key)
        {
            return true;
        }

        public void OnMenuItemPressed(string key)
        {
            if (key == MenuKeys.ShowMessage)
            {
                MessageBox.Show("Menu item pressed!");
            }
        }

        public void Create()
        {
        }
    }
}
```

05/07/22

- Before continuing with the example, notice the `IPlugin.Create` method. This method is the first method called after DesignBuilder has loaded a plugin. It is only called once and can be used to initialise a plugin.

If a plugin is to use the DesignBuilder API outside of the simulation hook methods (which are all passed an instance of the API), it is recommend that the `Api.Environment` object is stored as a member variable of the plugin at this point.

- Continuing on, once the plugin has been built, the only thing left to do is to copy the assembly to a location that DesignBuilder expects to find plugins.

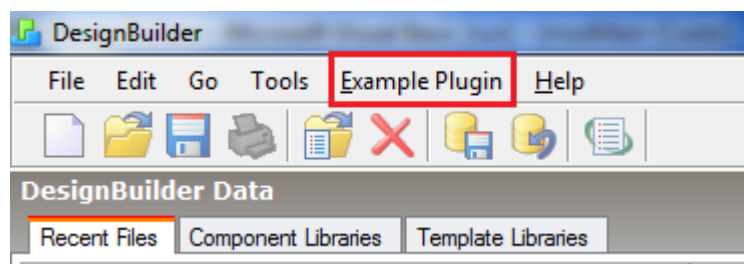
Navigate to your local application data folder (usually, this can be found at `C:\Users\username\AppData\Local\DesignBuilder`, where “username” is your user name. The AppData directory may be hidden in which case you’ll have to enable “Show hidden files, folders, and drives” in Windows Control Panel > Folder Options > View).

If there is no sub-directory called “User Plugins”, create one.

Each plugin assembly should be placed into its own directory in the “DesignBuilder/User Plugins” directory (the name of the directory does not matter). Note that `DB.Api.dll` and `DB.Extensibility.Contracts.dll` must also be present in this directory as well as any external assemblies that your plugin relies upon (E.G. `EPNet.dll`)

DesignBuilder will load all assemblies that export a class that implements `IPlugin` via MEF. If you do not wish a plugin to be loaded it must be removed from the User Plugins directory (or the `Export` class attribute can be removed and the plugin assembly rebuilt).

If everything went according to plan, you should see the following when you start DesignBuilder:



- Pressing `Example Plugin > Show Message` should display a message box with the message “Menu item pressed!”.

Writing a Plugin With a Dynamic Menu

Following on from section *Writing a Plugin*, this section will show you how to exploit `IPlugin.IsMenuItemVisible` and `IPlugin.IsMenuItemEnabled` to create a menu that can change based on user interaction.

05/07/22

This section assumes you've already read *Writing a Plugin* and will therefore not explain the process in as much detail.

1. For this plugin, we are going to create a menu that can change the visibility and state of its menu items based on the user's interaction. As a base we will start with the following code:

```
[Export(typeof(IPlugin))]  
public class ExamplePlugin : PluginBase, IPlugin  
{  
    class MenuKeys  
    {  
        public const string Root = "root";  
    }  
  
    public bool HasMenu  
    {  
        get  
        {  
            return true;  
        }  
    }  
  
    public string MenuLayout  
    {  
        get  
        {  
            StringBuilder menu = new StringBuilder();  
            menu.AppendFormat(  
                "*Example Plugin,{0}", MenuKeys.Root);  
            return menu.ToString();  
        }  
    }  
  
    public bool IsMenuItemVisible(string key)  
    {  
        return true;  
    }  
  
    public bool IsMenuItemEnabled(string key)  
    {  
        return true;  
    }  
  
    public void OnMenuItemPressed(string key)  
    {  
    }  
  
    public void Create()  
    {  
    }  
}
```

2. It's important to understand that when DesignBuilder loads a plugin for the first time, it will call `IPlugin.HasMenu` followed by `IPlugin.MenuLayout` (assuming `HasMenu` return true). This process is only carried out once because DesignBuilder needs to know the complete menu layout before it displays it's GUI. `IPlugin.IsMenuItemVisible` and `IPlugin.IsMenuItemEnabled`, however, are called throughout the lifetime of the plugin.

05/07/22

This is important because it means that `IPlugin.MenuLayout` must return the complete menu structure to DesignBuilder and only through toggling visibility can the structure be changed.

With that in mind, add the following code to the plugin:

```
[Export(typeof(IPlugin))]
public class ExamplePlugin : PluginBase, IPlugin
{
    class MenuKeys
    {
        public const string Root = "root";
        public const string State = "state";
        public const string Visibility = "visibility";
        public const string EnableAll = "enableAll";
        public const string DisableAll = "disableAll";
        public const string VisibleAll = "visibleAll";
        public const string InvisibleAll = "invisibleAll";
    }

    ...

    public string MenuLayout
    {
        get
        {
            StringBuilder menu = new StringBuilder();
            menu.AppendFormat(
                "*Example Plugin,{0}", MenuKeys.Root);
            menu.AppendFormat(
                ">State,{0}", MenuKeys.State);
            menu.AppendFormat(
                ">>Enable All,{0}", MenuKeys.EnableAll);
            menu.AppendFormat(
                ">>Disable All,{0}", MenuKeys.DisableAll);
            menu.AppendFormat(
                ">Visibility,{0}", MenuKeys.Visibility);
            menu.AppendFormat(
                ">>Make All Visible,{0}", MenuKeys.VisibleAll);
            menu.AppendFormat(
                ">>Make All Invisible,{0}", MenuKeys.InvisibleAll);
            return menu.ToString();
        }
    }

    ...
}
```

3. Now the menu structure is defined the plugin will need some way of keeping track of the state of each menu item. I.E. Whether or not a menu item is visible or enabled. This can be done however you like, but in this example we shall create a simple class and a dictionary:

05/07/22

```
using System.Collections.Generic;

...

[Export(typeof(IPlugin))]
public class ExamplePlugin : PluginBase, IPlugin
{
    ...

    class MenuItem
    {
        public bool IsEnabled { get; set; }
        public bool IsVisible { get; set; }

        public MenuItem(
            bool enabled = true,
            bool visible = true)
        {
            IsEnabled = enabled;
            IsVisible = visible;
        }
    }

    private readonly Dictionary<string, MenuItem> mMenuItems =
        new Dictionary<string, MenuItem>();

    ...

    public void Create()
    {
        mMenuItems.Add(MenuKeys.Root, new MenuItem());
        mMenuItems.Add(MenuKeys.State, new MenuItem());
        mMenuItems.Add(MenuKeys.Visibility, new MenuItem());
        mMenuItems.Add(MenuKeys.EnableAll, new MenuItem());
        mMenuItems.Add(MenuKeys.DisableAll, new MenuItem());
        mMenuItems.Add(MenuKeys.VisibleAll, new MenuItem());
        mMenuItems.Add(MenuKeys.InvisibleAll, new MenuItem());
    }
}
```

Notice that `IPlugin.Create` is used to initialise the plugin. You could do this in the constructor if you prefer.

4. `IPlugin.IsMenuItemVisible` and `IPlugin.IsMenuItemEnabled` can now simply return the `IsVisible` or `IsEnabled` properties of the specific menu item like so:

05/07/22

```
[Export(typeof(IPlugin))]  
public class ExamplePlugin : PluginBase, IPlugin  
{  
    ...  
  
    public bool IsMenuItemVisible(string key)  
    {  
        return mMenuItems[key].IsVisible;  
    }  
  
    public bool IsMenuItemEnabled(string key)  
    {  
        return mMenuItems[key].IsEnabled;  
    }  
  
    ...  
}
```

5. The menu structure is defined, the plugin has a means of keeping track of each menu item's state, and it can now report the state back to DesignBuilder. All that remains is to actually change the state of a menu item depending on the actions of the user.

Since each menu item will have a different action when pressed by the user, we will add a property to the nested MenuItem class to represent this:

```
class MenuItem  
{  
    public Action Action { get; set; }  
    public bool IsEnabled { get; set; }  
    public bool IsVisible { get; set; }  
  
    public MenuItem(  
        Action action = null,  
        bool enabled = true,  
        bool visible = true)  
    {  
        Action = action ?? delegate{};  
        IsEnabled = enabled;  
        IsVisible = visible;  
    }  
}
```

And then initialise each menu item with an appropriate action:

05/07/22

```
public class ExamplePlugin : PluginBase, IPlugin
{
    ...

    public void Create()
    {
        mMenuItems.Add(MenuKeys.Root,
            new MenuItem());
        mMenuItems.Add(MenuKeys.State,
            new MenuItem());
        mMenuItems.Add(MenuKeys.Visibility,
            new MenuItem());
        mMenuItems.Add(MenuKeys.EnableAll,
            new MenuItem(OnEnableAll));
        mMenuItems.Add(MenuKeys.DisableAll,
            new MenuItem(OnDisableAll));
        mMenuItems.Add(MenuKeys.VisibleAll,
            new MenuItem(OnVisibleAll));
        mMenuItems.Add(MenuKeys.InvisibleAll,
            new MenuItem(OnInvisibleAll));
    }

    private void OnEnableAll()
    {
        mMenuItems[MenuKeys.DisableAll].IsEnabled = true;
        mMenuItems[MenuKeys.Visibility].IsEnabled = true;
    }

    private void OnDisableAll()
    {
        // don't disable EnableAll
        mMenuItems[MenuKeys.DisableAll].IsEnabled = false;
        mMenuItems[MenuKeys.Visibility].IsEnabled = false;
    }

    private void OnVisibleAll()
    {
        mMenuItems[MenuKeys.InvisibleAll].IsVisible = true;
        mMenuItems[MenuKeys.State].IsVisible = true;
    }

    private void OnInvisibleAll()
    {
        // don't make VisibleAll invisible
        mMenuItems[MenuKeys.InvisibleAll].IsVisible = false;
        mMenuItems[MenuKeys.State].IsVisible = false;
    }
}
```

Note that changing the visibility or enabled state of a parent menu item also affects the state of its children. E.G. OnDisableAll changes MenuKeys.Visibility's enabled state to false, which results in MenuKeys.VisibleAll and MenuKeys.InvisibleAll also being disabled.

- Each relevant menu item now has an appropriate action or a default action to do nothing. The final piece in the puzzle is to forward IPlugin.OnMenuItemPressed to the correct menu item's action as follows:

05/07/22

```
[Export(typeof(IPlugin))]  
public class ExamplePlugin : PluginBase, IPlugin  
{  
    ...  
  
    public void OnMenuItemPressed(string key)  
    {  
        mMenuItems[key].Action();  
    }  
  
    ...  
}
```

7. The plugin is now complete and can be copied to the User Plugin directory as described in section *Writing a Plugin* item 7. Upon starting DesignBuilder the plugin should be loaded and you can see how pressing Example Plugin > Visibility > Make All Invisible changes the structure of the menu (from the user's point-of-view).

Plugin API Versions

When new plugin hookpoints are introduced a new plugin interface is created, which inherits the previous plugin interface. This ensures that plugins that support an older interface will still work without the plugin author having to change their code.

From v7.1.1 DesignBuilder supports two plugin interfaces, IPlugin and IPlugin2. In addition to supporting all IPlugin's hook points, IPlugin2 adds three new hook points – ScreenChanged, ModelLoaded, and ModelUnloaded. For your plugin to take advantage of the latest API you must define your plugin class accordingly:

```
[Export(typeof(IPlugin2))]  
public class ExamplePlugin : PluginBase2, IPlugin2  
{  
}
```

IPlugin supported hook points

```
void BeforeEnergyIdfGeneration();  
void BeforeEnergySimulation();  
void AfterEnergySimulation();  
void BeforeHeatingIdfGeneration();  
void BeforeHeatingSimulation();  
void AfterHeatingSimulation();  
void BeforeCoolingIdfGeneration();  
void BeforeCoolingSimulation();  
void AfterCoolingSimulation();  
void BeforeDaylightSimulation();  
void AfterDaylightSimulation();  
void BeforeCfdSimulation();  
void AfterCfdSimulation();  
void BeforeOptimisationStudy();  
void AfterOptimisationStudy();  
void OnDesignVariableChanged(int variableId, string value);  
void BeforeCostAndCarbon();  
void AfterCostAndCarbon();  
void AtCommandLine(string arg);
```

05/07/22

IPlugin2 supported hook points (in addition to all IPlugin hook points)

```
void ModelLoaded();  
void ModelUnloaded();  
void ScreenChanged(ScreenCode screenCode);
```