How to create your own plugins using Minimum Plugin for a template

Bob Zawalich November 10, 2020

The downloadable plugin **Minimum Plugin** (category **Developers' Tools**) is a user-modifiable plugin intended to be used for creating stable custom plugins. The intention of the plugin is that you should make a copy of the plugin, change the internal plugin name, install it, then change the code to accomplish what you want.

The code in **Minimum Plugin** (as of version 01.50.00) is in the public domain (unlike most downloadable plugins whose code is copyrighted by the author), so you may change code as you like and reuse the code for any use you desire. There is no support for using this code, and you use it at your own risk, but it is free to use.

As of the time this document was written, at the start of the Run method of **Minimum Plugin** you will find this block of comments:

```
// ************** Copyright/Public domain Comment Block ******************
// Minimum Plugin was originally written Bob Zawalich 2011.
// Minimum Plugin has been donated to the public domain.
// Minimum Plugin and all its routines may be used freely in other plugins without attribution.

// Plugins based on Minimum Plugin may be copyrighted by their author, but individual routines that
// come from Minimum Plugin and are unchanged are not intended (by Bob Zawalich, who is not a lawyer)
// to be copyrightable.

// See the routine aa_HowToCustomizeAPlugin for hints on using this plugin
// See the routine aa_INDEX_OF_ROUTINES for a list of the Methods that are included in Minimum Plugin

// Feel free to remove this block and the 2 aa_ routines when you use Minimum Plugin as a template for your
own plugin.
// ***************** End Copyright block ********************
```

Feel free to remove this block and the 2 aa_ routines when you use **Minimum Plugin** as a template for your own plugin. Feel free to copyright plugins derived from **Minimum Plugin,** but any code that is inchanged in **Minimum Plugin** will likely remain uncopyrighted.

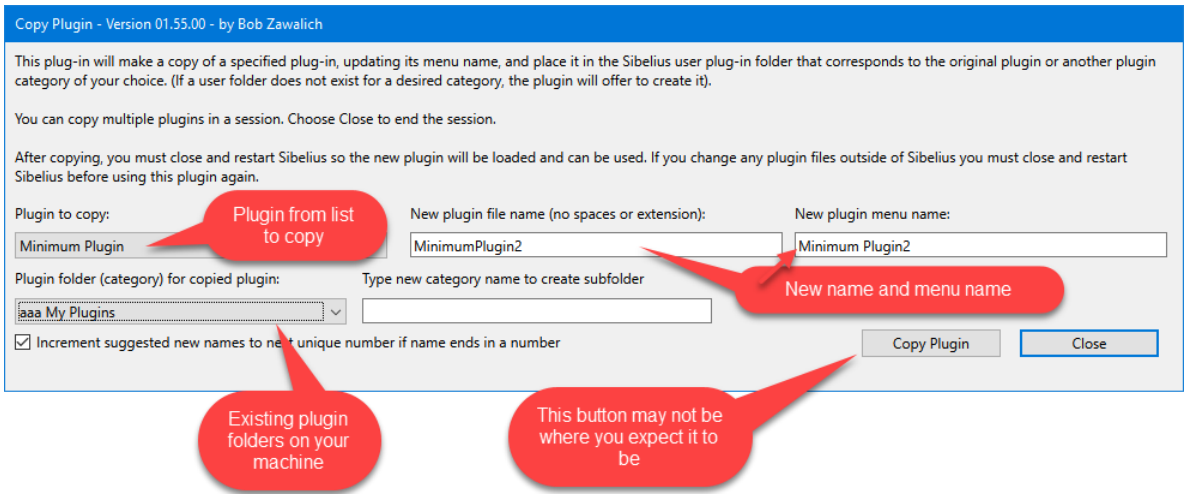
This plugin as written can only be run in the full version of *Sibelius 6.2* or later (in the 20xx timeframe, you can only use *Sibelius Ultimate*, not the cut-down *Sibelius* or *Sibelius First* products. The instructions that follow reference the Plugin Installer, which is only available in *Sibelius 7.1.3* or later. In this document, when I say *Sibelius* I usually mean *Sibelius Ultimate*.

Note that, while you can use a separate Text Editor to edit your plugin code, all the tools you really need to create plugins are included free with Sibelius. If you use **Edit>Plugins**, you have access to the dialog editor, which is a big deal, and you can also edit and immediately run a plugin without needing to close and restart Sibelius or use **Load** and **Unload** in **Edit>Plugins** to bring in an edited score.

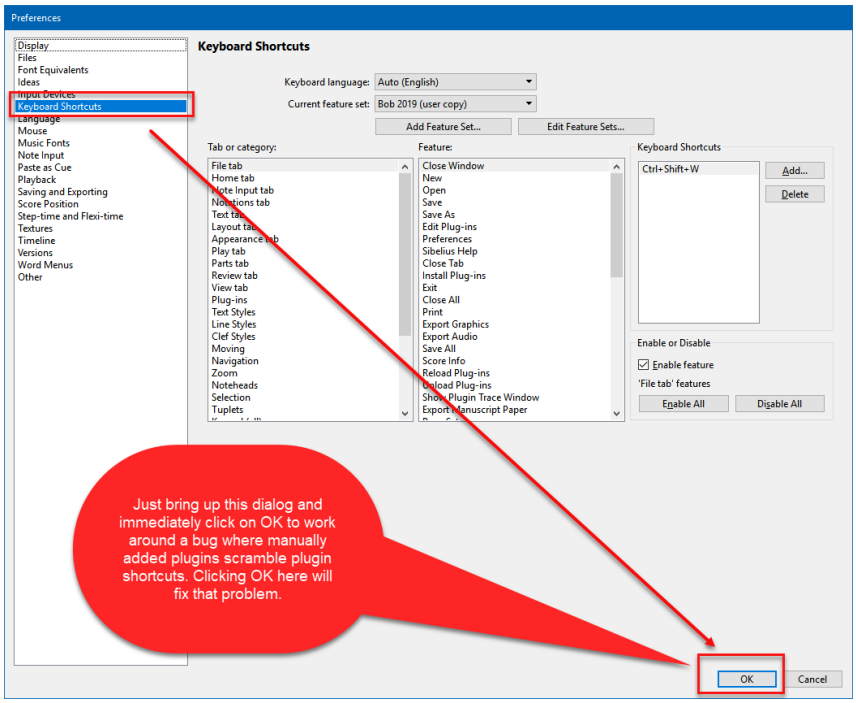
There are simpler ways to create a new plugin, (see the document *Write your own simple Sibelius plugin*), but using the **Minimum Plugin** template will give you a plugin that will do common score checks, need a small amount of initial tweaking, and provide the starting point for a stable plugin. When I create a new plugin from scratch (rather than modifying an existing plugin), I always start with **Minimum Plugin**.

Here is a set of steps I recommend using for setting up a custom plugin:

1. Using the Installer, install the plugins **Minimum Plugin** (category **Developers' Tools**) and **Copy Plugin** (category **Developers' Tools**).
2. Run **Copy Plugin**, choose **Minimum Plugin** from the list of installed plugins as the plugin to copy, and choose a unique name for your new plugin. Be sure to click on the **Copy Plugin** button, not **Close**. Choose a convenient category/plugin subfolder for your new plugin. In this case I created a new subfolder called **aaa My Plugins**, which will appear at the top of the list of categories so it will be easy to find. You could copy multiple plugins in the same plugin session, but we only need 1, so after copying, click on **Close**.

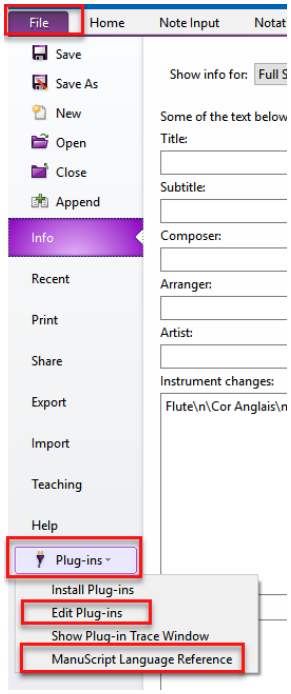


3. This will make a copy of **MinimumPlugin.plg**, changing the name of the file and the internal plugin menu name within the file, placing the new plugin file in the specified plugin subfolder.
4. Now you need to take 2 additional steps before editing your new plugin. Sibelius loads all the installed plugins into memory when it starts up, so it will not know about your new plugins (or any plugin file that you edit outside of Sibelius) until you close and restart Sibelius:
    a. Close and restart Sibelius, which will cause your plugin to be loaded into memory and made available to Sibelius.
    b. Click on the **File** tab, choose **Preferences>Keyboard Shortcuts**, and immediately choose **OK**. This fixes a Sibelius bug that scrambles plugin shortcuts when new plugins are added manually (or in our case, via a plugin).

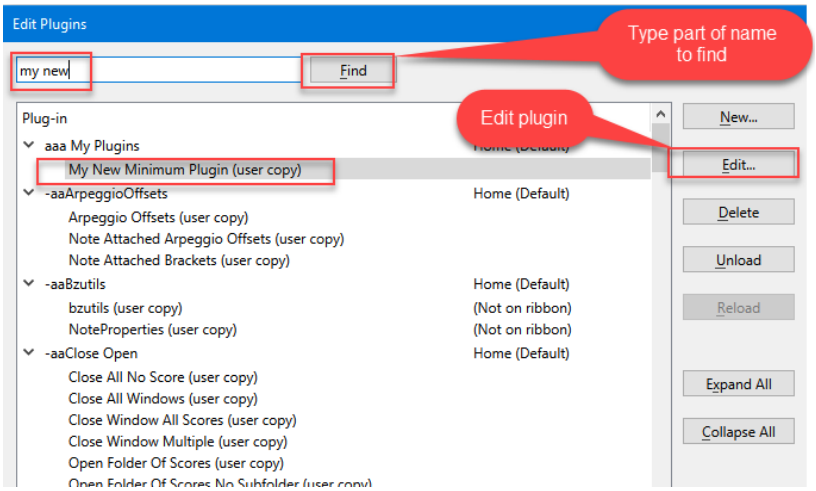


At this point the new plugin is equivalent, in Sibelius' eyes, to any other installed plugin.

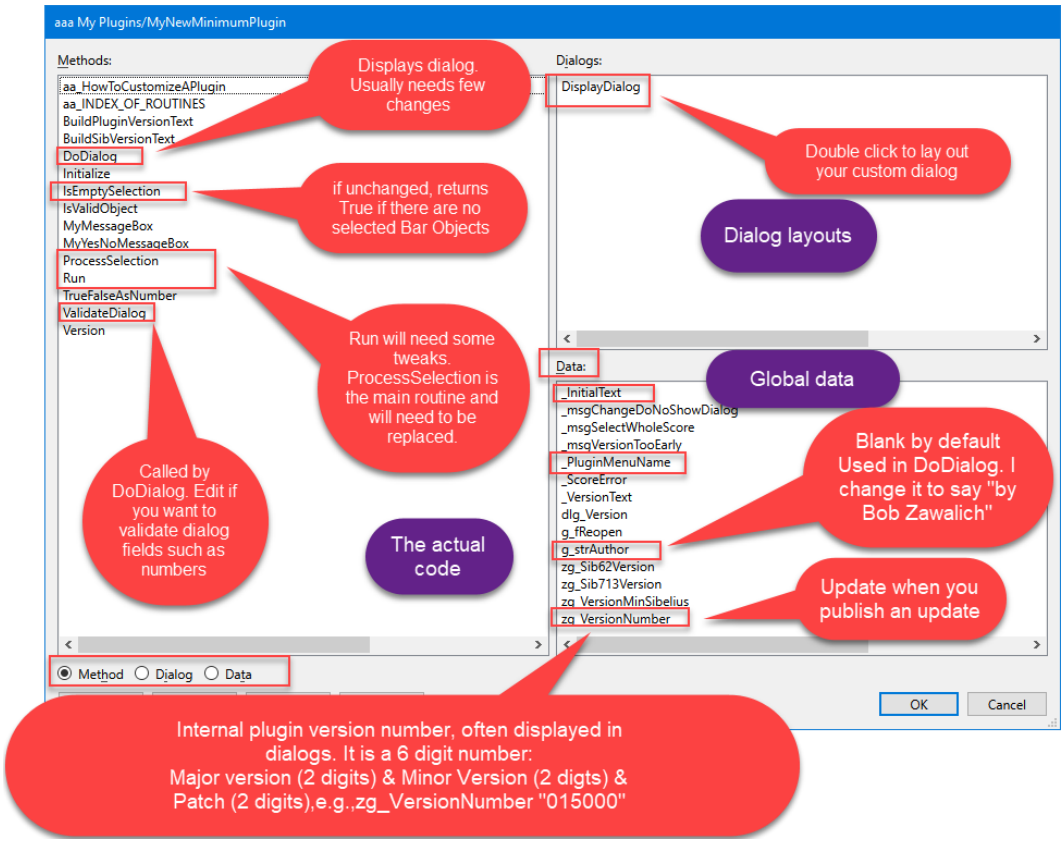
Now we can edit the new plugin. Go to **File >Plug-ins>Edit Plug-in**, and you will see the plugin editor. Type your new plugin menu name (the one with the spaces) into the **Find** box, click on Find, and then click on **Edit**. (Note that the Reference manual for **ManuScript**, the language plugins are written in, is also accessed from this menu).



Here is the **Edit Plugins** dialog:

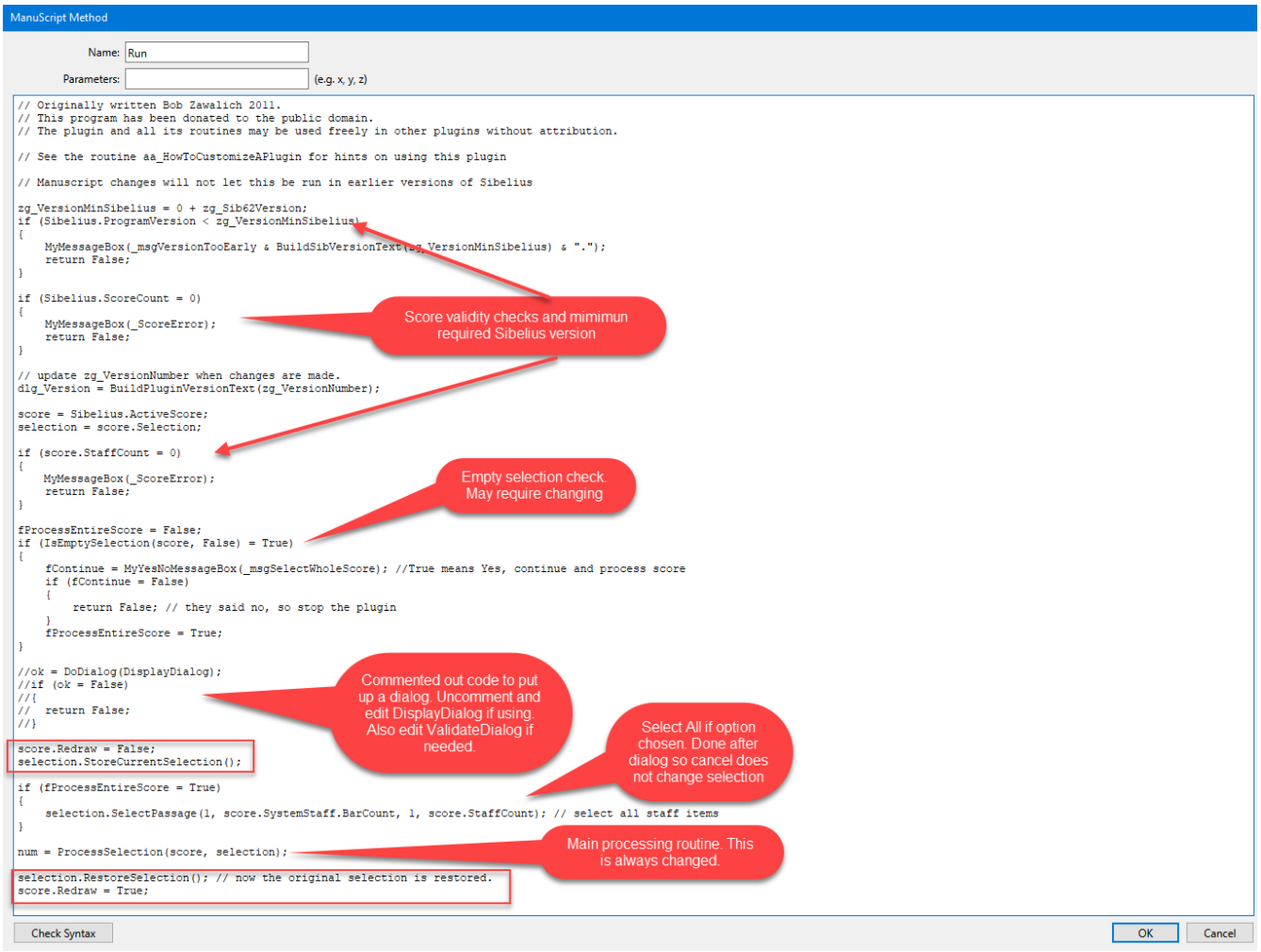


After clicking on Edit you should see something like this:

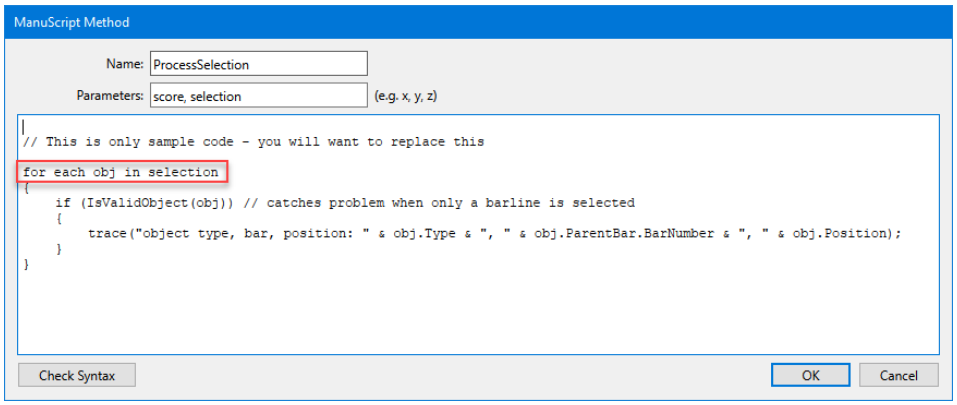


We will usually only need to significantly change **Run()** and **ProcessSelection()**, so let's have a look there.

Run() is the routine that is almost always called when you invoke a plugin from a menu or a shortcut. At the time this document was written, this is what the **Run** routine looked like.



- It starts by checking that the plugin can be run in the current version of Sibelius. By default it will run in Sibelius 6 or later. If you will use ManuScript commands that require later versions of Sibelius, change *zg_VersionMinSibelius* to be the minimum Sibelius version number required.
- It makes some checks that there is a score and that it has staves. If you have different requirements, change this code.
- It calls **IsEmptySelection**(), which can check that the selection has what you will need. By default it returns True only if there are no selected staff Bar Objects. If it fails, it gives the option to select the entire score or cancel.
- There is a commented out block that calls **DoDialog()**. If you want to bring up a dialog, remove the // comment markers so the code will be executed. You might need to make a few changes to **DoDialog()** and **ValidateDialog()**. You will also need to edit the dialog description for **DisplayDialog**, which by default is a stub with a few sample controls. Check some available plugins that use dialogs for examples of how they work.
- The plugin now calls *score.Redraw = False;* which keeps Sibelius from updating the screen whenever anything changes. This is probably the single most important way to speed up the processing in a dialog. *score.Redraw = False;* will be called later.
- It also calls *selection.StoreCurrentSelection()*; This will let you save the original selection before you change anything, and restore it later if desired. If the purpose of your plugin involves changing the selection, as a filter might do, you will not want to call *selection.RestoreSelection()* later.
- All that is left is to call **ProcessSelection(score, selection)**, which does the actual work of the plugin, usually by processing selected objects.



You can do anything you want here as long as it is legal in the **ManuScript** language. In a "*for each*" loop you cannot add or delete objects that are being iterated over, but you can change the properties of an object, or just analyze an object as we are doing here. You can make a separate loop to store the objects you get in a "*for each*" loop into a Sparse Array, and then process the objects in the Sparse Array without worrying about the "for each" limitations, and you will see a lot of plugins that do this.

You may find the ManuScript language cannot do what you want and will need to look for a different approach.

You can look in other plugins whose names might sound similar to see what they do, and you can look at the ManuScript Language Reference in **File>Plug-ins**.

This sample code just looks at all selected objects one at a time (*for each obj in selection*) and writes some information about each object to the plugin **Trace** window. There are lots of things you can do by just writing some new code in *ProcessSelection()*, and you can also change the code completely.

Version numbers

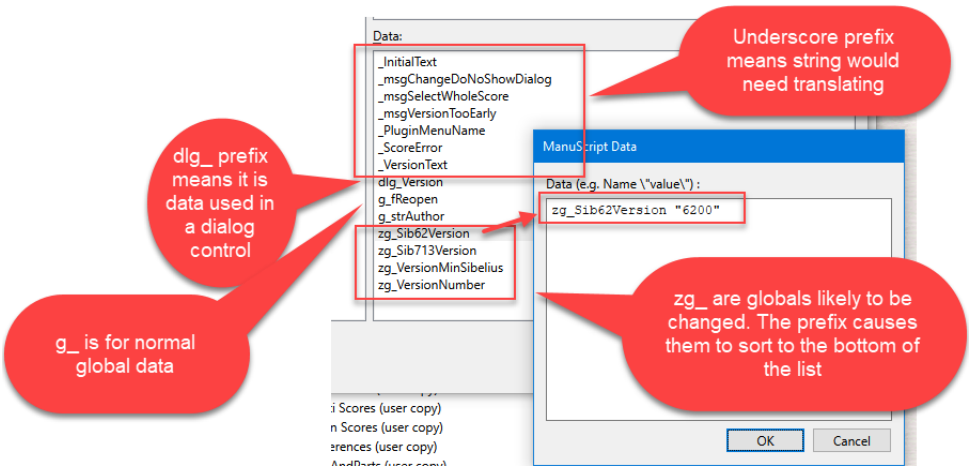
There are 2 kinds of version numbers you are likely to encounter in plugins. In both cases they are formatted as integers that can be directly compared, and are often formatted for display.

The first of these is the **Sibelius Version** number, which you can access in the variables **Sibelius.ProgramVersion** and **score.OriginalProgramVersion.**

You will see this used in many plugins as a way to test that the version of Sibelius that is running has access to all the ManuScript commands used in the plugin. At the start of **Run()** in **Minimum Plugin** you will see

```
zg_VersionMinSibelius = 0 + zg_Sib62Version;
if (Sibelius.ProgramVersion < zg_VersionMinSibelius)
{
   MyMessageBox(_msgVersionTooEarly & BuildSibVersionText(zg_VersionMinSibelius) & ".");
   return False;
}
```

I use the prefix "zg_" for global variables I expect to change. It puts them at the end of the data list, so they will be easy to find. Here is an example of a **Data** block in the **Plugin Editor**. I will discuss global variables in general later, but for the moment, see that I have defined several variables with zg_ prefixes, and that zg_Sib62Version is defined as "6200".

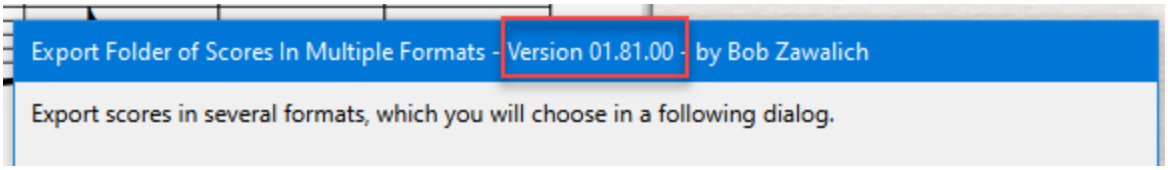


Thus, in the code above,

```
if (Sibelius.ProgramVersion < zg_VersionMinSibelius)
```

if **Sibelius.ProgramVersion** has a numerical value less than 6200, the plugin will put up an error message and terminate.

The second version number you will find in plugins *I* have written and published is stored in a global variable called **zg_VersionNumber**. I update this number every time I publish a new version of a plugin, and I recommend that you use it for the same purpose. It makes it easier to tell if someone has the most recently updated version of the plugin. In my plugins I display a formatted version of **zg_VersionNumber** in dialogs, and some plugins, such as **Are Your Plugins Up To Date**, access this variable in other plugins, so there are advantages of using this variable for this purpose.



zg_VersionNumber is a 6-digit string literal, such as "010500". The method **BuildPluginVersionText** formats it into three 2-digit fields, separated by periods. The fields are described as:

```
Major version (2 digits) & Minor Version (2 digits) & Patch (2 digits)
```

If a field number is less than 10, I add a leading zero so each field is 2 digits.

Typically, I change the minor version. If there are small cosmetic changes, I will update the patch, and for major rewrites I will change the major version.

Sibelius.ProgramVersion also has 3 fields, the *major version*, *minor version*, and *revision*, but it did not use leading zeroes, so version 7.1.3 would be stored as 7130. Once Sibelius switched to a year/month format, such as 2018.1, things got more complicated. 2018.1 was originally stored as 18100, with a 2-digit *year*, 1- or 2-digit *month*, and 2-digit *revision,* but that fell apart as time went on (specifically when we got to 2-digit months). Currently the internal representation of the **ProgramVersion** is an 8-digit number (YearMonthSubversion YYYYMMPP), which can still be compared numerically against any previous version number. For purposes of generating error messages, the new form is easier to format that the old one, since the number of digits in each field is fixed. The routine **BuildSibVersionText** will format either the new 8-digit format or the earlier 4- or 5 digit formats for display purposes.

Global Variables

I try to avoid using global variables (as stored in the Data area in the plugin editor), but sometimes they are necessary or useful.

- Variables used in dialog controls must be global variables (I usually use a dlg_ prefix for these)
- Methods called from dialog controls, such as buttons, can pass no parameters, so the called routines can only access global data.
- Literal text (text in double quotes) that would be translated is stored as global data with an underscore(_) prefix (_InitialText "This is the dialog box."). This is a standard for all shipping Sibelius plugins, and I use it in my own plugins as well. Doing so allows you to change displayed text without having to change the code in a method, which avoids errors.
- Data that I want to access from other plugins that call this plugin (such as **zg_VersionNumber** or sometimes dialog settings) need to be global data, or need to be passed back from a separate Method.
- Other data that I want to change without affecting the code text, or data I want to describe with a name rather than just being a number I will use as global variables. I use **zg_Sib62Version** rather than "6200" so I know what its purpose is, rather than just knowing its value.

One problem with global data is that if you assign a piece of global data to another variable, such as

```
ver = zg_VersionNumber;
```

it will not work, because both global variables and arrays (but not Sparse arrays) assign the *address* of the variable rather than its *value*. This has some utility when you are creating arrays or arrays, but it is mostly a bad idea. In most cases you need to force the assignment to be a *value*. Adding to zero or concatenating a string to an empty string is the usual fix:

```
ver = 0 + zg_VersionNumber;
strText = "" & _InitialText;
```

If you don't do this, things will go wrong in ways that are very difficult to debug. Also, do not store True or False in an array. Use 0 for false and 1 for True. The method **TrueFalseAsNumber** or **utils.CastAsInt** will make that conversion for you.

Variable Naming and prefixes

I was a software designer at Microsoft, and we had a way of naming variables and routines (Hungarian) that provided a fast way to choose a name. It provided a way to tell what the purpose of a variable was just by looking at the name, as long as you followed the rules. I follow a relaxed version of Hungarian in my plugins. I describe it in another document, but if you see arrNames, strNameCur, nrFirst, or fContinue, you can know that you have an array, a string, a NoteRest, or a *flag (or boolean)*, which is a value that will be either True or False. I use *f* as the prefix for historical reasons. In my convention, variables always start with a lower-case letter, and Methods always start with an upper case letter.

I always delineate blocks of code with the brackets on separate lines, because I find it easier to see blocks that way. I tend to use small blocks of code was well, to avoid deep nesting at the cost of not seeing all the code at one time. So in my code you will see

```
if (IsEmptySelection(score, False) = True)
{
   fContinue = MyYesNoMessageBox(_msgSelectWholeScore); //True means Yes, continue and process score
   if (fContinue = False)
   {
      return False; // they said no, so stop the plugin
   }
```

```
    fProcessEntireScore = True;
}
```

and not

```
if (IsEmptySelection(score, False) = True) {
    fContinue = MyYesNoMessageBox(_msgSelectWholeScore); //True means Yes, continue and process score
    if (fContinue = False)     {  return False; // they said no, so stop the plugin }
    fProcessEntireScore = True; }
```

My way takes up more space, but the block structure is immediately visible, which I find useful, especially when the editor does not color or mark blocks for you.

Also: I favor clarity over efficiency, though I still try to write the most efficient code I can, and will optimize if things are too slow. So the line

```
if (IsEmptySelection(score, False) = True)
```

could be written as

```
if (IsEmptySelection(score, False))
```

and would mean the same thing. But I can read the first one and understand it more quickly than the second, so I write it that way. It is intentional inefficiency.

I often say

```
if (fCreateFolderIfMissing)
{
…
```

though, because I know from the "f" prefix that this is a Boolean/flag variable, and that "if (x)" means "if (x = True)" in ManuScript.

My expectation is that plugins will need to be modified, so I design them to be as easy to modify as I can. I also expect that *I* will be the modifier, so while I try to be clear for anyone, I also want it to be clearest for *me*, so I can "sightread" the code when I come back to it 5 years later and can still see what I intended to do. It works for me.

Other plugin templates

There are several downloadable plugins that are set up to be freely modified, most of which are templates for filters. Each has default code that performs some simple action; that default code is meant to be replaced by your code. The places where code needs to be replaced are clearly marked.

These are in the plugin category **Filter and Find**:

- **Custom Filter**
    - This filters objects in the system staff and normal staves. The default code filters Text, SystemTextItem and LyricItem objects.
- **Custom Staff Filter**
    - This filters objects in non-system staves, skipping system objects. The default code filters NoteRest objects that contain a single note only, skipping rests and chords.
- **Custom System Filter**

    - This filters objects in the system staff only. If the original selection is a non-system passage selection it will be converted to a system selection covering the same bar range, and only objects in the system staff will be processed. The default code filters SpecialBarline objects.
- **Custom Note Filter**
    - This filters Note objects, effectively skipping the system staff. The default code filters quarter notes.

These are templates that are not specifically set up as filters:

- **Custom Simple Plugin (category Developers' Tools)**
    - This template has no error checking. It is meant for quick and dirty personal plugins. Some code in the methods **IsDesiredObject()** and **Run()** is meant to be changed. By default, the plugin filters Line objects in the system staff and normal staves.

And so…

Here is what I wish to impart: there are a lot of situations where you might want to do something that is not handled by Sibelius or an existing plugin. You might be able to find someone to write one for you, but there are not very many plugin writers out there. In many cases you could start from something that exists (either a public domain plugin or something you have permission to modify), and if you are lucky, a small tweak will get you what you want. And you can do it yourself.

Minimum Plugin and the other public-domain templates listed here can be a good starting place for writing your own plugin, and you can look at the code in any of the published or shipping plugins to see how other programmers have accomplished something like what you want to do.

There is a plugin developers' mailing list you can sign up for and people there are generally helpful about answering questions.

Good luck. Happy programming!