Parent/Child plugins

Bob Zawalich July 20, 2020

Many plugins present complicated dialogs with lots of options, and we might want to be able to run the plugin without having to deal with the dialog, especially when running it multiple times. Unlike Sibelius commands, which can store settings in Engraving Rules, a plugin needs to carry all its settings around with it.

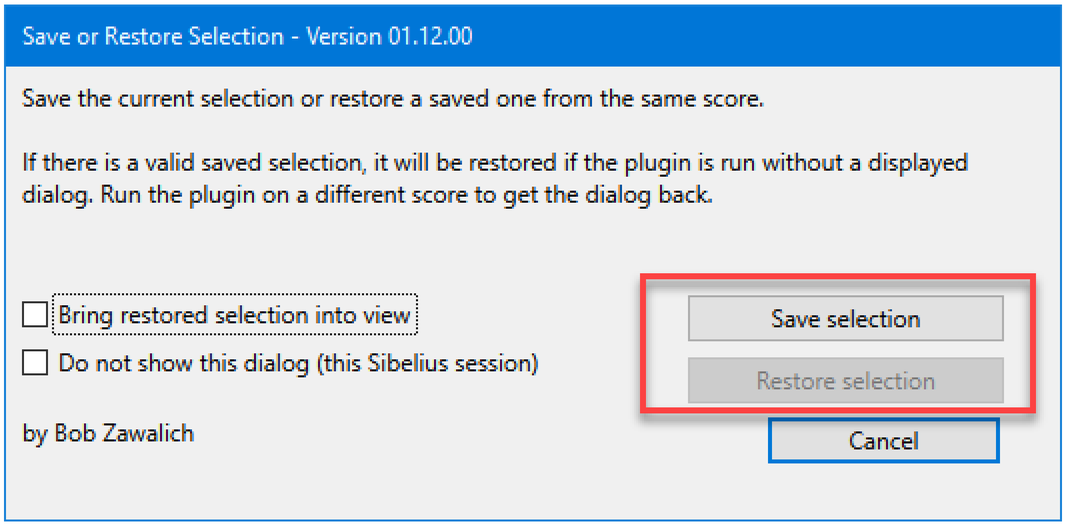
Some ways of doing this (Cloning plugins, using Do Not Show Dialog, and using the Run Plugin Hide Dialog model) are discussed in other documents, such as **On Hiding Dialogs in Sibelius Plugins.**

Here I want to discuss an approach of designing plugins so they can be easily called without bringing up their dialogs, a model I call **Parent/Child plugins**.

Parent/Child plugins

The Parent/Child plugin model creates a main, full-featured **Parent** plugin, and any number of smaller-featured, dialog-free, **Child** plugins.

The model is based on the work I have done in **cmdutls.plg**, which itself is based on some plugins like **Save and Restore Selection**, which is published with a *Parent* plugin, **Save Or Restore Selection**, and 2 *Child* plugins, **Save Selection** and **Restore Selection**.


Save or Restore Selection - Version 01.12.00

Save the current selection or restore a saved one from the same score.

If there is a valid saved selection, it will be restored if the plugin is run without a displayed dialog. Run the plugin on a different score to get the dialog back.

☐ Bring restored selection into view

☐ Do not show this dialog (this Sibelius session)

by Bob Zawalich

The children plugins pretty much do what the **Save Selection** and **Restore Selection** buttons do. In this case they need to be aware of whether there is a selection to save, or there is one to restore (in the screenshot, there is no saved selection and the **Restore** button is disabled, so **Restore Selection** has to deal with that scenario since it will not be able to restore anything. In such a case, it would just put up a message box describing the situation, and then the plugin exits.

The **Double Note Values/ Halve Note Values** plugins also use a parent child model, where **Halve Note Values** just calls the main routine of **Double Note Values**, and many of the **Transformations** plugins work the same way.

The Clipboards plugins family consists of the parent **Clipboards.plg** and a set of children: ClipCopy1, 2, 3... ClipPaste1, 2, 3...

The big advantage of this model is that when updates need to be made nearly all of the work can be done in the Parent plugin, whereas with clones, you would need to either recreate the clones from the parent after changing the parent, or in the worst case, merge changes to the parent into all the clones.

More Parent/Child-style plugins

Cmdutils.plg is a collection of slightly less formal Parent/Child plugins, with the child **_cu** routines almost always calling **_Full** routines, which are the Parent plugins. Even the full routines do not typically have dialogs, but it is the same basic idea. For example this **_Full** routine.

Add_Line_Full (score, selection, styleTextOrId, fSelectNewObject)
- Adds a line of the requested style. styleTexOrId must be a StyleAsText or StyleId valid in the score, spelled EXACTLY as ManuScript expects it to be.

is called by all of these **Add Lines** routines plus their corresponding **AddSelect_Line** routines:

Add_Line_cu(styleTextOrId)
- Adds a line of the requested style. styleTexOrId must be a StyleAsText or StyleId valid in the score, spelled EXACTLY as ManuScript expects it to be.

Add_Line_8va_cu()
Add_Line_Box_cu()
Add_Line_Bracket_Vertical_Left_cu()
Add_Line_Bracket_Vertical_Right_cu()
Add_Line_Ending_First_cu()
Add_Line_Ending_Second_cu()
Add_Line_Hairpin_Crescendo_cu()
Add_Line_Hairpin_Diminuendo_cu()
Add_Line_Plain_cu()
Add_Line_Slur_cu()
Add_Line_Trill_cu()
Add_Line_Vertical_cu()
- Adds a line of the specified style to the selection
- Some of the lines, especial vertical lines like brackets and box lines are given slightly different (and better, in my opinion) positions that Sibelius uses in the Lines menu.

Doing this lets me have nearly all the _cu routines require no parameters (the exception is **Add_Line_cu,** with a single parameter for the **styleNameOrId**), so you can just use **Add_Line_Trill_cu** to add a trill line, and it will pass the appropriate style id to **Add_Line_Full**, where all the work is done.

Each of the children set up a small number of parameters, and call the parent. The big win here is that the code in the children is minimal, and if I need to change code or fix bugs, all that code is likely to be in the **Parent** routine, and I will usually not need to touch the children.

I have designed or modified a number of other plugins to be less formally **Parent** plugins. Cmdutils calls a number of these, including

Apply Named Color
Bracket Text
Add Intervals
Transpose By Interval

These have methods, typically names with an **API_** suffix, that can be called by other plugins, who pass in parameters. The **API_** suffix is an indication that the routine will not use any global variables that a caller would not have access to.

In the future I plan to use the more formal Parent/Child model I describe below for any plugin with a reasonably complex dialog that could be usefully called with a simpler set of parameters, as the cmdutils routines are set up. It think it would be easier use such plugins as Parent plugins in the future if set up that way from the start.

It is relatively inexpensive to set up a plugin to be a Parent plugin.

This would provide easier ways to create child plugins that can use a subset of the features of the parent plugin, without needing to bring up any dialogs.

Designing a plugin to be a Parent/Child plugin

In my experience it is better to design plugins to work as Parent/Child plugins from the start, rather than retrofitting them later. When the design is done, you know in advance that you need to limit where global variables are used, and to call the parent with a subset of features is usually quite straightforward.

In the general Parent/Child model, the parent plugin is stand-alone. It has a dialog and a reasonable amount of options, including the ability to run without a dialog. It will often save preferences across Sibelius sessions.

A plugin dialog stores its settings in some global variables, one variable for each control. Potentially, these could be True-False Booleans for checkboxes and radio controls, or text strings for edit boxes and single-selection listboxes, and arrays of strings for multi-select listboxes.

However, there is a command available to **Execute Commands**, called **RunPluginEntry_cu** which is effectively a universal child plugin, and it is much easier to use it if the parameters in the Dictionary are all text strings.

So I recommend that parent plugin dialogs be set up using edit boxes or list boxes as much as possible. Use a list box instead of radio buttons, and if you really need check boxes, I suggest storing values like “yes” for True and “no” for False for these controls, as the text versions of “True” and “False” can be problematic.

My current Parent/Child design

I have published the template Parent/Child plugins **MinimumPluginParent** and **MinimumPluginChild.**

In a **Parent/Child** plugin, all the settings used by the dialog will be stored into a single structure (in my current design this is a Dictionary) right after the dialog would have come down had it been shown.

The advantage of using a dictionary, as opposed to using an array or Sparse Array, is that you can give each dictionary entry a name that reflects the contents, and thus do not have to copy the variables into named local variables, or use index references when you need to use the data to figure out what a specific parameter is to be used for.

Here is the code from **GetDictDialogProperties()** in my plugin **Minimum Plugin Parent.** This routine sets the dialog property values (which have **dlg_** prefixes) into the dictionary.

I like to use prefixes for variable names that indicate the nature of the data, so I am using **str_** prefixes for the fields. This way I know to expect them to be text strings and not Booleans or arrays.

There is no requirement to do this, and if you do, you really need to use the convention consistently, but I find this extremely useful when debugging or modifying code.

```
dict = CreateDictionary();
// Avoid using booleans in the dictionary. Here I map True to yes and False to no
if (dlg_fCheck1)
{
   dict["str_Check1"] = "yes";
}
else
{
   dict["str_Check1"] = "no";
}

if (dlg_fCheck2)
{
   dict["str_Check2"] = "yes";
}
else
{
   dict["str_Check2"] = "no";
}

dict["str_RadioOption"] = ("" & dlg_strRadioOption);
dict["str_Edit"] = ("" & dlg_strEdit);
dict["str_SeasonSelected"] = (dlg_strSeasonSelected);
```

In my template plugins, the externally callable plugins are called with a single parameter, which is a Dictionary data structure. The values in the Dictionary are nearly always text strings, **except for 2 variables which as always passed in, which are a score and a selection object**, usually the current active score and selection. These must use keys “score” and “selection” respectively.

In the main processing routine (**API_ProcessSelection**) the parameter **dictSettings** includes the data from the dialog as shown above, but it must also include “score” and “selection”.

The important things about the main processing routine (API_ProcessSelection) is that it cannot use any global variables except those that it creates itself. Any data it needs must be passed into it as a parameter.

API_ProcessSelection needs to be callable from another plugin without messing up the internal state of the parent plugin. All data the routine needs must be passed into it or created by it. Any variables passed in **dialogSettings** need to be copies of the dialog data, not links to it, so that changing the data will not affect globals in the parent plugin.

As long as this separation is maintained, then child plugins can call in with their own copy of a **dialogSettings** structure.

In my template files for child-parent plugins (**MinimumPluginParent** and **MinimumPluginChild**), the child can call the parent plugin routine

API_GetDictCurrentDialogProperties to get a dictionary containing all the dialog variables, but this is not required. The child can create its own dictionary, fill it, and then call **API_ProcessSelection.**

A warning about passing Booleans in dictionaries.

Booleans (True/False) are different from the text "True" or "False". **val = "True"**; is fundamentally different from **val = True;**, and they test differently. I suggest using Boolean values rather than text string for the values of Booleans if possible. If you must use a string, I **strongly** recommend passing in "1" and "0" instead of "**True**" and "**False**", because **if (str_1)** will resolve to True, but the text equivalent "True" will not.

It is perfectly possible to pass Booleans in a Dictionary. However, since the **cmdutils** command **RunPluginEntry_cu** has to parse a text string containing variable values it will store into a Dictionary, it could fill the dictionary with "True" or "False". I actually have **RunPluginEntry_cu** convert such text to Booleans, but not all Children will be so careful, so I recommend avoiding using Booleans as values in the Dictionary.

I have decided that, for any Parent plugins I create, I will use a list box instead of radio buttons when possible, to both reduce the number of variables I need to deal with, and to reduce the number of Boolean variables passed around. If I really need to use checkboxes I am using the text values "yes" and "no" to avoid having problems with code like if (str) failing when str could be the text string "True". Instead I am forcing the code to explicitly say if (str = "yes").

Passing the dialog settings to the Child

A child plugin can get a dictionary containing the CURRENT dialog settings, using the same code the Parent uses for itself. The advantage of this is that the child gets a structure set up with field names (which they can easily trace) and it could save time in setup.

When the child fills out the structure, **I recommend setting every field explicitly** so the results do not depend on the last change the user made in the dialog, unless that is something you want to capture.

The parent plugin method API_ routine, **API_GetDictCurrentDialogProperties(pDictIn)** can be called by the child plugin. Like **API_ProcessSelection,** it is passed in a dictionary containing the score and selection, plus (in **my** templates) a variable called "**str_Trace**", which should be a string with a value of "yes" or "no" If (str_Trace = "yes"), the names of the dictionary fields and their current values will be written to the trace window, which is useful when you are setting up the child call.

Passing the dialog settings to the Parent

The child must either use the dictionary from **API_GetDictCurrentDialogProperties**, or create its own, and fill in the required dialog property entries, plus the "score" and "selection" entries. It will then call A**PI_ProcessSelection** or an equivalent routines, passing the filled-in Dictionary as its own parameter.

You can install the template plugins **Minimum Plugin Parent** and **Minimum Plugin Child**, and make copies of them and change the parent dialog and other code.

You can use the **cmdutils** command **RunPluginEntry_cu** to create **Child** plugins for any **Parent**, with no coding required. This is discussed in great detail in the document **RunPluginEntry_cu and Parent-Child Plugins**.

Otherwise, this pretty much covers the concept of Parent/Child plugins.