From: The Unofficial Sibelius Wiki - your source for tips and tricks to the world's very best music notation software! [This site appears to be defunct as of 2025]

Plug-In Authoring For Dummies by Ed Hirschman

Table of Contents

Introduction

If you'd like to write your own plug-ins, it helps to have some sort of programming background. The official Sibelius reference manual is located here (recently updated for Sibelius 6) .
Unless you are a skilled programmer or are prepared to spend many hours studying and working the examples, you likely will not get very far.

There is a ManuScript developers' mailing list for those interested in getting help on writing plug-ins. To subscribe, send an email to majordomo@sibelius.com with the words "subscribe plugin-dev" in the body of the email (subject line must be blank). You will get an auto-reply back saying that the request "has been forwarded to the owner of the "plugin-dev" list for approval."

Several key learnings below are based on the ideas and writings of Bob Zawalich.

Overview

This online guide is intended to help accelerate the learning curve for authoring Sibelius plug-ins, especially for people whose programming skills may be nascent or rusty. If you have no programming skills whatsoever, you would likely need to gain some basics at another website in order to get the prerequisite skills and knowledge. An example of this (I have not tried it but it looks like what you would need) would be here

The other goal here is to provide a framework on how to approach writing a plug-in. One possible framework is to follow a generic SDLC (Software Development Life Cycle) which lays out the following steps:

1. Requirements
2. Design
3. Development
4. Testing
5. Deployment

Requirements

This is the step where you will initially define what you want the plug-in to do. It is a good idea to write these down in some sort of list in Excel or Word with some sort of prioritization to each item. I use the following simple prioritization scheme:

- P1 - Must be there

- P2 - Should be there
- P3 - Nice to have have

In addition, try to write what a finished plug-in announcement would look like on the Sibelius plug-in page. These helps to reach clarity and precision in what the plug-in is supposed to achieve. Note that the items you initially planned on doing may change as you move through other steps and see what is or isn't possible. You may also find important items that should have been included but were not in your initial list.

If you plan to share the plug-in with others, it is important to plan for making the plug-in more robust, such as providing more explanatory dialog boxes, error checking and graceful exiting upon errors.

Before you complete this step, you should check a few things:

1. Is there a function within Sibelius that already (reasonably) does what I am trying to do? You may or may not be aware of these capabilities so read the documentation, check the chat board and experiment to make sure you are not re-inventing the wheel (and wasting your time).
2. Is there a built-in or user authored plug-in that that already (reasonably) does what I am trying to do? Or perhaps one exists that I can quickly tailor to my needs?
3. Does the situation I'm looking to address with my proposed plug-in happen often enough that it justifies my time to write a plug-in? Is the manual effort low enough within Sibelius that I shouldn't bother?
   These two questions should be answered together in order to decide if it is worthwhile to proceed to the design step or if you should just stop right here.

Design

In the design phase you performing some due diligence to see if the requirements you've written can be fulfilled by a plug-in. Although ManuScript is a powerful scripting language that matures with each release of Sibelius, not all aspects of the program are accessible. Examples of things you *can't* do with ManuScript are (not comprehensive):

- Access the print dialog boxes
- Easily find the elapsed time when repeats and other bar jumps are involved

To validate that what you want to do can be done, it is a good idea to look through the ManuScript manual and/or other plug-ins for the various functions or examples that relate to your area of interest.

As part of your design, you may want to create some pseudocode as a guideline for the development of the code. The design should, at minimum, deliver on the P1 functionality described in the requirements phase. The design should be modular for ease of testing, troubleshooting and maximum reuse in the future. In layman's term this means that you don't want to write many complex operations into a single “Run” method. Plan to create a method for each major function and “call” these as needed.

Development

Development Workflow

There are two basic choices for the development environment to use, or you can use a combination. Each environment has its pro's and cons.

Choice 1 - The IDE built into Sibelius.

The allows you to create the methods, dialog boxes, etc. and has a rudimentary syntax checker.
Pros

- Simple…just fire up and go
- Superior way to create/edit dialog boxes (Windows only)
- Built in syntax checker
- Can test within same environment

Cons

- Need to restart Sibelius if you want to load new plug-ins
- Does not have any search/replace capabilities
- Can be cumbersome for big or complicated plug-ins

Choice 2 - A text editor.
Since the *.plg files are basically text files in UTF-16-le coding, you may use a text editor partially or exclusively to create your plug-ins. The internal file structure is a 'TreeNode' representation. It is sensitive to certain line breaks and some special characters like double quotes.
Pros
- Easy to copy parts or whole plug ins to begin creating new ones

Cons
- if you are not careful with your syntax, it is easy to create plug ins that don't load at all within Sibelius.
- It is more difficult to create dialog boxes in a text editor than by using the IDE.

For Macintosh
- BBedit
- TextMate
- TextWrangler

For Windows
- Windows notepad
- NotePad++

When using a text editor, be careful how you use double quotes in the body of text, as in
str = “This string has double quotes”;
If you do this the plugin will be unreadable. What you need to do is use single quotes:
str = 'This string has double quotes';
because the double quote character has structural meaning to Sibelius (end of file).

Choice 3. Hybrid (Bob Z Method)
- Open the plugin in Sibelius IDE
- Copy the method into a text editor and work on it doing replacements and other changes
- When done, paste text back into the IDE
- Test within Sibelius

Development guidelines & Coding Standards

- Inserting comments with your code is always a good idea
- Insert a copyright notice as a comment to identify yourself as the plug-in author in the first line of the “run” method. Something like:
  - Copyright 2009 John Doe. All rights reserved.
- In Sibelius 5, there was a quirk in ManuScript that you can mess things up in a “for each” loop if you are looking for things and then add (or worse) delete while you are in the “for each” loop. Now in Sibelius 6, you **MUST** build a “pseudo array” and then deleting (or adding) while processing the pseudo-array.
- Variable naming conventions
  - Prefix normal globals with g_
  - Prefix globals used as dialog variables with dlg_
  - To make globals easy to find, prefix them with zg_ which puts them at the end of the list
- Text strings, in general, should be prefixed with _. This is a requirement for any plugins that Sib publishes - their translation tools look for the underscore. It is a bit tedious to put all strings in global data, but having it this way saves a lot of work later.
-

Suggested plug-in structure

Assuming that you've convinced yourself that your plug-in can be written, I would advise you to follow some of the design guidelines that Bob Zawalich has provided:

- Code that does the actual work should put it in a method called “ProcessSelection”
- A standard “Run” method can be used unchanged as a template for all similar similar plugin.
- The plug-in name is put into the “_PluginMenuName variable”

You can download a generic plug-in file that follows Bob's recommended structure here genericplug-in.plg. The “ProcessSelection” method is empty except for a comment, so this plug-in doesn't do anything until you add some real code there.

Testing

The IDE allows you to perform tests on your plug-in. It is good to create some test scores that you can use to ensure that your plug-in is working. test that everything defined in the requirements section works as expected.

Plug-ins should be tested on Macintosh and Windows before considering the testing completed. Most people have just one platform, but willing volunteers can usually be found on the forum that have the other platform and can validate that it works on the other platform.

Deployment
You can store your plug-ins in a custom folders. A possible list of new folders within plug-in folder could be:
- “rough” - plug-ins that work but don't have all of the niceties of a “finished” plug-in
- “test” - plug-ins that are in development
- “final” - - plug-ins that completely done and have all of the niceties of a “finished” plug-in

You may also post your plug-ins on this wiki here.

Deployment checklist
- Set the filename the same as the plug-in name, except without spaces and with ”.plg” appended to the end of the filename.
- The filename may contain upper and lower case letters.

- If your plug-in is well written and that others can benefit from it, you may submit it for consideration on the Sibelius plug-in page. If you would like this to be published on the official Sibelius plug in page, zip the plug-in first and email it to daniel.spreadbury@avid.com, Sibelius Senior Product Manager. It generally goes more smoothly if you provide a well-written description that can be posted without additional editing. See the plug-in download page for examples.

- You can describe your plug-in on the Sibelius forum, but since the *.plg extension may not be attached there, you can not share them via the forum with the rest of the Sibelius community.

Troubleshooting

Calculation in manuscript is giving wrong result

Yes, this is a well-known issue with ManuScript. If you multiply or divide and want a floating point result, at least one number must be a non-integer. So if you are using a constant, append ”.0”. If you are dealing with variables, multiple one of them by 1
Remember that evaluation of expressions is totally left to right, so *ALWAYS* use parentheses in expressions. Do not say x = 3+4 * 5, but x = (3 + 4) * 5.

Plug in does not appear in menu

Ensure that your plg file contains an initialize method incldudng
AddToPluginsMenu(PluginMenuName,'Run');

Data stored as globals is not saving between Sibelius sessions

Globals are not saved between sessions. You will need to use a preferences file using via the preferences plug-in. See p30 of manuscript manual v6.2.

The plug-in runs fine on Windows, but when I test on Mac, it won't save files correctly (or vice versa)

If you are running a demo version of Sibelius on the “other” platform (e.g. Mac), the demo version does not allow saving of sib files or text files. Have someone with a full copy of Mac test your plug-in...it's probably fine.

utils.ExactFileName method is causing an error

This is caused by a typo in the v6.2 manual on p 123. Instead of: utils.ExactFileName It should say utils.ExtractFileName

I copied a new plug-in to my computer but it is not appearing in the plug-in menu

You need to relaunch Sibelius after the plug-in is in place for it to be picked up. Then you can run it from Sibelius's Plugins menu.

I overwrote a plug-in with another of the same filename and plugin name, but Sibelius still uses the old one

Even in this instance, you need to relaunch Sibelius for the new file to be recognized.

What is the rightmost position in any bar?

position = 32256;

Guidelines for Variable, Method and Dialog Naming in ManuScript

As a new plug-in writer, and someone who doesn't do programming for a living, I was looking for some coding guidelines that would help me to create plug-ins that are easy to understand, troubleshoot and maintain. I wasn't able to find quite what I was looking for, so I studied several of Bob Zawalich's plug-ins to derive some guidelines to follow. I have reviewed this with Bob and he has provided valuable additions and clarifications to what I was able to put together.

I am not implying that this is the best guideline there could be, but since Bob is the most prolific plug-in author, his style is the most prevalent out there for plug-ins not written by Sibelius staff. More importantly, it is logical and gets the job done. Please note that this scheme is not endorsed or sanctioned by Sibelius.

These guidelines don't cover every aspect one might wish for. As such, you are encouraged to contribute something (by editing the Wiki page) that you think that the plug-in development community could benefit from.

Prefixes for Global Variables

Variable Prefix Type	Examples	Used for	Note
dlg_Xxxx	dlg_lstAvailableParts dlg_fInstallPlugin dlg_strEditFolder	dlg_ for variables that are used in a dialog structure	
_Xxxxx	_ExistsText _PluginMenuName _captionSorting _msgNoneFound	Global text of a nature that would require translation	A Sibelius standard. Variables starting with underscore are triggers to Sibelius to substitute in alternate foreign language text when used with foreign language versions. Be careful to follow this

		for foreign language versions of Sibelius	conversion for text to be translated from the start, so if Sibelius wants to buy your plugin and ship it, you don't need to rewrite it to comply. For underscore globals, the convention that existing plugins use is to have some descriptive text starting with a capital letter, such as _InitialText, or _PluginMenuName. _PluginMenuName might be improved from a naming standpoint if it was called as _namePluginMenu, but there are many precedents of this naming, so it is not worth worrying about.
g_Xxxx	g_fDoTrace g_strExistsText g_valPreferencesOpen g_arrObjectTypesWithStyles	Globals not in a dialog and not subject to translation.	Use g_ for other globals not covered by the above prefixes Types, and I try to minimize these. Often these will be things that dialog routines will need to know about, and you cannot pass parameters to such routines.
zg_ Xxxx	zg_PreferencesVersionNumber	A global I might need to change often, so it forces it to the end of the items in the dialog editor	A global I might need to change often, so alphabetical sorting forces it to the end of the items in the dialog editor Data panel.

Local & Global Variables

Functional Types	**Used for**	**Note**
i	Index	Integer, usually starts at 0
arr	ARRay	
f	a boolean Flag (0/1 or true/false)	
str	STRing	Example, strNameInst - a string holding an instrument name
lst	for the contents of a LiSTbox or combobox.	
nr	NoteRests	
n	notes	nStavesUsed
s	staff	
barnum	bar numbers	A Sibelius convention. Correct – barnumFirst Incorrect - "barNum", barNumFirst, firstBarNumber
val	for numerical VALue	– not for 0/1 (use f for that), these are for storing constants, not values that change e.g. valPi = 3.14
caption		for what would appear in a progress dialog box
msg	MeSsaGe	shown to user in a dialog box

Notes and Guidelines for Variables

- Variable naming - Bob uses a form of "Hungarian", a technique developed by Charles Simonyi at Microsoft. He believed that good naming leads to better code, and that having a system for naming saves a lot of programmer work.

- “Functional Types” are in all lower case and are used to start the variable name, therefore all variables start with a lower case letter. The Functional type is usually followed by a descriptive noun, and possibly followed by an adjective. Examples:
    - “nStavesUsed”
    - “strNameInst”, a string holding an instrument name.
- Names after the Functional Type begin with an upper case letter and follow Camel Casing convention.
- If the variable is only used locally in a small routine, the Functional Type may suffice for the variable name, such as using “i” as the variable name for the index in a “for” loop.
- These rules are frequently broken, by the way, especially when interacting with pre-existing code, but in general this is the model to use.
- If a new Functional Type is needed, feel free to create a new one.

Methods

- Method names all begin with capital letters, and are mainly unstructured descriptions of functions.
- Methods begin with an upper case letter and follow Camel Casing convention.

- Examples
    - BuildVersionText
    - CheckExistingFile
    - GetPreferences
    - MyYesNoMessageBox

Dialogs

- Dialogs all begin with capital letters, follow Camel Casing, and are mainly unstructured descriptions of the Dialog boxes.
- Dialogs end with the word “Dialog”
- Examples
    - FileExistsDialog
    - RestoreDialog
    - HelpDialog