Plug-in interface for Svarog Tutorial

Marcin Szumski

February 19, 2011

Contents

1 Introduction

Plug-in interface was created to allow the development of Svarog without the necessity to change its code. In the long term it will increase the modularity of the project, as the code of the newly created plug-ins will be independent from most of the Svarogs' code. It should also result in a better stability, because it will be easier to locate in which plug-in the error occurs and simply switch the plug-in off.

There will be also a change from the user point of view, as he will be able to decide which extensions (from the set of plug-ins that may even exclude one another) he would like to use.

This tutorial is written with a lot of details, which for some developers may seem unnecessary or even foolish. It is done on purpose, to make the creation of plug-ins possible also to non-advanced developers or simply those who don't know Java and Eclipse very well. Moreover in this tutorial it is assumed that the plug-in is created in the Eclipse IDE. Although anyone, who got accustomed to a different IDE and would like to use it shouldn't have a problem.

Below is a short schema, which operations the developer should perform to create a plug-in:

1. create a new Java Project (section 2),
2. add libraries to the build path (section 3),
3. create a package and a starting class (section 4),
4. implement function `register(SvarogAccess)` (section 5),
5. write the functionality of the plug-in,
6. export the project as a `jar` file (section 6),
7. create an XML description (section 7),
8. put both files in the plug-in directory (section 8).

2 Creation of the project

Go to: `File -> New -> Project...`
Select: `Java -> Java Project` and press `Next >`
Type some custom `Project name` (in our example it is `ExamplePlugin`)
Press `Finish`

3 Adding libraries

Next step is to add libraries to the build path:

- Svarog snapshot (`svarog-\{version\}.jar`) - the necessary library, which contains the Svarog classes (e.g. plug-in interface). Can be found in the directory `svarog/target`.
- Apache log4j (`log4j.jar`) - add this library if you want to use logging framework consistent with the one used in Svarog. This library can be found in the directory

  `svarog/target/svarog-\{version\}-full/svarog-\{version\}/lib`

- Spring Context (`spring-context.jar`) - add this library if you want to use classes that inherit from `MessageSourceAccessor` (e.g. `Tag`, `TagStyle`). This library can be found in the directory

  `svarog/target/svarog-\{version\}-full/svarog-\{version\}/lib`

In order to add libraries you need to:

- copy selected libraries to the project directory,
- open `Package Explorer`,
- right click on the project folder,
- select `Build Path -> Configure Build Path...`,
- open tab `Libraries`,
- click `Add JARs...`
- expand project folder, select copied libraries and press `OK`,
- again press `OK`.

4 Creation of the starting class

In this section we are going to add a starting class that implements the interface

`org.signalml.plugin.export.Plugin`

At the beginning let's create the package:

- right click on `src`, select `New -> Package`,
- set the `Name` to `org.signalml.plugin.yourpluginname` (replace `yourpluginname` with the name of your plug-in),
- press `Finish`.

To the package add your starting class:

- right click on the created package, select `New -> Class`
- set the `Name`,
- click `Add..` to add an interface,
- type `Plugin` and click `OK`,
- press `Finish`.

5 *register(SvarogAccess)* implementation

`Plugin#register(SvarogAccess)` is the only method that has to be implemented. This method is called just after your plug-in is loaded and without it your plug-in won't be able to communicate with Svarog (as only to this method the `SvargoAccess` instance is passed). Here should be :

- added GUI elements (as adding them later won't be possible) (section 9.1),
- registered listening on changes (section 9.2),
- performed all other operations that initialize the plug-in.

6 Exporting project to jar file

When your plug-in is already created you have to export it to the `jar` file. To do it:

- Open `Package Explorer` and right click on the project folder.
- Click `Export`.
- Select `Java -> JAR file` and click `Next`.
- Make sure your plug-in is selected, choose the export destination and press `Finish`.

Note that Eclipse sometimes hangs during the export. The best way to deal with it is to restart it.

7 Creation of an XML file

Now it is the time to create an XML file, which describes the plug-in. This file should contain some basic information about the plug-in. Perhaps the easiest way to understand it is studying an example (figure [7]).

And here is the description of functions of xml tags:

- `<name>` - the **unique** name of the plug-in.
- `<version>` - the version of the plug-in in the form of non-negative integers separated by dots. The significance of the numbers is in the decreasing order (i.e. $2.1 > 1.9$).
- `<jar-file>` - the name of the `jar` file with the plug-in. This jar file must be located in the same directory as the description (XML file).
- `<starting-class>` - the fully-qualified name of the class that implements `org.signalml.plugin.export.Plugin` interface. The method register of this class will be called to load the plug-in.

```
<plugin>
  <name>
    Example plugin
  </name>
  <version>
    0.1
  </version>
  <jar-file>
        ExamplePlugin.jar
  </jar-file>
  <starting-class>
    org.signalml.plugin.exampleplugin.ExamplePlugin
  </starting-class>
  <dependencies>
    <dependency>
      <name>
        Svarog API
      </name>
      <version>
        0.5
      </version>
    </dependency>
  </dependencies>
</plugin>
```

Figure 1: ExamplePlugin.xml

- `<export-package>` - the full name of the package that this plug-in wants to share with another plug-ins. This field of the XML file is not used by Svarog and serves only as an information for developers of other plug-ins (so that they can use some functions of this plug-in).
- `<dependencies>` - the parent node for all dependencies (see next point).
- `<dependency>` - describes the single dependency of this plug-in (another plug-in (or Svarog) that is required by this plug-in; if at least one dependency is not satisfied, this plug-in won't be loaded).

  Each dependency contains two parameters of the plug-in (or Svarog) on which this plug-in depends:

  - `<name>` - the name. Equal to the specified in the description of the plug-in.
  - `<version>` - the minimum version. The current version of the plug-in must be greater or equal to the specified here.

  There is one special dependency that can be added - the dependency on Svarog Plugin API:

  - `<name>` - `Svarog API`
  - `<version>` - the version of Svarog Plugin API - currently 0.5

8 Moving plug-in to plug-in directory

Copy both exported the `jar` file and the created description to the plug-in directory. The default plug-in directory is located within your Svarog profile directory. Now you can start Svarog and you should see your plug-in on the list ☺

9 Description of the elements of plug-in interface

Below you will find the short description of functions provided in the plug-in interface. Note that this is only a short introduction and if you want to learn something more you have to read JavaDoc ☺.

Before we proceed to the description please read this two general remarks:

- when you want to perform operations that require intensive computations use `SwingWorker`[1]. This class allows you to transfer computations outside *Event Dispatch Thread*, so that you won't hang the entire GUI.

[1] http://download.oracle.com/javase/6/docs/api/javax/swing/SwingWorker.html

- with the plug-in interface are shipped some classes that are not necessary, but you may find them useful: `AbstractDocument`, `AbstractSignalTool`, `SignalSelection`, `SignalSelectionType`, `Tag`, `TagStyle`, `AbstractDialog`, `AbstractPopuDialog`, `AbstractTreeModel`. For their description see JavaDoc.

9.1 GUI elements

There are four types of GUI elements you can add:

1. buttons - must implement interface `javax.swing.Action`. To add them just call method `addButtonTo*(Action)`.
2. sub-menus - must be of type `javax.swing.JMenu`. They can not contain sub-menus as the actions from them are copied to create menus e.g. for different signal plots. To add them just call method `addSubmenuTo*(JMenu)`.
3. signal tool - it is a mouse event processor associated with a single SignalView instance. To implement it you have to know a little more how Svarog works. The easiest way to create it is to extend `AbstractSignalTool`, but if you do it you have to override method `createCopy()` (and of course not throw `UnsupportedOperationException`).

   For more information how the Signal Tools work, see some of the classes (`RulerSignalTool`, `SignalFFTTool`, `Tag*SignalTool`, `Select*SignalTool`). Adding a signal tool is done by calling the function `addSignalTool(SignalTool, Icon, String, MouseListener)`, which adds to every SignalView, a button created from the provided elements and a COPY of provided signal tool.
4. tabs - tabs can be added to 3 panes:
   - main (center) pane - have to have type `DocumentView`. Should be also associated with some `Document` (which can be obtained from the view by method `getDocument()`). These tabs are added by calling a function:

     ```
     addMainTab(DocumentView tab, String title, Icon icon,
                 String tip)
     ```
   - tree (left) pane - have to have type `ViewerTreePane`, and therefore a `JTree` associated with them. Can be added by calling:

     ```
     addTreeTab(ViewerTreePane treePane, String title,
                       Icon icon, String tip)
     ```
   - property (bottom) pane - they only have to have type `JPanel`, and there is no restrictions about them. Can be added by calling:

     ```
     addPropertyTab(JPanel panel)
     ```

The addition of first 3 (buttons, sub-menus and signal tools) may be performed only during the initialization phase, i.e. only in the function `register(SvarogAccess)`. Tabs can be added (and removed) at any time during the runtime of Svarog.

9.2 Listening

Listening for changes in Svarog is pretty simple, as it requires only to create a class, that implements a given listener and to add this listener with methods from `SvarogAccessChangeSupport`.

When the change, for which the added listener is listening occurs, the appropriate method in this listener is called (e.g. if the `Tag` is added the method `SvarogTagListener#tagAdded` is called). To this method is passed an event, which contains the objects that have changed.

Below are the possible types of listeners:

- SvarogCloseListener - waits for the information that Svarog is closing
- SvarogCodecListener - waits for the information that the codec was added or removed. The event for the methods in this listener contains the format name of the added or removed codec.
- SvarogDocumentListener - waits for the information that the document was added/removed form Svarog, the active document has changed or the `DocumentView` for the document changed. The events contain:
  - for fist two methods (addition and removal) - only the added/removed document,
  - for the third method (change of a view for a document) - the document and the old value of the changed view (as the new value can be obtained from document).
  - for the fourth method (change of the active `Document`) - old and new value of the active document.
- SvarogTagDocumentListener - waits for the information that the active `TagDocument` has changed. The event for the method in this listener contains the old and the new value of the active tag document.
- SvarogTagStyleListener - waits for the information that the `TagStyle` was added, removed or changed. The event for the methods in this listener contains the changed tag style.

  This listener can be added to listen for all tag style changes or to listen for changes associated with the given tag or signal document.
- SvarogTagListener - waits for the information that the `Tag` was added, removed or changed. The event for the methods in this listener contains the changed tag and the document in which the tag is located.

  This listener can be added only to listen for changes associated with the given tag or signal document.
- SvarogTagListenerWithAcitve - it is the extension of `SvarogTagListener`. Waits also for the change of an active tag. The event for the method associated with the change of the active tag contains the old and the new value of the active tag.

9.3 Documents, samples and tags

`SvarogAccessSignal` is an interface which allows to get from and add to Svarog some logical objects, namely:

- Get samples from the signal - samples are available:
  - either for the active signal or for the signal from the given document,
  - either from a single channel or from all channels in the signal,
  - either processed (after the montage and filtering) or raw (unprocessed).
- Get tags - available are:
  - tags from active or specified tag document
  - tags from all tag documents dependent from the active or selected signal,
  - tags from all tag documents
  - the active tag
- Add tags to the active or specified document.
- Open:
  - a book or a signal file and add it as a tab,
  - a file with codec and register a new codec,
  - a file with tags and add it to the active or specified signal.
- Get active elements:
  - `Document` - the document that is associated with the main tab,
  - `TagDocument` - the selected document with tags for the active signal,
  - the tag,
  - the selection of the part of the signal.

10 Plug-in options (in Svarog GUI)

The plug-in options in Svarog GUI allow to:

- specify directories in which plug-ins are located,
- select which plug-ins should be active,
- check which plug-ins were not loaded because of an error,
- check which plug-ins have unsatisfied dependencies and see the list of these missing dependencies.

All accepted changes will take place after the restart of Svarog.

To show these options you need to go to `Tools -> Plugins options`. Dialog shown in figure 2 should appear.

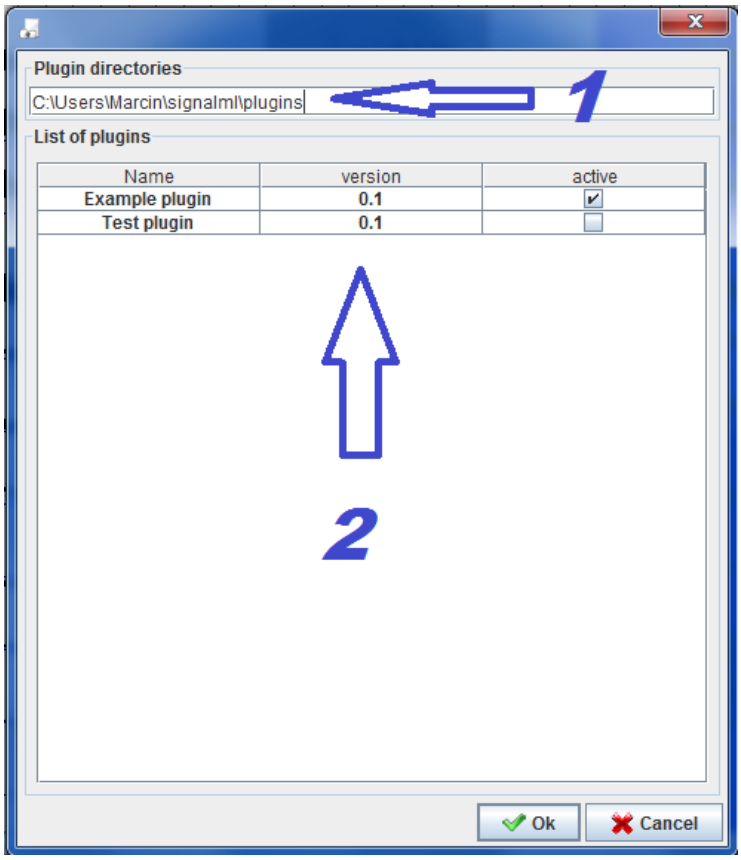

Figure 2: The dialog that allows to manage plug-ins. In the text field in the panel **1** are paths to the directories in which the plug-ins are stored separated by semicolons.
In the table **2** in every row one plug-in is displayed. Third column of this table is editable, which allows user to select which plug-ins should be active.

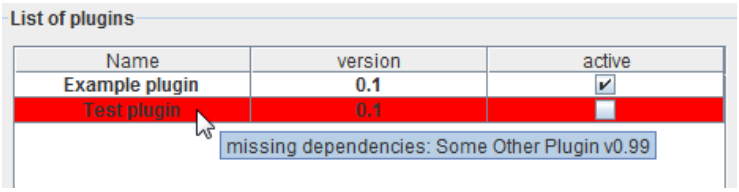

Figure 3: If the plug-in is not loaded due to an error or missing dependency, the row with this plug-in is marked red and can't be edited. Also if it is a missing dependency the tool-tip for this row, which enlists the missing dependencies, is set.
Here: plug-in TestPlugin has a missing dependency - *Some Other Plugin*, which should have version at least 0.99.