Graphical User Interfaces Composed of Plug-ins

Johannes Mayer

Department of Applied Information Processing and Department of Stochastics,
University of Ulm, D-89069 Ulm, Germany
jmayer@mathematik.uni-ulm.de
http://www.mathematik.uni-ulm.de/sai/jmayer/

Abstract. For programmers who are not involved in the development of a graphical user interface (GUI), it is usually not an easy task to adapt or extend the GUI. In this paper a possible solution being based on the plug-in concept is proposed which makes extensions of the GUI and independent development of GUI components much easier. Furthermore, GUIs designed that way can be easily extended with individual functionality for each user.

Keywords: GUI, plug-in, development method
Classification: Second year's PhD work

1 Introduction

"Traditional" graphical user interface implementations have certain disadvantages for the developer:

1. The individual source files of a GUI usually have many lines of code, e. g. the source file with the main menu (and the corresponding commands). Therefore, the effort to add new functionality is greater for programmers who were not involved in the GUI development, since the appropriate place for the extension must be found.
2. The structure of the GUI (software) design is rather complex. This makes extensions or adaptions which involve changes in several parts of the GUI costly, since basic knowledge about the design must be acquired first.
3. An extension of the GUI cannot be done without changes in other modules, since every extension must be called — directly or indirectly — from a main GUI module. When the changed modules have already been tested, their tests are invalidated.
4. Third party extensions are currently only possible for some tasks through plug-ins (cf. e. g. [1, 4]).

Visual development tools, such as Sun's Forte for Java and Borland's JBuilder — in the case of Java —, do not really help to solve these problems. They only reduce the initial coding effort.

In the GeoStoch project a Java library for image analysis and image processing applying methods from spatial statistics and stochastic geometry is developed at the University of Ulm. The first version of the GUI developed for this library was a "traditional implementation" and thus, the above mentioned problems were experienced. The mathematicians who contributed new functionality to the library and wanted to add it to the GUI needed up to two days only for the (simple) GUI part. The reasons were that the GUI design was so complex, the modules had many lines of code, and the mathematicians were not involved in the development of the GUI. With the new plug-in-based GUI, it takes them by far less time than before to do this job.

Plug-in-based application development, as proposed in [3], can be tailored to the design of GUIs. This work shows how this contributes to the solution of the above mentioned problems.

After a short explanation of the plug-in concept in Section 2, various possible uses of plug-ins in the development of GUIs are presented in Section 3. In Section 4 related work is discussed. Finally, the contributions of this work are summarized and discussed in Section 5.

2 The Plug-in Concept

According to [3] (where the plug-in concept is explained in detail), a plug-in loader, a plug-in interface and concrete plug-ins are important parts of a plug-in-based application. Each concrete plug-in implements the plug-in interface which enables the communication between the application and the plug-ins. All available plug-ins are searched by the plug-in loader (at runtime) and made accessible.

Applying the plug-in concept to the development of applications, usually results in reduced complexity of the design as shown in Figure 1. The plug-

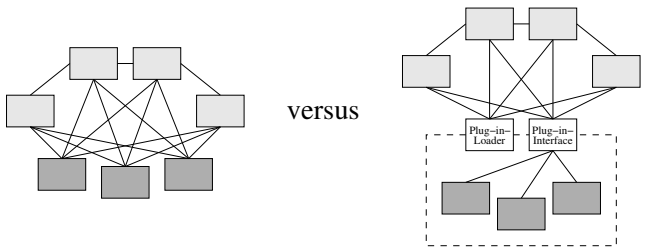

Fig. 1. The basic structure of the design with (*right*) and without (*left*) employing plug-ins. A greater number of (plug-in) components (*dark grey boxes*) even increases the differences between the design complexities. Dependencies are depicted by solid lines. And the facade is illustrated through the dashed box

in interface and the corresponding plug-in loader are the facade behind which

the concrete plug-ins are hidden. This usage of the Facade Pattern ([2]) helps simplifying the structure of the design. To develop further plug-ins, only the plug-in interface and the interfaces to the application's library must be known.

3 GUI Plug-ins

The plug-in concept is now applied to the development of GUIs, to tackle the mentioned problems. A number of *GUI plug-ins*, i. e. plug-in types, from which a substantial part of graphical user interfaces — but clearly not everything — can be built, are presented in the following. For each GUI plug-in, a generic plug-in interface and generic plug-in loader, which acts in this case also as a GUI element, can be given to assist the programmer. So a programmer must only derive an own plug-in interface with custom methods and write the functionality, i. e. the concrete plug-ins. As illustrating example, the well-known email program Netscape Messenger, which is not explained here, is used throughout this section to make explications better imaginable.

3.1 View Plug-ins

Applications typically should be open to future document types. Possible document types of an email program are, for example, HTML text, plain text, and images. It cannot be foreseen how another type of document can be displayed. So, a (GUI) panel which displays documents of a certain type is a good candidate for a GUI plug-in. Two requirements must be fulfilled: There must be a superclass for all documents, also future ones. Furthermore, the view plug-ins must provide a so-called "voting method". This method gets a document as parameter and returns whether this plug-in can display such a document. For each document which is to be displayed, the plug-ins are retrieved from the loader and all plug-ins are asked whether they can display such a document. If at least one plug-in can, one of them is chosen arbitrarily. Otherwise, a message is displayed, that such a document cannot be displayed.

3.2 Selection Plug-ins

Messenger offers different possibilities to retrieve mail, such as via the POP3 and IMAP protocol. The user has to select one of them and customize the selected protocol, e. g. an SSL connection may be used when the IMAP protocol is used, so this must be specified when IMAP is selected. For each protocol, a GUI plug-in (being a panel) can be designed which handles settings specific to this protocol. The name of the plug-ins, which can be accessed through a method call (to the plug-ins), are then displayed within a combo box, list box, or with radio buttons. When one gets selected, the panel implemented by this plug-in is displayed and the selected plug-in can be customized through user input (in the displayed panel). This may look like depicted in Figure 2a. Another possibility is that for the panel to be displayed a new tab within a tabbed panel is generated.

3.3 Preference Plug-ins

The first time, a user must enter information about his identity, such as name, email address, so that Messenger allows him to send mails. Therefore, a preference panel exists where such information can be specified. Furthermore, a view plug-in which displays HTML text may allow to customize colors and much more. This also requires a preference panel. Preference panels can easily be implemented as plug-ins. It is common, to use a tabbed panel for preferences. Each preference plug-in implements a panel which is displayed on an individual tab (cf. Fig. 2b). The name of this tab is the plug-in name which can be retrieved

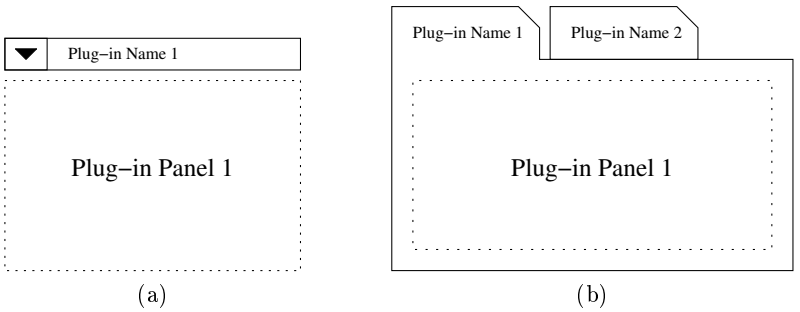

Fig. 2. Examples of GUI plug-ins: select plug-in (*left*) and preference plug-in (*right*)

from the plug-in through a method call. Instead of a tabbed panel, a (possibly hierarchical) list box is also often used for preferences, as in the case of Messenger.

3.4 Command Plug-ins

Messenger also has icon buttons, which offer frequently needed functionality, such as writing a new mail, retrieve new mail, and so on. These actions and much more can also be accessed through the menu. Furthermore, when clicking with the right mouse button, a context menu is displayed, which varies depending on the position of the mouse pointer, e. g. over an email text or the list of messages. For all these purposes mentioned, plug-ins can be used to implement a command as reaction on the click. This command may be arbitrarily complex, and can also involve that new dialogs are displayed. The text of the menu entry as well as shortcuts and icons for buttons are provided by methods of the plug-ins. Different context menus correspond to separate plug-in types, i. e. plug-in interfaces. Through configuration files, the menus may be pre-configured, i. e. the menu items may be named, sorted and connected with plug-ins. Further plug-ins which are found at runtime are added to the preconfigured menus. Thus, customization of menus only involves editing the configuration files.

4 Related Work

Grail[1] is an open source Internet browser implemented in the object-oriented programming language Python. On the website of this browser, the term "GUI plug-in" can be found. But GUI plug-ins are not explained there. This browser supports several plug-in types to extend the number of known file types, protocols, and HTML tags, and to add new preference panels. Preference plug-ins are here the first step towards a GUI composed of plug-ins.

Netscape plug-ins ([4]) are allowed to fill a part of the window or have a separate window to display certain MIME types. This is also a GUI plug-in, called view plug-in in the present paper.

5 Summary and Discussion

In the present paper a number of GUI plug-ins have been presented to ease the development of graphical user interfaces. Further possible GUI plug-in types are plug-ins for info dialogs, import and export filters, and maybe help.

The presented approach has proven useful for the development of the GUI for the GeoStoch library described earlier.

The plug-in concept helps to reduce the complexity of the GUI (software) design. Furthermore, modules which consist of many lines of code are not so common among plug-ins. The individual GUI plug-ins can be developed in parallel — even by third parties — only with the knowledge of the plug-in interface (and the library interface offered by the GUI application). Plug-ins do not require changes within other modules, since they are searched for at runtime.

However, it has to be pointed out, that the whole GUI cannot be modeled by plug-ins. Only components that are mutually independent and provide the same interface for the communication are candidates for plug-ins. For example, the main module of the GUI which uses the plug-ins, cannot be a plug-in. Another limitation of this approach is that it leads to a big number of GUI modules with not many lines of code. To keep the overlook, the modules have to be organized in relatively small packages.

References

1. Adobe Plug-in Component Architecture (PICA) – The Adobe PICA API Reference. Adobe Systems Incorporated, Version 1.1 3/97 (1997)
   `http://partners.adobe.com/asn/developer/graphics/docs/TheAdobePICAAPI.pdf`
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
3. Mayer, J., Melzer, I., Schweiggert, F.: Lightweight Plug-in-Based Application Development. NetObjectDays 2002 (to appear)
4. Plug-in Guide for Communicator 4.0. Netscape Communications Corporation (1998)
   `http://developer.netscape.com/docs/manuals/communicator/plugin/index.htm`

[1] `http://grail.sourceforge.net/`