Lightweight Plug-In-Based Application Development

Johannes Mayer, Ingo Melzer, and Franz Schweiggert

Department of Applied Information Processing, University of Ulm, Helmholtzstr. 18, D-89069 Ulm, Germany
{jmayer,melzer,srg}@mathematik.uni-ulm.de
http://www.mathematik.uni-ulm.de/sai/

Abstract. “Fat software” significantly reduces the effect of new and faster computer hardware. Such software is only possible due to the impressive success of the hardware developers. The main reason for this trend is the users’ demand for new gimmicks driving the software developers to include all possible features into their systems. Those features are loaded every time the program is executed and make the system bulky. The fact that most add-ons are simply integrated without a clear interface only adds insult to the injury. This paper shows a way to design software which helps to battle this development of bulky systems. The well-known plug-in concept is formally described as a design pattern. Based on this pattern a development principle is proposed. As a special case, GUI development is considered.

1 Introduction

The memory requirements of today’s applications have increased significantly. Ten years ago, computers were sold with 1 MB RAM and MS Word 5.0 required at least 384 kB memory, and today, the minimum requirements to start Microsoft Word XP on a computer running Windows XP are 136 MB of memory. One might be tempted to feel like “whenever I buy a new piece of software, I should also get more memory, or even a new computer”. One reason for this development of fast growing hardware requirements is, that the software industry has accepted Moore’s law of rapidly growing computer hardware performance. Another reason is that a lot of features are constantly added, which are at most nice to have. Programmers are often unable to remove old parts of software because no one knows whether and where those old parts are used.

Time pressure also prohibits a useful and lasting design — despite the fact that it is taught in almost every software engineering class. Wirth demanded in 1995 to keep software systems simple and load modules as they are needed [20] which is one way to solve this dilemma.

If the idea of the KISS principle, keep it small and simple, is carried out thoroughly, it leads to a modularized design with the replaceable parts implemented as plug-ins. Such an architecture also has the advantage of a low complexity based on a “higher level McCabe Metrics”. The idea of this metrics [9] is to
convert the code of a program into a graph on the basis of the listing's branchings. In a higher level version, the graph is not based on the code of individual modules or classes, but on connections among the modules or classes of the whole system. In other words, the different parts only communicate by means of a smaller number of interfaces.

This paper describes a way to develop applications employing the plug-in principle and extends this idea to the construction of graphical user interfaces.

In Section 2 the term plug-in is explained. Thereafter, a pattern for the plug-in concept is presented in Section 3 and a development principle for applications and GUIs is proposed (Sec. 4). In Section 5 techniques to determine subclasses at runtime are presented and discussed. Non-technical related work is then discussed in Section 6. Section 7 contains a discussion, conclusion, and hints for further work. Finally, a Java sample implementation of a GUI plug-in panel is given in the appendix.

2 Terminology

The plug-in concept is widely used (cf. e.g. [13]). For example, to view a PDF document within Netscape Communicator, it is sufficient to install a plug-in from Adobe which displays the document within Netscape's browser. But what is a plug-in? The CNET glossary contains the following entry for plug-in:

“This term refers to a type of program that tightly integrates with a larger application to add a special capability to it. The larger applications must be designed to accept plug-ins, and the software's maker usually publishes a design specification that enables people to write plug-ins for it. Two notable applications designed around a plug-in architecture are Adobe Photoshop and Netscape Navigator. Notable examples of plug-ins are Kai's Power Tools for Photoshop and Shockwave for Netscape Navigator.”

It is worth notable that a plug-in is unknown (and must not be used) at compile-time of the application for which the plug-in is designed. Therefore, nothing referring to a specific plug-in is hard coded into the application's source code. Dynamic loading of a plug-in is for this reason quite different from the well-known dynamic loading, where the dynamically loaded library is named explicitly in the executable file (the other case is covered in Section 5.3).

Furthermore, in contrast to stand-alone applications, plug-ins require the application they were designed for, although being deployable separately.

The proposal of this paper goes beyond the definition given in the above glossary entry. It is not only possible to “add a special capability” to an application through the plug-in concept, but to compose a whole application mainly out of plug-ins. A great deal of functionality which does not belong to the library is then implemented as plug-ins.