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,swg}@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 all 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 often no one knows whether and where those old parts are used.

Time pressure often 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, 10] is to convert the code of a program into a graph on the bases 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. (...)" [3]

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.

3 The Plug-in Concept

3.1 Motivation

Reuse of common elements is a well-known concept in architecture. In software engineering, reuse was restricted to code for a long time. The book of Gamma et al. [4] introduced reuse of design concepts. The so-called *design patterns* offer best practice solutions to common design problem. Advantages of the application of such patterns are that less experienced developers can improve their designs and designs being based on patterns are easier understandable.

Many *design patterns* have been presented to fit nearly all common needs. But, an adequate and detailed design pattern for the well-known plug-in concept is missing. This gap is hereby closed using the pattern template from [4].

3.2 The Plug-in Pattern

Intent

This pattern explains how to design an application in order to support the *plug-in concept* which allows an application being extended at runtime by dynamically loaded modules or classes not known during compilation (of the application).

Motivation

If, for example, an applications which is able to display a variety of different graphic formats is built, the developer might not be able to write a decoder for all future formats. This problem can be solved by allowing third parties to implement plug-ins which receive a stream containing the encoded information and return a decoded version of the stream. Such a plug-in is only loaded if an appropriate graphic format is demanded. This allows the application to start quickly, because at first, not a single decoder is loaded. At the same time, the required amount of memory is reduced.

Applicability

The plug-in pattern can be applied to fulfill the following requirements:

- Need for expansions during runtime, possibly unknown even at start-up
- Modularization of huge systems to reduce complexity
- Independent development of system components without modifying other modules or rebuilding the whole system
- Allow third party development based only on the knowledge of the interfaces.
- Allow easy deployment of new features and updates, after shipping the application.
- Short start-up time and low hardware requirements, especially memory (load features as needed)
- Create flexibility for long running servers which cannot be restarted.

Structure

In Figure 1 the classes and interfaces of the pattern are illustrated as an UML 1.4 [18] class diagram.

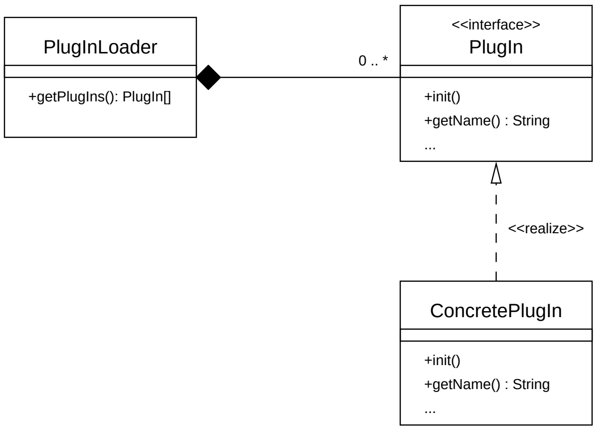


Fig. 1. UML class diagram for the Plug-in Pattern

Participants

PlugInLoader
- Searches for implementations of a PlugIn interface at runtime when needed.
- Possibly asks each concrete plug-in, whether it wants to be invoked.
- Grants clients access to all loaded plug-ins (via a call to the method getPlugIns()).

PlugIn (Interface)
- Provides the interface for the communication with all concrete plug-ins of the same type.
- Contains methods such as getName() to access the name of the plug-in, which may be displayed in a menu, for example.
- Possibly contains methods, so-called “voting methods”, which allow the plug-in to vote for or against its invocation in a given context.

ConcretePlugIn
- Implements the PlugIn interface and provides special functionality. For this purpose, library modules (of the application) — besides own supplementary classes — may be used.
- Possibly inherits from other classes and implements further interfaces.

Collaborations

- The PlugInLoader determines the names of the plug-ins via a call to getName() and allows to initialize themselves via init().
- A client of the PlugInLoader can access the plug-ins using methods of the plug-in interface and the plug-in loader.
- ConcretePlugIn possibly uses interfaces and classes provided by a library of the application.

Implementation

Technical issues in connection with the dynamic search for subclasses of an interface necessary for the implementation of the Plug-in Pattern are addressed in Section 5.

Sample Code and Usage

The *Plug-in Pattern* can be used to compose whole software systems as described in Section 4. Two examples of applications using the Plug-in Pattern are described in Section 4.3 and Java and Perl implementations are given in Section 5 and the appendix.

Known Uses

The popular image processing program Gimp[1] also allows the addition of new functionality through plug-ins. Furthermore, the SLC portal[2] and the GeoStoch[3] sample GUI, which are described in Section 4.3, are based on the plug-in concept.

Related Patterns

The *Pluggable Components Pattern* ([19]) is similar and discussed in Section 6.

The *Command Pattern* ([4]) is quite similar to plug-ins executing a command. The main difference is that plug-ins are not known at compile time. Their names, therefore, can — and must — not occur in the source code.

A concrete plug-in, i. e. the implementation of the PlugIn interface, is usually based on a number of classes, specifically used for this purpose. This is an application of the *Facade Pattern* ([15]), where the class ConcretePlugIn plays the role of the facade. Furthermore, the plug-in interface and the plug-in loader are the facade behind which all concrete plug-ins are hidden.

The method getPlugIns() from the PlugInLoader is similar to a factory method. The main differences are that getPlugIns() may return more than one instances and that the classes are searched at runtime by the plug-in loader whereas the class names are hard-coded in the implementation of a factory method.

Sometimes, it is desirable that plug-ins of a certain type are loaded at most once. Then, in order to guarantee this property, the PlugInLoader has to implement the *Singleton Pattern* ([4]).

4 Plug-in-Based Application Development

In the last section, a formal definition of the plug-in concept has been given in terms of a design pattern. But the plug-in concept is still more general. It is possible to develop whole applications based on the plug-in concept. This can support the development of lean software demanded by Wirth [20].

[1] `http://www.gimp.org/`
[2] `https://slc.mathematik.uni-ulm.de/`
[3] `http://www.geostoch.de/`

4.1 The Structure of a Plug-in-Based Application

The layers of a *plug-in-based application* are illustrated in Figure 2. There may

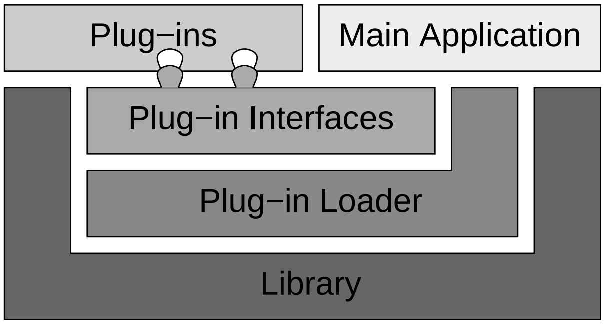


Fig. 2. Layers of a plug-in-based application

be several plug-in interfaces, but one concrete plug-in typically implements exactly one such interface. (It is not so common that a plug-in implements several interfaces. In addition, a plug-in can inherit from a class). Within the code of the plug-in, the classes of the library are used to perform tasks common to several plug-ins. The library contains general routines which would otherwise have to be implemented by a number of plug-ins or the main application. All (additional) functionality is provided by the individual plug-ins, which may belong to different (plug-in) categories depending on the implemented plug-in interface. At runtime, the plug-ins are loaded by the plug-in loader, possibly after being asked for invocation. Thereafter, the plug-in loader manages all plug-ins. (For each plug-in type there may be an extra plug-in loader). Typically, the plug-ins are initialized first and their names are retrieved. When the main application or the library wants to access a certain plug-in, the plug-in loader is called. The main application is implemented on top of the library and gets access to all plug-ins through the plug-in loader and via the plug-in interfaces. All parts of the system which cannot be modeled as plug-ins belong to the main application.

4.2 Plug-in-Based GUI Development

A special form of plug-in-based application development is the *plug-in-based GUI development* which is detailed in [8]. There, the main application is a GUI which is composed of plug-ins. It implements only the static part of the GUI — the part which cannot be implemented as a plug-in — and integrates the GUI plug-ins.

Import and export in different file formats can be done by plug-ins. Therefore, within the open and export dialog the plug-ins are displayed as file types and used for such files. Another type is a plug-in which is displayed as a menu item and performs a certain command.

This idea can be extended by far. For example, preferences can be implemented as individual plug-ins. Then each plug-in is a panel which can be displayed in a tabbed panel with the plug-in names on the tabs. It is also common, that a selection can be made and thereafter a panel is displayed which allows settings for this selection. Each such possible selection can be a plug-in which is a panel (like before).

Furthermore, info dialogs can be implemented by plug-ins. In a dialog the corresponding info plug-in is locally and known. Therefore, it can be used within the respective dialog. And, panels that display a certain kind of objects, such as images, can be designed as plug-ins, so-called view plug-ins. A Java sample implementation of a view panel is given in the appendix.

4.3 Sample Applications

SLC: an Internet portal sample application described in [11], is an example of a plug-in-based application. The main functionality of this portal is provided by individual *portlets* which are plug-ins loaded at runtime. These plug-ins implement a certain interface which allows the main part to ask these portlets about their desire to generate some output and to request content for certain parts of the output area. The screenshot in Figure 3 shows the typical starting screen with just three active portlets, message of the day, list classes, and login, because the user is not yet authenticated.

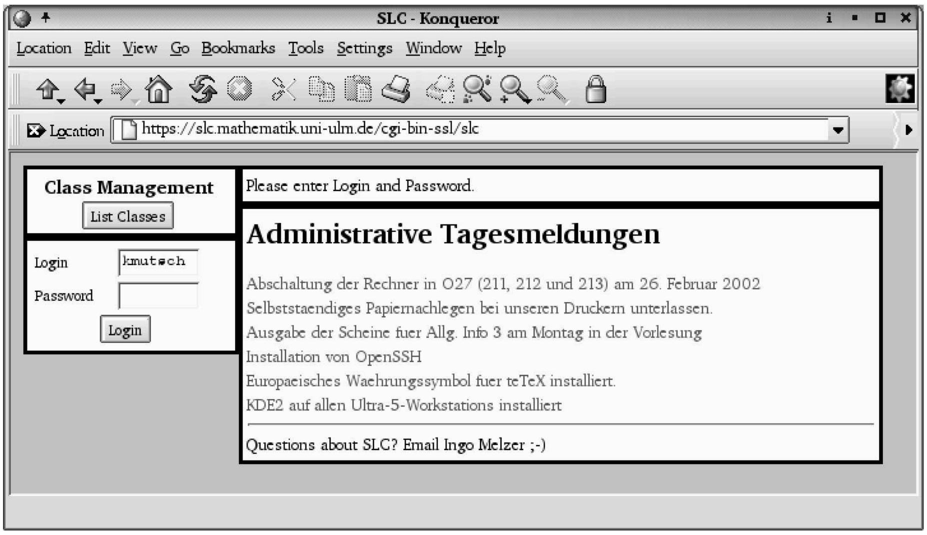
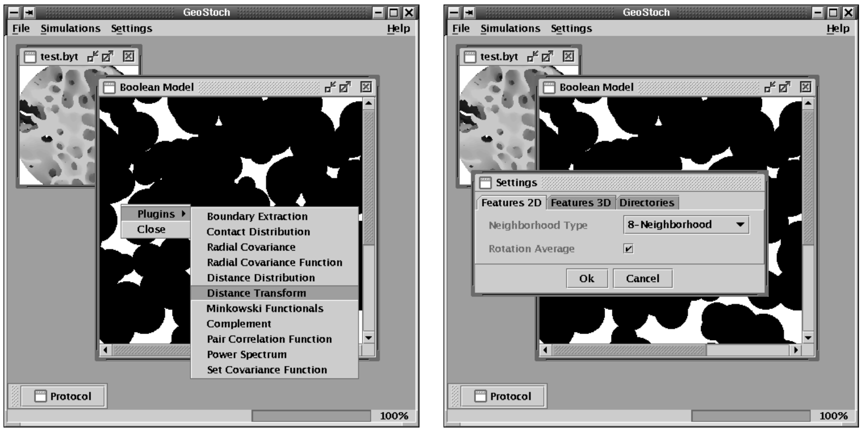


Fig. 3. Screenshot of the SLC starting page

The main application only loads the portlets, directs the flow control (to the individual portlets), and arranges the output. The main application and the portlets have access to a library which grants access to a persistent storage and contains output element for a consistent GUI.

GeoStoch: a Java library for image analysis and image processing applying methods from stochastic geometry and spatial statistics developed at the University of Ulm. In order to make the functionality of the library available for non-developers, there is a sample GUI application (cf. Fig. 4). Since not only



(a) (b)

Fig. 4. Screenshots from the GeoStoch sample GUI

computer scientists but also mathematicians are involved in the development of the library, the idea was to make it easy to extend the GUI as well as the library itself. To add functionality to a library, typically, a new class has to be written, which possibly inherits from and uses others.

So the goal was to develop the GUI in a similar manner. Therefore, the GUI has also been designed plug-in-based with several plug-in types. One plug-in type is used for image analyzers and image operators. For each new image opened or created, a plug-in menu is dynamically configured with all plug-ins applicable (cf. Fig. 4a). First, a plug-in loader determines all available plug-ins. Then, each plug-in is asked whether it can be applied to the new image. In case of a positive response, this plug-in is added to the plug-in menu of the new image with the name provided by the plug-in itself. Images with different types, such as two and three dimensional images, usually have different sets of plug-ins applicable and consequently different plug-in menus.

Plug-ins are also used for preferences (cf. Fig. 4b), selections, import, and export as described in Section 4.2. Plug-ins, such as preference plug-ins, which belong to an other class, are inner classes of that class. This improves the maintainability, since plug-ins which logically belong together are in the same source file.

5 Technical Issues

For an implementation of the Plug-in Pattern, the determination of subclasses of a given class or interface at runtime is necessary. There is no programming language support for this need. So to determine classes implementing a given interfaces at runtime, a file system search can be used, since often a one-to-one correspondence between files and classes exist. But this requires the programming language being capable of loading a class at runtime whose name was not known at compile-time. This is not directly possible for example in ISO C++ [12], but a non-object-oriented solution is possible as discussed in Section 5.3.

5.1 Java

For Java, the technique based on a file system search is described in [6] and a fully functional implementation which searches for all subclasses of a given interface or class is given. The program scans directories as well as jar archives. The name of the class or interface must be specified, as well as the package name and a list of all classes which are subtypes (of the given class or interface) is returned. This technique is type-safe since each found class is attempted to load and a type test is performed. For this purpose, the method Class.forName() which yields a Class object from the name of the class is used.

A class which implements a plug-in interface should always have a public constructor without parameters (in order to permit instantiation). This requirement is common in the context of Java Beans and no real restriction. Then, it is possible to use the method newInstance() from the Class object in order to construct a new object.

5.2 Perl

It is possible to implement a similar solution in Perl. The following listing demonstrates a basic approach.

```
sub scan {
   my ($path) = @_;
   my $dir = new IO::Dir $path;
   die "Unable to scan $path: $!" unless defined $dir;
   my $filename; my %plugins = ();
   while (defined($filename = $dir->read)) {
      next unless $filename =~ /^(.*)\.pm$/;
      my $name = $1;
      my $module = "Plugins::$name";
```

```
      my $plugin;
      eval qq{
         use $module;
         \$plugin = new $module;
         die "Wrong Interface"
            unless (\$plugin->isa(\'Plugins::Base\'));
      };
      if ($@) { warn "$module ignored: $@"; next; }
      $plugins{$name} = $plugin;
   }
   return \%plugins;
}
```

A given directory is scanned for possible plug-ins. For each found module, it is tested, whether the module can be loaded dynamically and whether it is an implementation of the plug-in interface. The function `scan` returns a pointer to an associative array of found plug-ins.

5.3 Discussion of Alternative Techniques

The techniques discussed in the previous sections are object-oriented and easily implemented without great effort. Additionally, they are platform-independent due to the platform-independence of Java and Perl.

Plug-ins for Netscape's Communicator are native code libraries (cf. [13]), shared objects, which are common on many UNIX-systems, or dynamic link libraries, which are used by the various Windows operating systems. Both solutions can be used to implement plug-ins, since a native code library whose name was unknown at compile-time can be loaded at runtime, through `dlopen()` under Unix ([12]) and `LoadLibrary()` under Windows. But, such programs are basically ready compiled for a *specific* platform without a full program loader. Moreover, it requires more effort to build such a library.

3D-FTP[4] implements plug-ins on top of Microsoft's Component Object Model (COM). The latter is an architecture that specifies interaction of binary software components [16]. Each component is an executable or a native code library which provides at least the interface IUnknown. Interaction of components is transparently possible across processes and over the network. Applications with plug-ins do not need this feature. So, this approach has not to be used because there are simpler possibilities — as mentioned in the previous sections. As with native code libraries, the binaries are platform dependent. In addition, COM is mainly focused on the Windows platform, although there are implementations for other platforms.

Overlays in Pascal [5] — a quite old technique — are like plug-ins. They are loaded only when needed. But, overlays must be known at compile-time. Therefore, they are not suitable to implement plug-ins.

[4] `http://www.3dftp.com/`

6 Related Work

This section focuses on non-technical related work. Technical alternatives were discussed in Section 5.3.

Plug-in-based application development is a special case of component-based software development and is not as universally applicable, since concrete plug-ins must be mutually independence of each other.

In [19], a pattern for pluggable components is given. Pluggable components are dynamically configurable software components. There may be several types, each with several implementations. For each type, one implementation has to be selected and configured at runtime. The available types and implementation are hard-coded in the source code or in a file. Therefore, pluggable components, as defined in [19], are no plug-ins, since they are not searched for at runtime and for each type only one implementation is configured and thus usable.

A set of patterns concerning plug-in-based application development is provided by [7]. For each aspect and problem a pattern is presented, while mostly omitting implementation details. Even class diagrams are missing. Therefore, one may be confused by such a huge amount of patterns. In contrast, the Plug-in Pattern presented here is kept as simple as possible to describe precisely what is meant by a plug-in. Additionally, implementation details and examples are provided. Plug-in-based application development — as proposed here — goes further than a pattern by encouraging the development of whole applications based on the plug-in concept.

Recently, the plug-in concept is being used in the context of mobile agents, where negotiation capabilities, including protocols and strategies, can be embedded and removed through plug-ins during runtime [17]. An architecture for collaborative interface agents whose response is generated by plug-ins is presented in [14].

Netscape Communicator is a well-known application supporting pluggable extensions which are native code libraries [13]. Therefore, all comments on dynamic link libraries apply. Particularly, these plug-ins are platform dependent which implies that a specific plug-in must be coded and built separately for every platform. Each plug-in is written to handle some specific MIME types and can be displayed within a rectangular region of the browser window or its own window, if it is not hidden. Installed plug-ins are searched by the Communicator in a special directory. A plug-in must implement a number of methods — the plug-in interface. For the implementation of a plug-in, an API provided by the Communicator with functions for drawing, memory management, streams, and URLs can be used. This has the disadvantage that functionality of standard libraries is re-written instead of using such a library. In addition, the API and plug-in methods have overcrowded and not easily understandable signatures, which is due to the lack of object-orientedness. From within a plug-in, it is possible to call Java or JavaScript by using LiveConnect, but there is no direct way for Java or JavaScript plug-ins.

A similar approach has been made by Adobe [1] for their products Photoshop, Illustrator, and its Acrobat suite. All plug-ins are controlled by an adapter which also handles the basic communication with these plug-ins. A technology called Plug-in Component Architecture, short PICA, is used to allow one common interface for different applications. Each plug-in must have a Plug-in Property List, short PiPL, which contains information about the plug-in and how it should be called. Based on this list, the adapter is able to decide, whether a plug-in can be used in a given situation. Each plug-in must be compiled for a certain platform, which is also listed in the PiPL.

The public domain image processing and analysis program ImageJ[5] which is written in Java also supports plug-ins [2]. Each plug-in implements one of two plug-in interfaces. Possible plug-in types are command and image filter plug-ins. The loading mechanism is also based on a file system search. Plug-ins within jar archives are not supported. The actions behind menu items are implemented by plug-ins, which are assigned in a property file or dynamically loaded. Since there are no explicit plug-in types for image import and export, it is not an easy task to add such a plug-in. Huge interfaces, such as that from the basic class `ImageProcessor`, make it quite difficult to write non-trivial plug-ins for the first time.

An interesting approach to design an integrated development environment, short IDE, has been made by IBM, for its Eclipse Platform[6]. The standard platform is an IDE for nothing in particular, but it can be expanded by plug-ins depending on the actual project. The platform runtime locates plug-ins on start up. However, it is not possible to load any further plug-ins after this phase.

7 Discussion and Conclusion

The plug-in concept has proven useful in many fields. Many of applications such as Netscape Communicator and Adobe Photoshop permit substantial extensions through their plug-in architecture.

The concept of plug-in-based application development and particularly plug-in-based GUI development goes one step further. Thereby, it is possible to divide the development of big systems into manageable small components which can be evolved independently. This has the advantage that third parties can contribute their work independently. Furthermore, due to the plug-in concept, it is possible to add small parts, which — in contrast to classic modularization — does not necessarily involve changes in other parts of the system, without knowledge of the design of the whole application. This reduces the need for repeated tests caused by little changes. Additionally, the use of plug-ins reduces the complexity of the design and makes it more understandable.

However, it has to be pointed out that not all parts of a system can be modeled as plug-ins. The main application which calls the plug-ins is one example. A disadvantage of the proposed approach is that it leads to a high number of modules, which can be dealt with by a well organized package structure. Further-

[5] `http://rsb.info.nih.gov/ij/`
[6] `http://www.eclipse.org/`

more, modules which must be directly referenced cannot be modeled as plug-ins. This is also true for dependent system components.

The approach of plug-in-based GUI development proposed in [8] and the present paper requires further work. It has to be tested in various other fields for practicability.

Acknowledgment

The authors are grateful to Matthias Grabert for helpful comments on a draft version of this paper.

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. Bailer, W.: Writing ImageJ PlugIns – A Tutorial. (2001) `http://webster.fhs-hagenberg.ac.at/staff/burger/ImageJ/tutorial/`
3. CNET Glossary: term "plug-in". `http://www.cnet.com/Resources/Info/Glossary/Terms/plugin.html`
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
5. Joehanes, R.: Pascal Lesson 2 Contents: Chapter 8 – Overlays. (2000) `http://www.geocities.com/SiliconValley/Park/3230/pas/pasl2008.html`
6. Le Berre, D.: Java Tip 113: Identify subclasses at runtime. Java World (2001) `http://www.javaworld.com/javaworld/javatips/jw-javatip113.html`
7. Marquardt, K.: Patterns for Plug-Ins. In: Proceedings of the Fourth European Conference on Pattern Languages of Programming and Computing, EuroPLoP '99, Bad Irsee, Germany (1999)
8. Mayer, J.: Graphical User Interfaces Composed of Plug-ins. In: Proceedings of the GCSE Young Researchers Workshop 2002 (to appear)
9. McCabe, T.J.: Structured Testing. IEEE Computer Society (1983)
10. McCabe, T.J.: Structured Testing. IEEE Transactions on Software Engineering **2** (1976) 308–320
11. Melzer, I.: An Abstraction to Implement Internet Portals. PhD Thesis, University of Ulm (to appear)
12. Norton, J.: Dynamic Class Loading in C++. Linux Journal **73** (2000)
13. Plug-in Guide for Communicator 4.0. Netscape Communications Corporation (1998) `http://developer.netscape.com/docs/manuals/communicator/plugin/index.htm`
14. Rich, C., Lesh, N., Rickel, J.: A Plug-in Architecture for Generating Collaborative Agent Responses. In: Proceedings of Autonomous Agents and Multi-Agent Systems 2002, AAMAS 2002, Bologna, Italy, ACM Press (to appear)
15. Shalloway, A., Trott, J.R.: Design Patterns Explained: A New Perspective on Object-Oriented Design. Addison-Wesley (2002)
16. The Component Object Model Specification. Microsoft Corporation, Version 0.9 (1995) `http://www.microsoft.com/Com/resources/comdocs.asp`

17. Tu, M.T., Griffel, F., Merz, M., Lamersdorf, W.: A Plug-in Architecture Providing Dynamic Negotiation Capabilities for Mobile Agents. In: Proceedings of the Second International Workshop on Mobile Agents, MA '98, Stuttgart, Germany. Lecture Notes in Computer Science, Vol. 1477. Springer-Verlag (1998) 222–236
18. Unified Modeling Language (UML), version 1.4. Object Management Group (OMG), formal/2001-09-67 (2001)
`http://www.omg.org/technology/documents/formal/uml.htm`
19. Völter, M.: Pluggable Components – A Pattern for Interactive System Configuration. In: Proceedings of the Fourth European Conference on Pattern Languages of Programming and Computing, EuroPLoP '99, Bad Irsee, Germany (1999)
20. Wirth, N.: A Plea for Lean Software. IEEE Computer **28** (1995) 64–68

A Java Sample Implementation

The following Java implementation for the view panel which contains at most one view plug-in that displays an object, demonstrates the usage of plug-ins.

```
import javax.swing.JPanel;

public class ViewPanel extends JPanel {

        // abstract class for all view plug-ins
    public static abstract class ViewPlugIn
        extends JPanel implements PlugIn {
            // returns true, if this panel can display o
        public abstract boolean canDisplay(Object o);
            // sets the object to be displayed
            // (it is assumed that canDisplay returned true for o)
        public abstract void display(Object o);
    }

        // the used plug-in loader
    private PlugInLoader loader;

        // constructs a new view panel
        // (plugInType must be a subclass of ViewPlugIn)
    public ViewPanel(Class plugInType) {
            // test whether plugInType is a subclass of ViewPlugIn
        if (! ViewPlugIn.class.isAssignableFrom(plugInType))
            throw new IllegalArgumentException("wrong plugInType");

            // create a new plug-in loader for the given type
        loader = new PlugInLoader(plugInType);
    }

        // displays the given object, if possible
```

```
    public void display(Object o)
        throws PlugInNotFoundException {
            // first, the previous view panel is removed
        removeAll();

            // all plug-ins are retrieved
        PlugIn[] plugins = loader.getPlugIns();
            // ... and tested
        for (int i = 0; i < plugins.length; i++) {
                // the type of all plug-ins is guaranteed to
                // be a subtype of ViewPlugIn (see constructor)
            ViewPlugIn plugin = (ViewPlugIn) plugins[i];
                // if a plug-in can display o, it is used
            if (plugin.canDisplay(o)) {
                plugin.display(o);
                add(plugin);
                return;
            }
        }

            // in this case, not suitable plug-in has been found
        throw new PlugInNotFound("no such view plug-in");
    }

}
```

The used Java implementation of the `PlugInLoader` is based on the technique described in [6].

After a call to the method `display()`, all plug-ins of the specified type are retrieved. Thereafter, the plug-ins are asked via the "voting method" `canDisplay()`, whether they can display the given object. The first plug-in which gives a positive answer is chosen to display the object.

Even though, `ViewPanel` uses a `PlugInLoader` object, it can also be regarded as a "higher level" plug-in loader.