

Plugin-Based Systems with Self-Organized Hierarchical Presentation

Boto Bako, Andreas Borchert, Norbert Heidenbluth, Johannes Mayer*

Ulm University

Dept. of Applied Information Processing

D-89069 Ulm, Germany

Email: {boto.bako,norbert.heidenbluth,andreas.borchert,johannes.mayer}@uni-ulm.de

Abstract—Plugin-based applications can be dynamically extended at runtime. This leads to highly extensible systems on the one hand, but structuring the representation of plugins is difficult on the other hand. Often plugins are represented in graphical user interfaces (e.g. in menu structures) which are generated at runtime. But the kernel does not have—and may not have—any knowledge of the participating plugins. Therefore, the kernel can not care about the plugins’ order. This may lead to menu structures with varying order each time the application is run. Obviously, this results in less intuitive and ergonomical graphical user interfaces. Self-organizing mechanisms producing a linear structure of plugins have been successfully applied to plugin-based applications in previous work. But with an increasing number of plugins involved, linearly ordered structures will soon become crowded. Therefore, in the present paper the self-organizing mechanisms are extended to enable hierarchically ordered plugins and presented in a pattern. The hierarchical order can be embedded in GUI tree structures, which lead to ergonomic and intuitive self-organizing menu structures. A sample application of the presented pattern in the context of Internet portals is described.

Keywords: Plugin, Self-Organization, Hierarchic Presentation, Plugin-Based System

I. INTRODUCTION

Most of today’s software architectures have to be extensible and scalable. One solution to achieve these goals is designing these systems plugin-based. Usually, each plugin provides a certain set of functionality which can be added to (or removed from) the system—even at runtime. Hence, the most obvious advantages of plugin-based systems are that their kernel can be kept very small and the end-user can decide (at runtime) which plugins (i.e. functionalities) he likes to use, and add or remove them accordingly.

Thus, plugin-based systems provide a maximum of freedom in adjusting an application’s functionalities. Eclipse¹ is a well-known example for a plugin-based system.

However, with plugins being added and removed, providing an ordered representation of the plugins is a non-trivial task. Plugins are typically represented within the GUI (by menu items, preference panels, etc.) [1], [2], but their order is mostly arbitrary and may even vary each time the application is run. With respect to ergonomics, this is not satisfactory. On the other hand, the application’s kernel must not organize the

structure either, since it has to be completely independent of the plugins involved. Otherwise, this would contradict to the notion of plugin-based systems.

As mentioned in [3], self-organization is one possible solution for making the plugins cooperate and create a reproducible order at runtime. In [3] it is shown how plugins can be linearly structured by self-organizing mechanisms.

A linear menu structure would become more and more confusing when the number of plugins involved increases. Thus, it would be desirable to order the plugins hierarchically. The present paper extends the self-organizing approach presented in [3] and derives a design pattern for hierarchically structured plugins. A hierarchical structure is a tree structure which orders the participating plugins and can be used to build e.g. reproducible menu structures if plugins can insert themselves at well-defined positions. An example for the application of the presented pattern in the context of Internet portals will also be described.

Section II introduces to plugin-based systems and self-organization. Based on this, a design pattern for hierarchically ordered plugins is presented in Section III. An application of the presented pattern is shown in Section IV, where a case study in the context of Internet portals is described. Section V describes related work on plugin-based systems and self-organized presentation in GUIs. Section VI summarizes and concludes the present paper.

II. PRELIMINARIES

In the following, the basics of plugin-based systems and self-organization will be briefly introduced.

A. Plugin-Based Systems

The basics of plugin-based systems are introduced as in [3].

In [4], Szyperski introduced dynamically extensible applications which are nowadays known as plugin-based systems. They have an important feature: they can be extended even after their deployment. These systems offer several *plugin interfaces* (or *extension points*) that can be used for extensions. *Plugins* can easily be installed and added to a plugin-based system even by end users at any time. This is also the main difference between plugin-based systems and component-based systems [5], [6]. A *plugin* can be seen as a component that has some *plugin type*, i.e. provides and conforms to some *plugin*

* Corresponding author

¹<http://www.eclipse.org/>

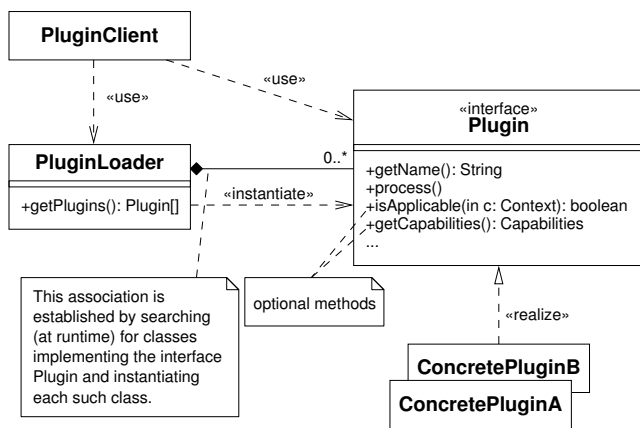


Fig. 1. Class diagram for the Plugin Pattern

interface (or extension point). Plugins also depend on the execution environment provided by the plugin-based system and can thus not be used standalone. The important properties of plugins and plugin-based systems are as follows:

- Plugins can be added at any time (also by end-users).
- Plugin-based systems offer certain plugin interfaces (or extension points).
- Plugins have a plugin type, determined through the provided plugin interface (or extension point).
- Plugins are components that can only be used with the application (or environment) they have been developed for.
- Plugin-based applications can be executed with no plugins—having minimal functionality in this case.

The plugin concept may be used on an architectural level [7]. In this case, one might have systems of different *degrees of plugin-basedness*. For example, a browser may allow plugins to display custom formats by one plugin interface—a low-degree plugin-based system. Only a certain aspect of such a system can be extended by means of plugins. Whereas, one might imagine systems built entirely from plugins (besides a small kernel)—such as the GeoStoch GUI² [1] which is a plugin-based GUI [1], [2]. Plugins are used in [1], [2] to model every small element of a GUI. If the GUI classes are well-structured in packages, the system can easily be understood and extended. A design pattern for the plugin concept is presented in [8] to enable the simple application of this technique at design level. In Figure 1 (taken from [3]) the structure of this design pattern is shown. In the following, the individual elements of the pattern are explained:

- Plugin

The *plugin interface* contains all methods necessary to let the concrete plugins provide their functionality. It also defines the semantics of these methods and the interface protocol, i.e. the allowed sequences of method calls. Usually, there is a method to retrieve the name of a plugin (e.g. for a menu entry). Furthermore, plugins may

only be invoked on certain objects. This can be tested with so-called *voting methods* like `isApplicable()`. Another possibility to assess the applicability of a plugin is to “ask” it for its *capabilities* (cf. method `getCapabilities()`).

- ConcretePlugin
Concrete plugins implement the plugin interface and provide their specific functionality accessible through this interface. Concrete plugins may not directly depend on each other, but use each other via the Plugin interface (and PluginLoader) only.
- PluginLoader
The PluginLoader makes a file system search and discovers all installed plugins (possibly added by the end-user through a plugin installer). It instantiates and provides them to all its clients.
- PluginClient
A plugin client uses a PluginLoader object to load all accessible plugins, acquires the loaded plugins, and uses the plugins based on their provided plugin interface. Optionally, a plugin client first uses *voting methods* or checks the *capabilities* of plugins to only retain those plugins that can be used in the given context.

In [1], [8], the above pattern is presented and explained in detail, and a Java implementation of the class PluginLoader is also provided.

B. Self-Organization

The idea of self-organization was first introduced by René Descartes in 1691 [9] (see [10], page 6, for the relevant quotation in English) and coined as a scientific term for the first time by W. Ross Ashby [11]. It gained popularity in the description of various phenomena in a surprising number of scientific areas including biology, chemistry, physics, geology, and sociology. In the context of computer science, techniques of self-organization and emergence found application in nature-inspired algorithms (see [12] for a gentle introduction) and agent-based systems (see [13] for an overview).

Unfortunately neither the (mathematical) definitions of self-organization by Ashby [11] nor by Shalizi [10] help us in the actual construction of self-organizing systems. Therefore, they are omitted here. This is perhaps the reason why surveying papers like that by Wolf and Holvoet [14] avoid any formal definitions and present the following informal description instead:

“Self-organisation is a dynamical and adaptive process where systems acquire and maintain structure themselves, without external control.”

In the context of a user interface, we have menus, control elements, and viewers. Each of these elements can be associated with one or more parts (i.e. plugins). As a user interface provides just limited space, an organization is required that selects parts useful for the current task and arranges them ergonomically. A hard-wired organization will not work if parts come and go dynamically—such as plugins. Likewise, a user should not be burdened with the organization as this

²<http://www.geostoch.de/>

would become an on-going task. Hence, the only scalable solution is a framework that allows individual parts to organize themselves autonomously and cooperatively.

III. DESIGN PATTERNS FOR SELF-ORDERING PLUGINS

The plugin architecture has analogies with a self-organizing system: each plugin is independent and provides a certain part of the system’s functionality like entities in a self-organizing system. By providing a certain type, a plugin also decides which part of a system to contribute to. This is already a simple form of self-organization. With each plugin being removed from or added to the system, the degree of functionality varies without affecting the system’s stability [15]. By providing voting and capability methods through the plugin interface (see [1], [8] for details), plugin invocation reveals a self-organizing system.

Another crucial point in self-organizing systems is the ability to structure the components. Melzer’s Internet portal (see [16]), for example, allows portlets (i.e. portal modules designed as plugins) to be selected from a menu structure. This menu can have a simple structure like a linear list or a hierarchic structure. Such a plugin (i.e. a so-called portlet in this context) must have the option to contribute to this structure without referring to other plugins. In the following, a design pattern is presented, describing how hierarchic structures can be realized in general. This design pattern roughly follows the notation of [17]. However, the pattern notation is used here not to describe a known best practice solution, but a new solution.

The Hierarchically Structured Plugin Pattern

Intent: Realization of hierarchically structured plugins by means of self-organization.

Motivation: A hierarchic organization of elements is also common, besides linear structures. A tree structure of elements has a big advantage over a linear structure if there is a large number of elements. In Figure 2, we can see the preferences

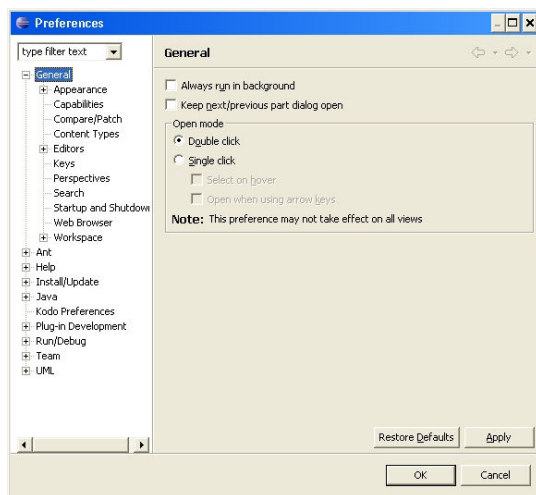


Fig. 2. Example of a hierarchic structure

panel of the Eclipse IDE. As we can see on the left side of

this figure, the settings are grouped according to topics which can be expanded by clicking onto the “+”.

As each one of these settings panels is independent of all other panels, we could easily realize such a preferences structure by using the Plugin Pattern: each settings panel is provided by a single plugin. Adding or removing panels can then be done by adding or removing the appropriate plugin.

When realizing a hierarchic structure, some important aspects must be considered. To satisfy the requirements of the plugin concept, it is not allowed for the kernel to impose a certain structure. A plugin must be able to add itself (or a sub-tree) to the tree without the knowledge of other plugins. Furthermore the hierarchic structure must be reproducible. That means the structure should not be generated arbitrarily. Thus the ordering of sibling nodes must be considered. It is not always important for a plugin to know the entire path up to the root, it might be desirable to refer to a parent node only.

Structure: Figure 3 shows the interfaces and classes of the pattern.

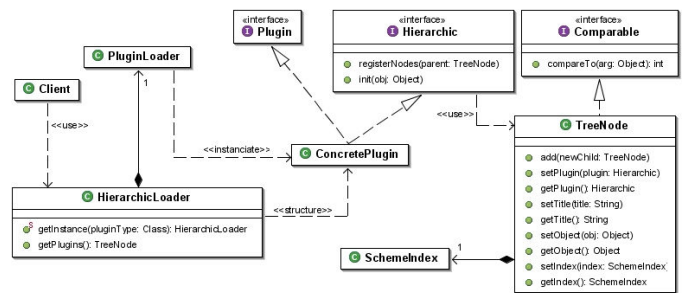


Fig. 3. Class diagram for the Hierarchically Structured Plugin Pattern

Participants:

- **TreeNode**
 - this class represents a node in the hierarchic structure.
 - a new child node can be inserted via the method `add()`.
 - a plugin that is associated with this node can be set via the method `setPlugin()` and determined by calling the method `getPlugin()`.
 - a title of the node that is displayed e.g. in a menu can be set and retrieved through calls to the methods `setTitle()` and `getTitle()`.
 - optionally, an object can be associated to the node by calling the method `setObject()`. This option allows for the multiple registration of a plugin in the tree. The respective registered object is passed to the plugin by the client using the `init()` method when the node is selected and before the plugin is used. The object that should be passed to the plugin can be retrieved via a call to the method `getObject()`.
 - the position of the node among the sibling nodes can be set and retrieved through the methods `setIndex()` and `getIndex()`. Implementing the **Comparable** interface, the sibling nodes can be ordered according to

their position.

- *SchemeIndex*
 - is used for the linear ordering of sibling nodes as described in [3].
- *Hierarchic*
 - is implemented by plugins that participate in a hierarchic structure.
 - a plugin is requested to register its sub-tree when the method `registerNodes()` is called. The parameter parent is the node where the sub-tree can be inserted, but if desired, the plugin could also determine the root node and register an entirely new tree path. A plugin can thus be registered at several locations with different initialization objects set at the different nodes. If a node is selected, the method `init()` has to be called by the client to permit a plugin initializing its state with the passed object.
- *HierarchicLoader*
 - loads the plugins of a certain type using a `PluginLoader` instance.
 - iterates over the loaded plugins and asks them to register in the hierarchic structure, calling the method `registerNodes()`.
 - allows clients to access the hierarchic structure via the method `getPlugins()`.
- *Client*
 - retrieves all plugins of a certain type via a call to the method `getPlugins()`. The returned node is the root of a tree that contains the plugins.
 - has to initialize a plugin obtained from the tree with the object from the selected node before calling other methods of the plugin.

Consequences: The Hierarchically Structured Plugin Pattern allows plugins to register several times in a hierarchic structure. A plugin has to provide an initialization object for each registered node. A client that uses such a structure has to initialize a plugin after the selection of a node. After a plugin has been initialized, it is ready to perform tasks asked for by the client.

Implementation: A concrete plugin has to implement an additional interface, the interface `Hierarchic`. This interface contains the method `registerNodes()` in which a plugin has to register itself to the provided tree structure. In the simplest case, a plugin inserts only one node. Listing 1 shows how this can be implemented.

Listing 1. A possible implementation of the method `registerNodes()` of a concrete plugin

```
public void registerNodes(TreeNode parent) {
    TreeNode node = new TreeNode();
    node.setIndex(new SchemeIndex(new int[] {2,1}));
    node.setTitle("Plugin title");
    node.setPlugin(this);
    parent.add(node);
}
```

First an empty node is constructed. Then the required attributes are set. These are the index for the ordering of sibling nodes,

a title that is displayed e. g. in a menu, and the plugin itself that is called after its selection. (An initialization object can be omitted in this case since the plugin has registered only once within the tree structure.) Finally, the node is inserted in the tree structure.

The presented design pattern can be applied in most cases in which a hierarchical structure is necessary. However, a disadvantage of this pattern is that a plugin has to know the details of the class `TreeNode`. For many applications a simpler solution can be applied at the expense of flexibility and functionality. As described in [18], a plugin might register itself by providing a string containing the path information. Figure 4 shows the interfaces and classes of the adapted pattern.

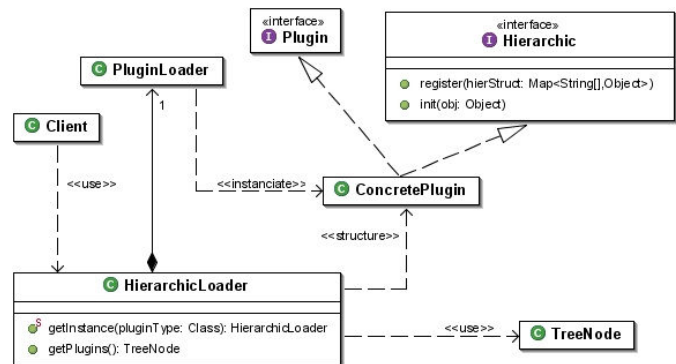


Fig. 4. Class diagram for the adapted Hierarchically Structured Plugin Pattern

A concrete plugin is asked to register itself in the tree structure by calling the method `register()`. The passed parameter is a map to which the plugin adds its positioning requests. The key is a tree path, realized as a string array, containing the names of the nodes as its elements; the value is an initialization object that is passed to the plugin if the leaf is selected. A plugin can thus register itself several times in the tree. The `HierarchicLoader` object loads the plugins of a certain type with the `PluginLoader` object, and iterates over the loaded plugins to collect the information necessary to construct the hierarchical structure built e. g. from `TreeNode` objects.

The drawback of this implementation is that plugins have to know the entire tree path from the leaf up to the root. Furthermore, the ordering of sibling nodes is not solved, causing them to be randomly or lexicographically ordered.

Manual reordering of plugins can be implemented for hierarchic structures analogously to the linear case described in [3]. For hierarchic structures, the user can move whole subtrees of the hierarchic structure.

The client can often be implemented in a reusable way.

IV. CASE STUDY

While the example mentioned before shows a possible application for plugins with self-organized hierarchical presentation, the following case study describes a real application example and therefore demonstrates the feasibility and useful-

ness of the presented idea and pattern as well as the Linearly Ordered Plugin Pattern described in [3].

In [16], Melzer describes a general abstraction for Internet portals and developed a framework which also applies the idea of the Plugin Pattern [8]. According to whether a user is logged in or not, each plugin (here called “portlet”) decides if it wants to appear in the navigation structures or not. The same holds for portlets that need to be run with special privileges. Thus, the framework uses *voting methods* (cf. Section 2.1) here.

The framework generates exactly one linear navigation structure which contains a pitfall though, as the order of its entries depends on e.g. the file system, that determines the order in which the files of a directory are traversed.

Hence, these navigation structures do not only depend on the plugins incorporated, but also on low level details.

In [18], it is shown how those pitfalls can be avoided for the linear case and how the Portal framework can be extended by hierarchical navigation structures. Thus, Melzer’s portal framework evolved to a self-organizing portal framework using the improvements and extensions of [18]: each portlet (i.e. portal module) can contribute one or more entries to an (arbitrarily deep) menu structure under which it will be accessible. The key point is that this menu structure will be generated at runtime.

The example below shows how those menu structures can be built and in which way the portlets can articulate their desired position within a hierarchical structure.

Example: Consider a set of three portlets out of a collection of portlets within a website of an online store:

- Portlet 1: “Terms and conditions” which should be accessible within the navigation structure under the categories “Legal stuff” and “Information for our customers”.
- Portlet 2: “Privacy policy” (to be accessible under “Legal stuff” in the top level of the menu hierarchy and under the level “Privacy” which is a sub-level of the item “What you should know”)
- Portlet 3: “About us” (to be accessible only under “Information for our customers”)

Using the extension of Melzer’s portal framework described in [18], the portlets articulate their “requests for positioning” within the navigation structure by passing each desired position to the framework as a string that uses the “/” as a separator for each hierarchy level.

In our example this would mean:

- for Portlet 1:
 - Legal stuff/Terms and conditions
 - Information for our customers/Terms and conditions
- for Portlet 2:
 - Legal stuff/Privacy policy
 - What you should know/Privacy/Privacy policy
- for Portlet 3:
 - Information for our customers/About us

For the second version of the Hierarchically Structured Plugin Pattern that can be used here, e.g. `Legal stuff/Privacy policy` can be translated into a String array with elements "Legal stuff" and "Privacy policy".

The portal collects all those “requests for positioning” received by its portlets and constructs hierarchical menu structures.

Selecting the menu item “Information for our customers”, one would find the entries “Terms and conditions” and “About us”. Selecting those entries would directly call the functionality in the corresponding portlet (i.e. plugin)—after passing the respective initialization object. Figure 5 illustrates two possible representations of the sample navigation structure. For the navigation structure at the top it is assumed that

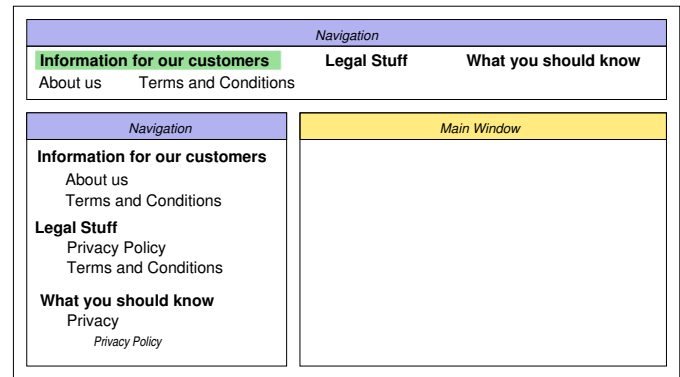


Fig. 5. Representation of the navigation structures

“Information for our customers” has already been selected since its sub-items are displayed. The main window contains the real content of the page.

So far the illustrating example. These ordered navigation structures have successfully been applied to the SLC portal³, the student, lecture, and class management portal of the Faculty of Mathematics and Economics of Ulm University. This portal is implemented using Melzer’s framework [16] with Heidenbluth’s extensions [18]. The SLC system is used day by day by hundreds of students of the faculty for Mathematics and Economics at Ulm University.

The implementation of the SLC system distinguishes between abstract and concrete navigation structures. An abstract structure contains the entries of either a linear or a hierarchical menu but it cannot be displayed. Instead, it must be associated to a corresponding *concrete* structure which is meant to display an abstract structure in exactly one way (for example horizontally or vertically).

Hence, the implementation of the SLC system depends on the presented design patterns—but uses other names for the involved objects and classes. In the following, the mapping between the classes of the presented patterns and the classes of SLC is explained: The classes `NavigationScheme::Linear` resp. `NavigationScheme::Hierarchy` represent the abstract navigation structures and correspond to the classes `LinearLoader` (cf.

³<http://slc.mathematik.uni-ulm.de/>

[3]) resp. HierarchicLoader of the pattern. The concrete menu structures (which are subclasses of the class Navigation) are general, reusable implementations of the Client. Moreover, the Registrar maps to the PluginLoader of the patterns, and the concrete Portlet AboutUs corresponds to the ConcretePlugin in the patterns.

The sequence diagram in Figure 6 shows the first steps within the process of creating a Page object which is used to generate the HTML output by the SLC framework. The steps are explained in the following.

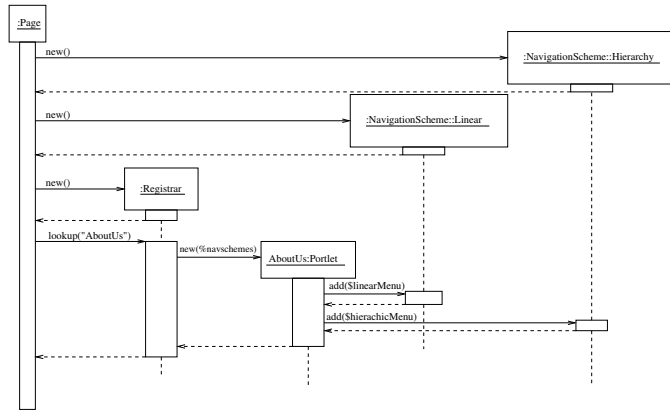


Fig. 6. Constructing abstract menu structures in SLC

- First of all, two abstract navigation structures (a linear and a hierarchic structure) are instantiated (method new()).
- The Registrar object is constructed. This object will search for existing portlets and ask them to contribute to the (abstract) navigation structures constructed before.
- Each portlet found is asked to add entries to the linear and/or hierarchic structure. This example demonstrates the activity of a portlet called AboutUs, but this process is also done for every other portlet.

The hierarchic structure is represented by a complex data structure (an associative array) which leads to an alphabetical order of each level's menu entries whereas the entries of a linear structure are stored within a linear list and indexed with integer numbers. Hence, they can either be displayed in a straightforward way (in ascending or descending order) or can be rearranged according to users' preferences. See [18] for details on personalization within this application.

After the last step above, no HTML code can yet be constructed since the Page object only knows about the abstract menu structures but nothing about their concrete representation.

Figure 7 shows the continuation of the process of generating a HTML page in the SLC system: the next steps embed the abstract menu structures into concrete ones and generate the HTML output. (The variables \$headerMenu and \$leftMenu in Figure 7 represent the abstract navigation structures.)

- A Layout object is instantiated which will contain information about the page's layout (i.e. where to put the

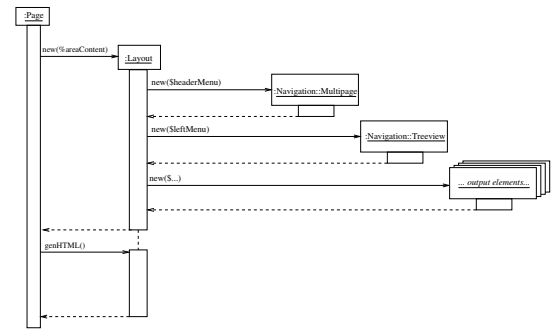


Fig. 7. Associating abstract structures with concrete ones in SLC

menu structures, where to put the main content, etc.). This information is delivered by the parameter %areaContent.

- Depending on this layout information, the layout object will create the required *concrete* navigation structures as well as the content for the main area.

After the last step, the Page object has enough information to generate the HTML code. Calling the object's method genHTML() will produce the HTML code for a Web page like the one sketched in Figure 5 (with the difference that there is no linear navigation structure in the figure and Fig. 5 contains two different representations of the same abstract hierarchic navigation structure).

V. RELATED WORK

In [3] a design pattern for linearly ordered plugins is introduced. The pattern uses concepts of self-organization. When a plugin is loaded it is asked for the desired position in the linear structure. Furthermore the pattern allows the reordering of the structure by end users. A linear organization of plugins can be used for a number of applications, but in many cases hierarchic structures are necessary. Therefore these concepts were used in the present paper and extended for the hierarchic case. The following related work is common with [3].

A new plugin framework is presented in [15], [19], [20]. This framework is especially well-suited to handle plugin interaction. In [20] the implementation of this framework is discussed. For the present paper, however, the simpler pattern and framework given in [8] is sufficient.

In addition to the related work mentioned in the general description of the plugin concept and self-organization, some common plugin-based applications are presented and discussed in the following. First some basic plugin-based applications are examined especially according to their features regarding self-organized presentation. In the second part plugin-based systems with hierarchic plugin structures are evaluated.

A landmarking plugin-based application was *Write*, an extensible text editor for the Oberon system [21]. Its extraordinary small kernel supported just plain text and the option of including text elements as plugins. Within the application framework, elements had to provide methods for storing

and loading from persistent storage, cloning, displaying, and printing. Optionally, elements could provide their own user interface and stay active. Elements were not just implemented for tables and graphical figures but also for spreadsheets, forms, and simulations. Write did not provide any menus of possible elements as users could organize their menus themselves in ordinary text files within the Oberon system. These files could easily be shared with other users or adapted to special purposes.

Netscape's so-called "Gecko-based" Web browsers can easily be extended by plugins.⁴ The most common plugins for these browsers are probably multimedia viewers like for instance the Macromedia Flash plugin or the Adobe Acrobat plugin. For the Web browser each plugin is a platform specific "dynamic code module" contained in an explicit (platform dependent) directory. When the browser starts, it checks this directory and registers all plugins that could be found. As each plugin is designed to handle one (or even more) specific kind(s) of MIME type(s), registration means that all supported MIME types are collected. If more than just one plugin tries to register the same MIME type, the first plugin registered will handle that type. A particular plugin is executed if the browser encounters data of a MIME type registered by this plugin. This is a simple form of self-organization.

The popular image processing tool *GIMP*⁵ also allows for extensions by plugins. *GIMP* loads all plugins at startup. The plugins have to register themselves in the "procedural database" (PDB). During registration, the plugins provide the menu path they want to be incorporated in. This menu path also decides about the type of the plugin (image acquisition, image processing, etc.) and consequently about some of its arguments. The menu path exhibits some form of self-organization. The entries in a sub-menu are ordered arbitrarily (more precisely, in the order in which they are loaded). Therefore, the amount of self-organization is a bit limited.

*ImageJ*⁶, another popular image processing program, can also be extended by plugins [22]. There are two types of plugins: one type for plugins that do not require an image as input (interface *Plugin*) and another one that acts as an image filter (interface *PluginFilter*). An *ImageJ* plugin has to implement one of these interfaces. Plugins with an underscore in their name are automatically installed—otherwise, only manual installation is possible. When automatically installed, a plugin is in the plugin menu. Through manual installation, a plugin can be put in any menu. The order of the menu entries is however determined by the order of the installation. Therefore, *ImageJ* does not really use self-organization for the presentation of its plugins—besides the plugin type.

Another common plugin-based system is the Eclipse platform. Characteristic for Eclipse is that almost the whole functionality is provided by plugins, resulting in a highly extensible system. A plugin can define so-called extension

points, where other plugins can be integrated. A plugin is described in an XML file where the contributions to extension points and own extension points are specified. This file is read by Eclipse at start up and the plugins are loaded according to this specification. In [23] it is described how the Eclipse Workbench can be extended by plugins. A plugin can contribute to the main menu bar by specifying the id of the menu (e.g. file) together with a group id (e.g. save.ext). Such a configured plugin would appear in the save group of the file menu. A menu plugin can provide own groups where other can contribute to. This way a hierarchic structure can be realized, but the order within a group is arbitrary (it depends on the loading order and the ids of the plugins).

VI. CONCLUSION

Plugins may have representations on the GUI, e.g. in menu structures. The more plugins are involved in a plugin-based application, the harder it gets to generate reproducible and ergonomic menu structures. This is, because the kernel must not have any knowledge of the participating plugins and therefore isn't a suitable instance for generating menu structures.

Self-organizing linear menu structured—as presented in previous work—are one applicable solution, but with a large number of plugins involved, linear structures will soon become crowded. Extending the idea of self-organizing menu structures, a design pattern has been presented for hierarchically structured plugins in the present paper. This pattern enables all participating plugins to express their "requests for positioning" within a hierarchical structure. A successful practical application of this pattern in the context of Internet portals has been described. In this application, the pattern has been used to organize menu structures plugins contribute to.

Obviously the presented pattern can not only be used in GUI context but also to order arbitrary plugins such as batch jobs which can also be structured hierarchically similar to makefile targets. Furthermore, self-organization can also be used to allow plugins to cooperate and structure themselves in whatever form. Such a generalized self-organized mechanism could be used to build a more flexible and powerful plugin architecture.

REFERENCES

- [1] J. Mayer, "On quality improvement of scientific software: Theory, methods, and application in the geostoch development," Ph.D. dissertation, Ulm University, 2003.
- [2] —, "Graphical user interfaces composed of plug-ins," in *Proceedings of the Fourth European GCSE Young Researchers Workshop 2002*, ser. Fraunhofer IESE Technical Report, vol. 053.02/E, Kaiserslautern, Germany, 2002, pp. 25–29.
- [3] B. Bako, A. Borchert, N. Heidenbluth, and J. Mayer, "Linearly ordered plugins through self-organization," in *Proceedings of the Workshop on Self-Adaptability and Self-Management of Context-Aware Systems (SELF 2006)*. IEEE Society Press, 2006, (in press).
- [4] C. Szyperski, "Independently extensible systems: Software engineering potential and challenge," in *Proceedings of the 19th Australasian Computer Science Conference*. Computer Science Communications, 1996.
- [5] C. Szyperski, D. Gruntz, and S. Murer, *Component Software – Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley / ACM Press, 2002.

⁴http://developer.mozilla.org/en/docs/Gecko_Plugin_API_Reference

⁵<http://www.gimp.org/>

⁶<http://rsb.info.nih.gov/ij/>

- [6] G. T. Heinemann and W. T. Council, *Component-Based Software Engineering*. Upper Saddle River: Addison-Wesley, 2001.
- [7] H. Cervantes, H. Almeida, J. Mayer, and A. Perkusich, "Plugin-based software architectures," 2006, preprint (submitted for publication).
- [8] J. Mayer, I. Melzer, and F. Schweiggert, "Lightweight plug-in-based application development," in *Proceedings of the Net.ObjectDays 2002*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2003, vol. 2591, pp. 87–102.
- [9] R. Descartes, *Discours de la méthode pour bien conduire sa raison, et chercher la vérité dans les sciences*, Leiden, 1637.
- [10] C. R. Shalizi, "Causal architecture, complexity, and self-organization in time series and cellular automata," Ph.D. dissertation, University of Wisconsin at Madison, 2001, <http://cscs.umich.edu/~crshalizi/thesis/>.
- [11] W. R. Ashby, "Principles of the self-organizing dynamic system," *The Journal of General Psychology*, vol. 37, pp. 125–128, 1947.
- [12] M. Resnick, *Turtles, Termites, and Traffic Jams*. The MIT Press, 1991.
- [13] G. Di Marzo Serugendo, N. Foukia, S. Hassasand, A. Karageorgos, S. Kouadri Mostéfaoui, O. F. Rana, M. Ulieru, P. Valckenaers, and C. Van Aart, "Self-organisation: Paradigms and applications," in *Engineering Self-Organising Systems: Nature-Inspired Approaches to Software Engineering*, ser. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2004, vol. 2977, pp. 1–19.
- [14] T. D. Wolf and T. Hovoet, "Emergence versus self-organisation: Different concepts but promising when combined," in *Engineering Self-Organising Systems: Methodologies and Applications*, ser. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2005, vol. 3464, pp. 1–15.
- [15] R. Chatley, S. Eisenbach, and J. Magee, "Modelling a framework for plugins," in *Proceedings of the Workshop on Specification and Verification of Component-Based Systems*, ser. Technical Report 03-11. Iowa State University, 2003, pp. 49–57.
- [16] I. Melzer, "An abstraction to implement internet portals," Ph.D. dissertation, Ulm University, 2002.
- [17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [18] N. Heidenbluth, "Zur Selbstorganisation und Personalisierung von Navigationsstrukturen in Internet-Portalen," Master's thesis, Ulm University, 2004.
- [19] R. Chatley, S. Eisenbach, and J. Magee, "Magicbeans: a platform for deploying plugin components," in *Proceedings of Second International Working Conference on Component Deployment, CD 2004, Edinburgh, UK*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2004, vol. 3083, pp. 97–112.
- [20] —, "Painless plugins," Department of Computing, Imperial College London, Tech. Rep., 2003, <http://www.doc.ic.ac.uk/~rbc/writings/pp.pdf>.
- [21] C. A. Szyperki, "Write-ing applications: Design of an extensible text editor as an application framework," in *Seventh International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'92)*, Dortmund, Germany, 1992.
- [22] W. Bailer, "Writing ImageJ plugins—a tutorial," Version 1.6. FH Hagenberg, Austria, June 2003, <http://mtd.fh-hagenberg.at/depot/imaging/imagej/ijtutorial.pdf>.
- [23] S. Arsenault, "Contributing actions to the eclipse workbench," 2001, <http://www.eclipse.org/articles/Article-action-contribution/Contributing%20Actions%20to%20the%20Eclipse%20Workbench.html>.