Plug-ins: an Architectural Style for Component Software

Jörg Rathlev
Universität Hamburg
Department Informatik
Vogt-Kölln-Straße 30
22527 Hamburg
Email: rathlev@informatik.uni-hamburg.de

Abstract—**Plug-ins are a common mechanism for extending applications, and have recently become popular as the building blocks for whole applications in so-called pure plug-in architectures. How are these architectures related to software components? In this paper, we propose that plug-in architectures can be seen as an architectural style for component software. We discuss the architectural elements and properties of this style and describe how the style can be specialized in domain-specific ways.**

I. Introduction

Many applications provide a mechanism that allows those applications to be extended by plug-ins. Plug-ins are installed into the host application and integrate additional functionality into it. Recently, another kind of architecture has gained popularity: so-called pure plug-in architectures [3].

In these architectures, there is no longer a monolithic host application which is extend by plug-ins. Instead, the applications itself is composed of plug-ins, and each of these plug-ins can itself be a host for other plug-ins. Probably the most well-known application built with this kind of architecture is the Eclipse IDE, which is designed based on the idea that all functionality is a contribution provided by a plug-in [6].

In pure plug-in architectures, there is only a thin run-time layer which implements the plug-in model. On top of that, the actual application is built entirely with plug-ins. The run-time layer is responsible for loading the plug-ins and for resolving their dependencies [3].

The main motivations for using a plug-in based architecture are higher modularity, which promises the ability to react to changing requirements more quickly. Plug-ins may also offer more opportunities for reuse. Plug-ins can also make it easier to integrate software provided by different vendors, and help users to customize software by integrating site-specific extensions.

The contributions of this paper are:

- We describe the pure plug-in architectural style as an architectural style for component-based systems (section II).
- We discuss how the style can be specialized to domain-specific requirements (section III).
- We briefly discuss some of the advantages and disadvantages of pure plug-in based architectures (section IV).

Finally, we review related work and conclude.

II. Pure plug-in architectures

In this section, we compare plug-ins with traditional software components and discuss the architectural properties of pure plug-in architectures by defining an architectural style for those architectures.

An architectural style defines the elements of an architecture and their allowed relationships [9]. It can be described by a set of element types and their semantics and semantic constraints, a topological layout of the elements, and a set of interaction mechanisms [2].

The context in which plug-in architectures are used is the development of applications which are extensible by third parties, and the development of families of applications which share common parts. When developing applications for those contexts, architects need to balance the following forces:

- The application should be extensible without having to modify the application itself.
- Different extensions, possibly developed by different parties, should be able to interoperate with each other.
- Parts of the application should be exchangeable without affecting other, unrelated parts.
- Components are easier to reuse than monolithic platforms. The common parts of different applications should not reside in a single, big and inflexible platform.
- Different developers should be able to work on different parts of the system at the same time without affecting each other. Developers may work in geographically distributed locations, and may not even know about each other's work in the case of third-party extensions.

In a pure plug-in architecture, a system is comprised of a set of plug-ins which extend each other. An *extension model* defines how plug-ins can contribute functionality to other plug-ins.

In the following discussion, we assume an extension model similar to that of the Eclipse platform, which is based on *extensions* and *extension points*. This extension model and the general concept of extension models are discussed in more detail in the section on interaction mechanisms.

A. Architectural elements

Plug-ins are the software components from which a plug-in-based application is composed. Plug-ins are pieces of software which are individually packaged and can be individually added to and removed from a plug-in-based application, i. e., they are the atomic units of deployment in a pure plug-in architecture.

A plug-in must explicitly declare all of its dependencies. This declaration can take at least two different forms: the plug-in can specify the specific plug-ins it depends on, or it can specify the APIs that it requires (for example, in Java, by specifying a list of required packages). In the former case, the runtime will check during deployment that the required plug-in is available in the system; in the latter case, the runtime will ensure that there is some other plug-in which provides the required API.

Specifying a plug-in's dependencies by listing the required APIs is slightly less simple because developers must specify the dependencies in more detail. The advantage of this approach is that it creates a dependency only on the API that is actually required, not on a specific unit of deployment that exports the API.

In addition to specifying its dependencies, a plug-in must also specify which of its APIs (if any) it provides to other plug-ins by exporting them. Exported APIs can be accessed by other plug-ins that depend on the API or the exporting plug-in.

These properties make plug-ins *software components* according to the widely used definition by Szyperski [10]. A plug-in is a unit of composition, it explicitly specifies its dependencies and interfaces, and it can be independently deployed.

In addition to these properties, a plug-in can also allow other plug-ins to extend it by declaring one or more extension points, and it can extend other plug-ins by declaring extensions. We call a plug-in that declares an extension point a *host plug-in*, and a plug-in which provides an extension an *extension plug-in*. Note that a single plug-in can assume both of these roles.

The *runtime engine* is responsible for the execution of the plug-ins. It must provide functionality for discovering and loading plug-ins, resolving their dependencies, and running them.

When starting a plug-in based application, the runtime engine controls the application's bootstrapping process. It must specify how it finds, loads, and runs the initial set of plug-ins, and how control is handed over to the plug-ins.

The runtime engine also implements the lifecycle management for the plug-ins.

B. Topology

Dependencies on APIs exported by other plug-ins must not be cyclic. If cyclic dependencies were allowed, the plug-ins in a cycle could only be installed together, which would effectively make them a single plug-in. The dependency graph of installed plug-ins with resolved dependencies is therefore a directed acyclic graph.

The extension model does not impose such constraints. In particular, plug-ins are allowed to extend themselves, and to extend each other in cyclic relationships. (The latter, of course, only works if doing so does not require a cyclic dependency on exported interface definitions.)

In practice, extension relations between different plug-ins are usually acyclic. Plug-ins which extend themselves, however, are quite common, based on the principle that a plug-in should provide as much functionality as possible via its own published interfaces and extension points rather than using internal mechanisms which are unavailable to other plug-ins. The designers of the Eclipse platform call this principle the "fair play rule" [6].

C. Interaction mechanisms

There are two primary ways for plug-ins to interact:

- Firstly, a plug-in can interact directly with another plug-in by using that plug-in's API, provided it has declared a dependency on that API or plug-in.
- Secondly, a host plug-in can interact with the extensions to its extension points. This kind of interaction is mediated by the architecture's extension model.

The extension model defines how a plug-in declares the ways in which it can be extended, how it can discover and use its extensions, and how a plug-in can provide extensions to other plug-ins. In this section, we have assumed an extension model based on the concept of extension points and extensions.

An extension point is a declarative specification contained in a host plug-in which specifies the kinds of extensions that can be contributed to this extension point. Essentially, an extension point is the specification of the contract between the host plug-in and its extensions. It can specify the kind of functionality or resources it expects the extension to provide, the interfaces it must adhere to, and so on. The precise format of this specification of course depends on the specific description format used by a specific extension model.

An extension point also defines the semantics of its extensions in the context of the host plug-in. We will see in the next section why this is an important property of plug-ins when creating specialized architectural styles.

An extension is the counterpart to an extension point. It contributes additional functionality to a host plug-in via one of its extension points and is provided by an extension plug-in. It consists also of a declarative specification, which specifies the extension point to which the extension applies as well as any additional declarative information required by that extension point. The declaration may include references to executable code (for example, classes) exported by the extension plug-in. The interfaces that this code must implement are specified by the extension point.

The *extension registry* is used by host plug-ins to discover their extensions. The extension registry is made available as an API, so that host plug-ins can access it directly. The extension registry serves as a mediator which prevents dependencies from host plug-ins to their extension plug-ins. Note that it is the responsibility of the host plug-in when and how to use its extensions. The extension registry does not itself activate or otherwise use the extensions in any way, it only provides information about them.

The number of extensions that can be registered for an extension point is not restricted, in particular, there may also be no extensions at all (in which case the host plug-in should still be able to operate without failure). If the host plug-in can only use one extension at a time, it must implement some arbitration mechanism to select one of the available extensions in case more than one are registered. In an interactive application, this could for example be done by asking the user to choose from list of available extensions.

The implementation of the extension model can itself be provided a plug-in. The underlying runtime engine must provide adequate support for the implementation, for example, by providing a way of obtaining a list of the plug-ins installed in the application (as well as a notification mechanism if plug-ins can be added and removed dynamically at runtime).

Because the implementation of the extension model can be provided by plug-ins, multiple different extension models can co-exist in an application. This can be used when migrating to different technologies, and in fact has been used for this exact purpose in the Eclipse project: When Eclipse migrated from its own plug-in model to using OSGi as its underlying component platform, a compatibility plug-in was added to Eclipse which implements the old plug-in model by automatically translating legacy plug-ins [7].

III. Specializing the style

The architectural style described in the previous section is a relatively low-level style. Its architectural elements are primarily based on the technology used, and the interaction mechanisms are mostly on the level of plumbing and wiring, in the sense that they provide the technical means for plug-ins to interact with each other, but no domain-specific semantics for those interactions. We therefore expect projects that use a plug-in style to specialize this style in domain-specific ways appropriate for the project. In this section, we describe how this can be done.

There are many different ways of specializing an architectural style. We will focus on the following two options:

- Specializing an architectural style by combining it with some other architectural style.
- Specializing the style by creating a domain-specific style for the class of applications that can be built on top of a given set of plug-ins which forms a platform for the domain. In other words, we want to find an architectural style which describes the architectures of those systems that can be built by developing plug-ins that extend the given set of plug-ins.

These options do not exclude each other, and are often used in conjunction.

A. *Domain-specific styles*

One important property of extension points is that they specify a contract between the host plug-in and its extensions. Only those extensions that conform to the contract are accepted as extensions by the host. This enables the host to associate semantics with its extensions. Those semantics are specified from the perspective of the host plug-in, and are therefore specific to its domain.

As explained above, a host plug-in is also responsible for finding and activating its extensions, and may control their whole lifecycle.

A host plug-in can also provide additional, domain-specific interaction mechanisms to its extensions; it can provide domain-specific services; and it can supply base classes, i. e., a framework, that can be used by other plug-ins to implement extensions.

Together, these properties mean that a plug-in that declares an extension point can be seen as a component platform for those components that provide extensions to the extension point; in other words, the plug-in provides a platform for domain-specific plug-ins.

This is an important difference of plug-in based platforms to traditional, container-based component platforms like EJB, which can only provide relatively domain-neutral, technical services to their components because they do not (and cannot) incorporate domain-specific knowledge into their component models. In plug-in-based architectures, this knowledge can be embodied in the extension point specifications and the semantics associated with extensions, without requiring a fundamentally different component model for different domains.

In traditional component models, components specify their dependencies on the context, but the context does not specify any specific requirements that the components have to fulfill (beyond the component model's component specification). In a plug-in architecture, host plug-ins can specify the specific type of functionality they expect their plug-ins to provide.

What does this mean for the specialization of the architectural style?

The domain-specific platform comes with a set of terms that describe the main concepts of its domain. For example, the extensions provided by plug-ins are specific elements of the domain. The types of these elements can become new elements in the specialized architectural style.

Similarly, if the platform provides additional interaction mechanisms, these can also be added to the architectural style.

Thus, a first approach for deriving an architectural style from the set of plug-ins that are used as the platform for applications built with style is to use the kinds of extensions defined by the platform's extension points as new architectural element types, and to add any new interaction methods provided by the platform to the set of interaction methods defined in the architectural style.

For example, the Eclipse Rich Client Platform (RCP) provides a platform for the creation of so-called rich clients [8]. The usage model of this platform is centered around the concept of a workbench. Extension points are provided to extend this workbench with views and editors. Views and editors are two element types that could be incorporated as architectural element types into a domain-specific architectural style for plug-in-based rich client applications.

B. Combining styles

The plug-in architectural style can be combined with other architectural styles. One style that is well-suited for combination with the plug-in style is *layers* [4].

As explained above, host plug-ins can be used to implement platforms for other plug-ins. The different plug-ins can often be associated with different abstraction layers: a plug-in which implements parts of the user interface belongs to the presentation layer, a plug-in which implements the business model belongs to the business logic layer, and so on. The architecture of a system containing such plug-ins is a combination of the plug-in architectural style and the layered style. In this architecture, the plug-ins providing a platform are at the lower layers, and their extensions at the higher layers (which may again be platforms).

Note that this does not imply that an extension is always associated with a higher layer than its host plug-in. A host plug-in can also declare extension points for extension towards lower layers. For example, a plug-in might declare an extension point through which it can be provided with connectors to different backend data stores.

To illustrate this discussion with concrete examples, we can look at two extension points provided by the Eclipse Rich Client Platform (RCP):

- Eclipse provides an abstract file system API. An extension point[1] is used to extend this API with implementations mapping the abstract API to concrete file systems. Extensions to this extension point are on a lower layer than the abstract file system API.
- Eclipse's workbench (the central part of the RCP's user interface) can be extended with additional views[2]. Views are on a higher layer than the underlying workbench implementation, which defines the usage model and controls the life cycle of its views.

C. Open issues

The combination of different architectural styles requires these styles to be compatible with each other. In this paper, we have not discussed how to determine whether two styles are compatible.

On the topic of domain-specific specializations, this paper has described mainly how to document the architecture of a given platform as an architectural style. In practice, the more important question is how to derive an architectural style from requirements, before a concrete architecture is designed. For example, instead of describing the architecture of Eclipse RCP, an architect would want to find a suitable architectural style for plug-in-based rich client platforms.

[1]`org.eclipse.core.filesystem.filesystems`
[2]`org.eclipse.ui.views`

IV. Discussion

As we have seen, plug-in architectures provide several benefits such as extensibility (because additional functionality can be contributed via plug-ins), flexibility and exchangeability (because platforms need not be monolithic but can themselves be comprised of plug-ins), reuse (because plug-ins can be reused in different applications), and interoperability and integration (because a platform can provide domain-specific integration mechanisms). There are, however, also some potential downsides that should be considered.

- To enable extensibility and exchangeability, the interfaces between plug-ins must be published. Publishing an interface means that the interface cannot be changed any more without breaking compatiblity [5]. The desire to use the plug-in platform's extension mechanisms as much as possible conflicts with the recommendation of publishing as few interfaces as possible. Therefore, architects are forced to balance the flexibility of being able to change interfaces against the flexibility offered by extensibility.
- Efficiency may be lower due to the need to dynamically look up available extensions.
- Plug-in-based systems are more difficult to test because of the high number of potential configurations.
- Extensibility by third parties may raise security concerns, especially if extensions can be installed by end users.
- Deployment can become harder because it becomes more difficult to ensure consistency of different installations. Again, this issue is further complicated if end users can install additional extensions.

V. Related work

In [10], Szyperski discusses the idea of component frameworks. He describes a component framework as "a software entity that supports components conforming to certain standards and allows instances of these components to be 'plugged' into the component framework." [10, p. 425]. A plug-in (or set of plug-ins) which implements a domain-specific platform as described in this paper could be seen as an example of a component framework.

The extension-points-and-extensions extension model was popularized by the success of the Eclipse platform, but has also been adopted by other plug-in platforms. For example, Wolfinger et al. have developed a plug-in framework for .NET which is also based on the concept of extension points and extensions (called slots and extensions in their framework) [11].

As an alternative to the extension-points-and-extensions model, a service-oriented extension model can be used. Instead of declaring an extension point, a plug-in can declare and publish the interface it expects services to implement; and instead of declaring an extension, a plug-in can provide a service. Bartlett compares these options in the context of Eclipse in [1].

VI. Conclusion

Plug-ins are an important mechanism for independently extending applications with additional functionality. Applications built with a pure plug-in architecture take the concept of plug-ins one step further by removing the conceptual boundary between the host application and its plug-ins: the application is assembled from plug-ins, and plug-ins are hosts for other plug-ins.

In the context of a pure plug-in architecture, plug-ins are software components that can interact via the architecture's extension model. The design rules that these architectures follow are captured by the pure plug-in architectural style, which describes their architectural elements and constraints.

Plug-ins can also be used to implement extensible platforms for domain-specific applications. To describe the architectures of those applications, specializations of the pure plug-in architectural style can be created. One focus of our future research will be to address the question of how such architectural styles can be designed based on the requirements of the domain.

References

[1] Neil Bartlett. A comparison of Eclipse extensions and OSGi services. Published online, URL http://neilbartlett.name/downloads/extensions_vs_services.pdf (last retrieved 2008-09-10), February 2007.

[2] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, Massachusetts, 2nd edition, 2003.

[3] Dorian Birsan. On plug-ins and extensible architectures. *Queue*, 3(2):40–46, 2005.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons Ltd, Chichester, 1996.

[5] Martin Fowler. Public versus published interfaces. *IEEE Software*, 19(2):18–19, 2002.

[6] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*. Addision-Wesley, Boston, 2003.

[7] O. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, and T. Watson. The Eclipse 3.0 platform: Adopting OSGi technology. *IBM Systems Journal*, 44(2):289–299, 2005.

[8] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison-Wesley, Amsterdam, 2005.

[9] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[10] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, London, 2nd edition, 2002.

[11] Reinhard Wolfinger and Herbert Prähofer. Integration models in a .NET plug-in framework. In Wolf-Gideon Bleek, Jörg Raasch, and Heinz Züllighoven, editors, *Software Engineering 2007*, volume 105 of *GI-Edition Lecture Notes in Informatics*, pages 217–229, Hamburg, mar 2007. Gesellschaft für Informatik e.V.