Software Extension Mechanisms

Benjamin Klatt and Klaus Krogmann
Chair for Software Design and Quality (SDQ)
Institute for Program Structures and Data Organisation (IPD)
University of Karlsruhe, Germany
Email: {klatt, krogmann}@ipd.uka.de

Abstract—**Industrial software projects not only have to deal with the number of features in the software system. Also issues like quality, flexibility, reusability, extensibility, developer and user acceptance are key factors in these days. An architecture paradigm targeting those issues are extension mechanisms which are for example used by component frameworks. The main contribution of this paper is to identify software extension mechanism characteristics derived from "state-of-the-art" software frameworks. These identified characteristics will benefit developers with selecting and creating extension mechanisms.**

I. Introduction

In 1979, David Parnas [1] wrote about the advantages of extensible software and why developers should start adopting to this. He already named typical problems of software engineering like:

- Delivering an early release with a subset of features.
- Adding simple features without enormous code changes.
- Removing or adding functionalities for product variants.

These are only examples for the need of developing flexible and reusable software in a fast and reliable way. In recent years, a lot of research has been done in this area. Object Oriented Programming (OOP), Component Based Software Development, Service Oriented Architectures (SOA), Model Driven Software Development (MDSD) and a lot more have influenced the software development discipline. Extension mechanisms are no fundamentally new innovation. They have existed for a long time and are a design decision and combination of existing techniques. Extension mechanisms can vary a lot in their features and implementation strategies, but they all are sharing the intention to make a software extensible.

On the one hand, providing a good extension mechanism can lead to a successful and widely accepted product or framework (e.g. OSGi [2], Eclipse [3], Typo3 [4]). On the other hand, a bad one can restrict the evolution of a software, increase costs for maintenance and support and lead to rejection by developers and users.

This paper results from an analysis of general issues on extension mechanisms in a product-independent way. Only certain aspects of extension mechanisms have been considered in the existing literature (e.g. evolution [5] or life cycle [6, pp. 459]). In this paper, we provide a broad overview on important characteristics, when designing or selecting an extension mechanism.

This paper uses three key terms. An *extension point* is the definition of the provided interface for extensions. An *extension* itself is an implementation according to an extension point; equal to an implementation of a component. And an *extension mechanism* includes the system support and environment of a software allowing it to be explicitly extensible. In this paper we also review the relation to the environment and to the context of the extension mechanisms.

The contribution of this paper is the identification of different characteristics of software extension mechanisms. These characteristics are a result of an thorough analysis and practical evaluation of existing solutions. We investigated the widespread used solutions OSGi [2], Eclipse [3], Typo3 [4], and osCommerce [7] to find out commonalities and unique features of their extension mechanisms. To allow a better overview on available options, we organised the results in a classification tree.

However, we do not claim to offer a comprehensive overview on all existing software extension mechanisms.

The remainder of this paper is structured as follows: In Section 2 present some related work to this topic. In Section 3 we discuss different characteristics of extension mechanisms and present the classification developed. Section 4 gives a conclusion and sums up the results of this paper.

II. Related Work

Parnas [1] stated why extension mechanisms are required and what the challenges and advantages are. He introduced many design principles to consider. Well designed extension mechanisms are an application of his ideas and add results from the research of the last years as for example from the related work below. An overview on component frameworks also including extension mechanisms can be found in [8].

The design of *Application Programming Interfaces (API)* and *Service Programming Interfaces (SPI)* define the contract between an extension mechanism and the implemented extension itself. For this, they are key factors for good extension mechanisms. They define how extensions execute a part of a system like a library (API) or how extensions are executed by the system in case of a framework (SPI), respectively. Joshua Bloch has summarized a good set of principles for API design [9].

Frameworks have been developed to simplify repeated work and to support a short time-to-market development. They combine generalised code and components, best practices, and API/SPI design. There are different specifications for the term "framework" as for example in [10, pp. 552] or [11, pp.

340]. In this paper we separate frameworks from libraries very similar to [11, pp. 340] and [11, pp. 456]. The main difference is that frameworks provide a basic application that is the main executed part of the system. Libraries are called and not the leading part in the system. Extension mechanisms are included in most of the existing frameworks and thus a lot of experience can be gathered from the work done in this field.

Component based software development [12] must make use of software extension mechanisms. The components are extensions themselves and the extension mechanism consists of connecting the extension and the extended system using well defined interfaces. Some software including an extension mechanism, or at least the subsystem of the software connected to the extensions, can be handled as a component based software system. Often, the extension mechanisms are provided by component frameworks like Eclipse RCP or Enterprise Java Beans (EJB) but the design of extension mechanisms is not necessarily related with components.

The concept of *software factories* is to use state-of-the-art development techniques to develop a modular software that can be assembled by the vendor according to the customer specific requirements. The goal is to optimize the development life cycle and the reuse of existing components. Jack Greenfield has written one of the first books [13] about this concept. Extension mechanisms can be used as the foundation for customer specific software assemblies.

Extension mechanisms are not a new idea and every software that can be extended not only by changing its code provides something like an extension mechanism. Most of them are *product specific extension mechanisms* (i.e. [14]) and with a wide range of quality. In any case, they can provide a lot of experience for how to design an extension mechanism.

Klatt [15] classifies existing products according to the characteristics presented in this paper. Here, OSGi, Eclipse, Typo3, and osCommerce are discussed in detail and some guidelines for implementing extension mechanisms are given.

III. Characteristics of Software Extension Mechanisms

This section describes the characteristics of extension mechanisms identified by the analyses of the investigated systems OSGi, Eclipse, Typo3, and osCommerce. The characteristics can be specific to an extension mechanism as well as important for a whole software development process. They will be discussed for both aspects.

To visualize the characteristics, feature diagrams as defined by Czarnecki and Eisenecker [16] are used in this paper. Figure 1 gives a short introduction to reading feature diagrams.

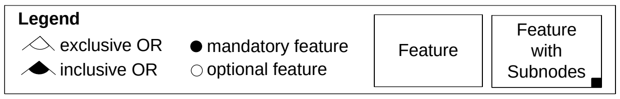

Fig. 1. Feature Diagram Legend

Figure 2 gives a top-level overview of the characteristics. Which of the characteristics are most important depends on the goal to be reached with an extension mechanism for a specific software. However, all of them should be considered when designing a software with an extension mechanism. It may be enough to consider a characteristic as unimportant but at least this should be done.

A. System Access

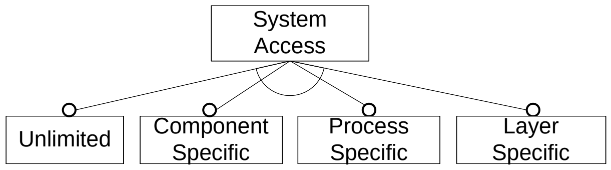

Fig. 3. System Access

A major decision when designing a software with an extension mechanism is the system access for the extensions – the way an extension can access the rest of the software system. As shown in Figure 3 this can reach from extensions on a specific *layer*, as for example the presentation layer, to the point of the system core. Additionally, extensions can be *specific to a component* or to *a process* (such as a business process). Accessing business process logic can imply access for multiple components or layers at once.

The deeper the extension mechanism is integrated in the system the more flexible the system can be extended, but also issues like complexity, security and reliability have to be handled more carefully.

In OSGi, access is not restricted by the framework itself; no pre-defined layering exists. It is up to the software developer to define them. In Eclipse access is restricted by extension points which exist for every extension. It allows designing components, processes, and layers in an arbitrary way. For Typo3, the extension API is fixed within pre-defined layers, but own extension points (so-called *hooks*) can add different architectural styles and architectural extensions. The latter is then specific for a certain component or process. osCommerce does not limit access at all.

B. Execution Strategy

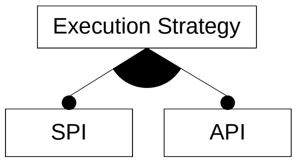

Fig. 4. Execution Strategy

There are two different *execution strategies* for extensions. Either the extension is executed by the system or the extension executes the system or at least a part of it. Those two different strategies are also known as *Service Programming Interfaces (SPIs)* respectively *Application Programming Interfaces (APIs)* (Figure 4). A framework for example makes use of extensions implemented according to a Service Programming

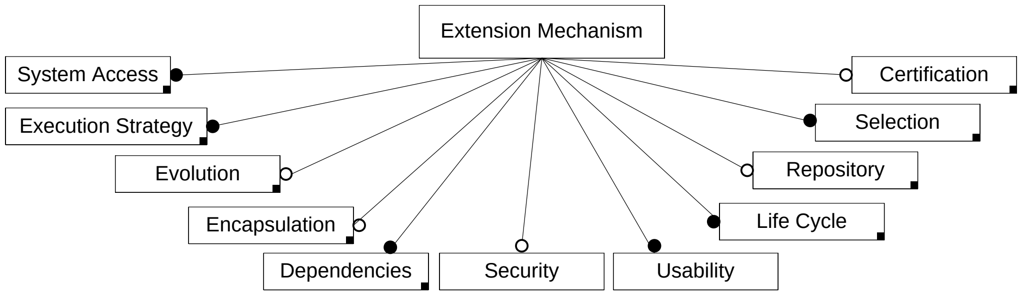

Fig. 2. Top-Level Characteristics

Interface. It represents a base application that calls the installed extensions. Libraries provide an Application Programming Interface to be executed by an extension which is the main executing part. This is like a user interface or scheduler executing the application.

In OSGi and Eclipse extensions are usually called by the framework. Patterns like publisher and subcriber allow changing the control flow. For Typo3, extensions are embedded into a whole, fixed call stack. This stack involves portions of system actions before and after execution of an extension. Usually, extensions are called by the framework; inverting the control flow is only seldomly used as it is not supported by the framework. osCommerce does not restrict the execution strategy.

C. *Evolution*

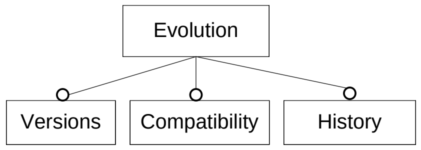

Fig. 5. Evolution

The evolution of extensions as well as of the extension mechanism itself is an issue as it is with any software system. Figure 5 presents key factors for the evolution. The extension mechanism itself has to be as stable as possible or at least backward compatible. Otherwise all existing extensions will not work anymore.

For the extensions, adding functionalities to an API or removing some from an SPI should not change the execution behavior or the feasibility of an extension [12, pp. 83] because of a still valid set of functionalities. But the other way around, to add required calls to an SPI or remove some from an API should be avoided to ensure backward *compatibility*. Otherwise migration support has to be provided.

The extension dependencies section III-E discuss another aspect that is tightly connected to the evolution of extensions. Managing the extension *history* becomes necessary if it is required to get back to an older *version*, e.g. if an invalid new extension was installed or a product bundle with an older functionality set should be packaged. The extension history is also necessary if the mechanism has to be able to support parallel instances of different versions of the same extension, e.g. for differing extension dependencies. Furthermore, the history provides a good feedback about an extension's development activities for the one who has to select an extension.

So-called *manifests* in OSGi allow explicit versioning of extensions. Eclipse allows additional descriptors and also checks versioning on installing new extensions. Here, an update mechanism exists to update existing extensions. Eclipse also supports multiple versions of the same extension (*plugin*) being installed at the same time. For Typo3, descriptors explicitly capture versioning information. Versioning is checked on installation and execution. An update mechanism is built-in to ease updating extensions. Only one version of an extension is allowed at a single point in time. osCommerce has no built-in versioning. Instead, versions are managed through external repositories.

D. *Encapsulation*

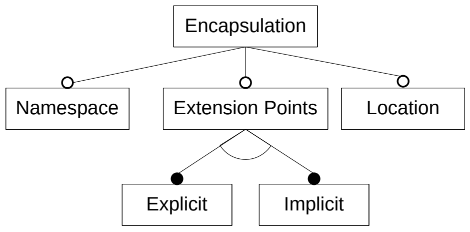

Fig. 6. Encapsulation

The encapsulation is a characteristic to separate different extensions from each other and/or from the main software providing the extension mechanism. In Figure 6 the three aspects namespace, extension points and location are visualized.

Namespaces are used for the identification of extension artifacts. This includes objects as well as configurations or any other elements that should be identified as connected to an extension.

Extension points represent a well defined (provided) interface between the extension and the extended system or other extensions. As extension points are clear interfaces, they can support the architectural quality. Extension mechanisms can provide facilities to describe extension points *explicitly* or they can be described informal e.g. within a system documentation (*implicit*).

Explicit Extension Points provide the possibility to create a test framework that implements an extension point with the capability to test the extensions without the whole extended software in place. Furthermore, explicit extension points become important for packaging releases with a subset of functionality. This can be used either for early releases or for reduced variants in conjunction with product line development [13].

The *location* characteristic is important to identify the resources of a specific extension.

Without any of these features, extensions are just direct code manipulations in the core software. In this case the extension can only be identified by the code difference of the old system and the new system.

OSGi and Eclipse allow sealed packages (Java capability), locations for finding and loading extensions can be specified, and namespaces are supported. Eclipse additionally allows restricting access to extension points or dedicated exported packages. Access rules can be defined in the so-called *plugin.xml* file. Typo3 expects extensions at a fixed location. Extension points (realised as low-level configurations) manage access visibility. Namespaces are supported but are not hierarchical. In osCommerce, specific modules have conventions for placing files. Additional encapsulation is not supported.

E. Dependencies

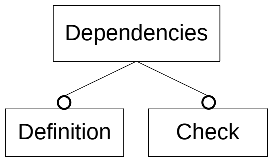

Fig. 7. Dependencies

An issue that needs to be handled explicitly are dependencies between extensions. They are stated in required interfaces and resources dependencies. For example multiple extensions may need to access the same resources. Additionally, it is not only possible to design an extensible software, but also to design extensible extensions. This brings in a lot of flexibility but increases complexity as well. As extensions can change independently, it is important to take care for this dependencies to not get too complex or impossible to be resolved. This can happen if two extensions A and B require different versions of a third extension C but the software supports only one instance for each extension. This may result in an irresolvable problem.

To handle this, two basic facilities exist, making these dependencies explicit (Figure 7). Dependencies *definition* makes it possible to better identify them for the user and automate resolving for the extension mechanism. For example the extension mechanisms of OSGi, Eclipse and Typo3 have all a facility to list the dependencies in their extension descriptors. *Check* of these dependencies by the extension mechanism is an additional feature making it possible to handle more complex dependencies in an automated way even with more than two extensions involved. Eclipse and Typo3, both have a check mechanism integrated in their extension management which generates warning if any dependency is not fulfilled.

For OSGi and Eclipse manifest files explicitly declare dependencies among extensions (*bundles*). Extensions can require a specific version of an extension. Thus, defintion and check of dependencies are enabled. For Eclipse, there are additional checks available, if extensions are loaded via the repository (find/update mechanism). In Typo3, required plugins and their range of versions or exact version are specified explicitly. Dependencies are checked before loading. If they are not fullfilled, corresponding messages indicate the reason. osCommerce has no explicit dependency declaration and thus check support.

F. Security

Security deals with the protection of information stored in the software system. It is influenced by the data an extension provides access to, the rights to whom it provides this access and the data the extension is able to change. It has to be ensured that only those subjects have access to the data who are intended to. Also extensions should only be able to manipulate data they are designed for. A more sophisticated way is for example to provide a configuration for the data access in the extension mechanism.

OSGi and Eclipse use the encapsulation mechanism to ensure security. Eclipse additionally allows signing trustworthy plugins. Typo3 secures applications by limiting access depending on namespace declarations. Additional security is available through the PHP environment, Typo3 runs in. osCommerce relies on the PHP environment only.

G. Usability

Independent from the goal of an extension mechanism it is important to design it to be usable as intuitively as possible. If developers can use it with a clear understanding and without a huge training effort, this will support a broader acceptance and use. As with the API and SPI design, extensions have to be as easy to understand as possible. The already mentioned overview of principles on interface design given by Joshua Bloch [9] also applies to extension mechanisms.

OSGi provides basic means of structuring an application, but only low-level infrastructure is provided by the framework itself. Eclipse additionally supports plugins, rich client platform applications and a “full comfort” framework including utility classes and installation managers. Typo3 has a so-called kickstarter for extension development, a comprehensive framework including dozens of utilities and also features an installation manager. osCommerce has no specific framework usability features.

H. Life Cycle

The *life cycle* of an extension defines which states it can run through. Most component frameworks have such life cycles; sometimes with specific enhancements (e.g. [6, pp. 459]).

The states that extensions run through do not have to be explicit in a system but can be identified as follows:

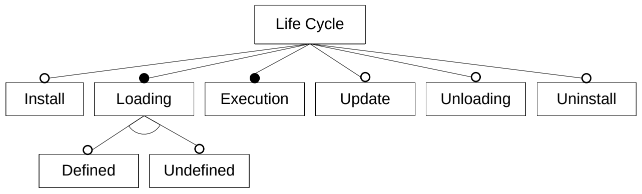

Fig. 8. Life Cycle

- *installation* The physical extension resources are brought into the system.
- *loading* The required resources and configurations are loaded by the active system instance. They are analyzed and initialisation steps might be performed.
- *execution* The real execution of the extension. This combines start, run and stop of processes.
- *update* Any changes of the extension configuration or resources are performed.
- *unloading* The resources and configurations are removed from the active system instance.
- *uninstallation* The physical extension resources are removed from the system.

The point in time where extensions are loaded can affect the start-up time of the system, the first execution time and/or each execution of an extension. It has to be clarified whether the start-up and reboot, the reaction time or for example a check with each execution are important or not. Depending on the implementation of an extension the loading strategy can influence the functional behavior, too. For example, an extension estimates being loaded once at system start-up time and prepares some global resources. But the extension mechanism reloads it with every execution and the extension tries to prepare those global resources over and over again and the system performance decreases. It is important to decide that extensions either have to be independent from the loading time or to define when the loading will happen.

The life cycle of OSGi, Eclipse, and Typo3 supports all mentioned features. osCommerce has a light-weight life cycle with only mandatory features.

I. Repository

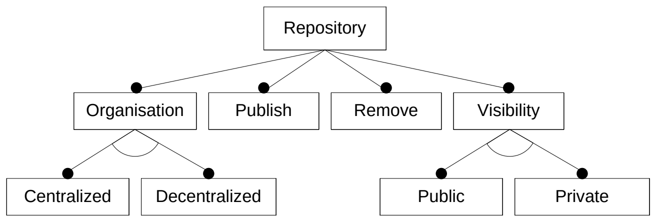

Fig. 9. *Repository* of an extension mechanism

Writing extensions not only for designing flexible software, but also for providing reusable components, requires a place to store those components. Such a place is called an extension *repository*.

Depending on the extensions and the designed extension mechanism, the repository can range from a simple directory in the file system to a web based, deployment supporting service point. It at least has to support publish and retrieve actions.

Specifying a repository requires to define its *organisation*. It can be built either *centralised* or *decentralised* according to the typical advantages and disadvantages of creating a bottleneck or loosing control of a consistent structure. Another aspect is the *visibility* of the repository itself, the list of all extensions or some specific extensions. Depending on the requirements it might makes sense to introduce a separation of *public* and *private* visibility.

A well designed repository is also influenced by other characteristics identified in this paper like version control, life cycle, selection, and certification (see below).

OSGi has no dedicated repository support. Eclipse has a decentrally organised repository structure. Its framework does not support privacy or remove. Support of the latter features is up to a *update site* maintainer. Typo3 by default has a centralised repository, but does not support private extensions; otherwise it has fully repository support. osCommerce uses a centralised website as repository to publish and remove extensions.

J. Selection

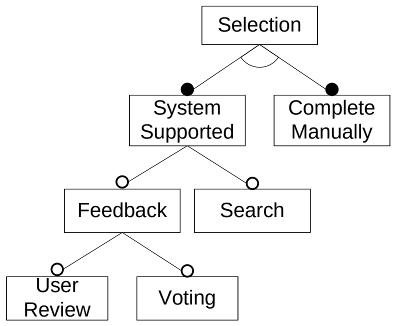

Fig. 10. Component *selection* from a repository

Selecting a suitable extension can either be completely manual or supported by the extension mechanism that can access a repository (Figure 10). The selection consists of two major steps. First, potential candidates have to be found. Afterwords, the best one of them has to be chosen. With extension mechanism support a search can be provided for the identification of potential candidates. This can range from a simple keyword search to a semantic or context sensitive one (cf. [17]). To select the best candidate a feedback system can be implemented. Voting mechanisms or user reviews can provide information to support a decision. It is important to take functional as well as non-functional requirements into account. This is also tightly connected to the certification and repository characteristics of the extension mechanism [18].

OSGi does not support selection of extensions. Eclipse has numerous community websites to ease selection of extensions, but features no support by the framework itself. Typo3 provides search, browsing, and number of downloads statistics to select extensions. The framework integration is very lightweight. osCommerce allows browsing extension websites, its selection facilities are comparable to the ones of Eclipse.

K. Certification

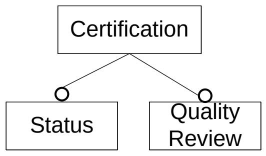

Fig. 11. *Certification* for extensions in a repository

The evaluation of candidates for a required extension may become an expensive or at least a time consuming task. Certifications can provide some metrics of quality attributes and well defined specifications to support this evaluation process. Certificates are not only a technical aspect. If business requires selling extensions, providing quality guarantees is likely to increase revenues.

Figure 11 shows two aspects of the certification. One is the status of the extension. Which identifies the maturity of the extension and provides information on whether it can be used for example in productive systems. Another aspect which can be found in certifications are reviews. Those reviews have to be performed according to a standard process but can provide a trustworthy feedback and quality assurance. Moreover, a manual review will lead to personal signature by the reviewer.

The drawback of certifications is the effort of the certification specification itself as well as the certification of the individual extensions. This effort makes especially sense for very often reused or commercial extensions [18].

OSGi, Eclipse, and osCommerce do not support certification of extensions. In the case of Typo3, extension from the central repository have a release status. Additionally, its extensions are partially reviewed by a dedicated security team and can get approved.

IV. Conclusion

Extension mechanisms can support the general requirements of faster, cheaper, more flexible, and evolving software development. Those goals are mainly achieved by the support of modularisation to break down complexity, distributed development, reusability and more controlled software evolution. With a good extension mechanism it is also possible to deliver early working releases with a subset of functionality and to develop product families.

Extension mechanisms can lead to a better software but only if they are done right. Vice versa, a bad extension mechanism can result in higher complexity, decreased efficiency and waning acceptance by the developers.

Most of today's software architects have already developed extensible software or at least have been in contact with one. Some of them may have taken care for issues covered by the related work but only a few have explicitly thought and talked about all characteristics of extension mechanisms. This paper presented a classification of characteristics which are important for them. The development of new software products or the refactoring of existing ones can use the presented characteristics to select and design appropriate extension mechanisms according to the requirements of a software.

Extension mechanisms are once more not the silver bullet but even if the challenges of software development are not solved, they are supported in a very advanced way. At least their concepts should be known and used by every software architect.

OSGi is a high-quality framework also suitable for other frameworks built on top of it (such as Eclipse), but lacking support of repository-related features. Web-frameworks like Typo3 are currently being developed to ease development of high-quality "web-architectures" beyond a web-based content management system. Eclipse has become a powerful framework inheriting the features of OSGi, but is (intentionally) missing central repository infrastructure and certification, which is available for Typo3. osCommerce lacks architectural support for a good extension mechanism. Here, extensions are realised in an ad-hoc manner. Little control and community support ease quick-wins when implementing extensions while potentially originating maintainability issues in the future.

For future work, we like to extend the number of reviewed frameworks supporting extension mechanisms to allow a comprehensive market overview. Additionally, we are planning to extend the feature tree to capture more aspect of extension mechanisms.

References

[1] D. L. Parnas, "Designing software for ease of extension and contraction," in *Proceedings of the Third International Conference on Software Engineering*, 10-12 1978, pp. 264–277.
[2] O. Alliance. (2008) Osgi alliance — main / osgi alliance. [Online]. Available: http://www.osgi.org
[3] E. Foundation. (2008) Rich client platform - eclipsepedia. [Online]. Available: http://wiki.eclipse.org/Rich_Client_Platform
[4] T. Association. (2008) Typo3 content management system. [Online]. Available: http://www.typo3.org
[5] A. Stuckenholz, "Component evolution and versioning state of the art," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 1, p. 7, 2005.
[6] J. McAffer and J.-M. Lemieux, *Eclipse rich client platform*. Addison-Wesley, 2006.
[7] osCommerce. (2008) oscommerce - open source e-commerce solutions. [Online]. Available: http://oscommerce.org/
[8] K.-K. Lau and Z. Wang, "A taxonomy of software component models," in *Conference on Software Engineering and Advanced Applications. 31st EUROMICRO*, September 2005, pp. 88–95.
[9] J. Bloch, "How to design a good API and why it matters," in *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2006, pp. 506–507.
[10] V. Claus and A. Schwill, *Duden Informatik A - Z*, 4th ed. Dudenverlag, 2006.
[11] H.-J. Schneider, *Lexikon Informatik und Datenverarbeitung*, 4th ed. Oldenbourg, 1997.
[12] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. New York, NY: ACM Press and Addison-Wesley, 2002.
[13] J. Greenfield and K. Short, *Software factories*. Wiley, 2004.

[14] A. S. Incorporated, *Third-party plug-ins for Photoshop*, Adobe Systems Incorporated, August 2008. [Online]. Available: http://www.adobe.com/products/plugins/photoshop/

[15] B. Klatt, "Software Extension Mechanisms," Fakultt fr Informatik, Karlsruhe, Germany, Interner Bericht 2008-08, 2008. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000009113

[16] K. Czarnecki and U. Eisenecker, *Generative programming*. New York, NY: Addison Wesley, 2002.

[17] S. Overhage, "Towards a Standardized Specification Framework for Component Development, Discovery, and Configuration," in *Eighth International Workshop on Component-Oriented Programming (WCOP2003)*, 2003.

[18] A. Alvaro, R. Land, and I. Crnkovic, "Software component evaluation: A theoretical study on component selection and certification," *MRCT Report*, 2007.