Encapsulating Crosscutting Concerns in System Software

Christa Schwanninger, Egon Wuchner, Michael Kircher
Siemens AG
Otto-Hahn-Ring 6
81739 Munich
Germany
{christa.schwanninger,egon.wuchner,michael.kircher}@siemens.com

ABSTRACT
System software has to encapsulate crosscutting concerns properly. Aspect Orientation (AO) is a paradigm that supports modularization of crosscutting concerns. But as AO is relatively new it still lacks support suited for the industry in many domains, e.g. support for the programming languages C and C++ which are heavily used in the embedded domain exists but not yet in the desired scope and quality. To compensate for missing tools and languages we need architectural solutions for the problems around crosscutting concerns. Different system software layers, starting from simple libraries to full blown component containers can be used to provide support for concerns that cut across whole applications. Patterns can help to establish good architectures for this purpose. This position paper briefly describes how design patterns can be evaluated for their suitability to solve problems caused by crosscutting concerns.

Keywords
System Software, Patterns, Frameworks, Components, Aspect Oriented Programming.

1 INTRODUCTION

This position paper documents experiences in building run-time system software in several domains. The company, for which the authors work, typically does not build commercial off-the-shelf (COTS) system software, but it develops software for its hardware products. Those hardware products stem from several domains, including telecommunication, medical systems or automotive systems. Associated with the hardware product families, are the software product families, needed to operate the hardware. Such software product families need to be supported with frameworks or even custom made component containers, which play the role of system software for the software application developers. As system software they foster reuse and help to develop good software in short development cycles. With our experience in building platforms for product families we want to contribute to the field of system software.

During the last years, the authors saw several attempts to build frameworks for system families fail, because the architects of the frameworks were not aware of the crosscutting concerns in the system. Because the project did not capture and localize the concerns in the architecture, the project faced several problems, such as redundant implementations of the same functionality, wastage of system resources, missing resource consumption traceability, uniform error handling and communication strategies resulting in cumbersome integration.

This paper describes how crosscutting concerns can be captured and localized in system software and how patterns can help to build software that separates concerns properly.

Section 2 will explain our view on system software. Section 3 lists the software artifacts used to localize crosscutting concerns, while section 4 enumerates selected patterns for building architectures considering crosscutting concerns. The paper concludes with a brief discussion of related work and a conclusion in section 5 and 6, respectively.

2 SYSTEM SOFTWARE

According to [FODC00], system software is defined as: "Any software required to support the production or execution of application programs but which is not specific to any particular application."

System software can be aligned in to two categories:

- **Production software** – Production software includes tools that help developers in the process of designing, writing and managing software e.g. compilers, linkers, debuggers, profilers or complete IDEs, but also version control, building tools, tracers, runtime checkers and analyzers,
- **Run-time software** – Software that is needed for execution of applications at run-time or integrated in the application, e.g. OS, supporting libraries, middleware, services like persistency or event services, frameworks or even component containers, that offer their own runtime environment.

Figure 1 shows typical layers in software. The layers range from application software, to middleware, to the operating

system. Besides those layers, also the supporting compilers, configuration management software, etc. is considered as system software.

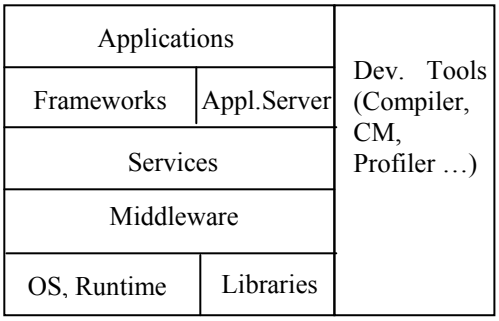

Figure 1: System Software

Software production tools, such as compilers and profilers, are "stand-alone" applications and are usually not part of any delivered system in our organization; therefore they are not of interest for us in the context of this paper.

System software that runs or is part of the application software, such as frameworks and component containers, faces different challenges than stand-alone software. It has to be built for reuse in various projects, or even domains, of which many requirements are not known up front. Additionally, the run-time system software has to be built to integrate into other software.

System software, in our context, mainly deals with resource provisioning and management (OS), communication (communication middleware), event handling (application frameworks), and GUI management (GUI frameworks).

In the next chapter we give an overview on the different kinds of run-time system software and explain how they can be used to support the localization of crosscutting concerns.

3 CAPTURING CROSSCUTING CONCERNS IN SYSTEM SOFTWARE

Depending on the layer, shown in Figure 1, and the domain it is used in, system software has to handle one or several of the following crosscutting concerns:

- Adaptability to application needs, e.g. configuration of middleware, and exchangeability.
- Optimized resource management, e.g. memory management or thread management.
- Transparent, non-invasive inter-process and network communication.
- Initialization and destruction for efficient start up and secure shutdown in resource restricted systems
- Event dispatching and handling

The listed crosscutting concerns (also referred to as aspects) are non-functional. Many domains also have additional functional aspects, for example mobile phones require messages to be passed without copying of the message data, or the sharing of personalization information across all applications in an automotive multimedia system.

Generally, AO tries to achieve the following goals via encapsulation and localization of crosscutting concerns (CCC):

- Modularity – The code for one CCC should be located in one source code file.
- Uniformity – A CCC should be treated uniformly in the whole application.
- Non-invasiveness – It should be possible to change or extend the implementation of the CCC non-invasively.
- Transparency – The CCC should be transparent to the developers.
- Reusability – Reusable software components have to be developed that can not know about the environment and the crosscutting concerns they will be reused for.

The previous two lists show the big overlap between the problems faced in system software and the promised solutions of AO.

Encapsulation of Crosscutting Concerns

Once the crosscutting concerns are identified, there are several ways how to capture them in an architecture:

The simplest way for handling crosscutting concerns is to provide an implementation in form of a *library* together with guidelines how to properly use this functionality. This is something that is usually done for simple crosscutting concerns such as tracing and logging, but also for resource management, where a library is provided that is the only access point for acquiring and releasing a specific resource.

Libraries offer a collection of functions for dealing with crosscutting concerns, *frameworks* do more. They not only provide reusable code, but also influence the architecture, for example by the inversion of the control flow. Also, frameworks often address several related functionalities, e.g. GUI frameworks implement GUI elements and the mechanisms to deal with user events. Frameworks need to be extensible, therefore they typically are built using patterns, like Strategy and Interceptor, which allow framework users to extend and customize the framework functionality.

Component containers are advanced frameworks, separating technical concerns, such as resource and

lifecycle management, from business concerns, containing the actual logic and functionality. They provide a run-time environment for components that relieves the developer from the technical concerns. Commercial component containers are often only suited for business or finance applications because they mostly cover only enterprise-specific technical concerns, but not those of typical embedded software or at least not as configurable or lightweight as required.

Aspect-oriented (AO) programming seems to be the most appropriate way of implementing crosscutting concerns in a modular way. AO brings a number of advantages. Appling AO crosscutting concerns can be modularized in exactly one place, they can be weaved in or out as needed, and their implementation and application is transparent to the developer. On the downside, AO is a rather young paradigm and there are not enough proven languages and tools on the market, yet. Except AspectJ [Kicz97] [Referenz to AJDT] no language can claim to provide industrial strength stability and tool support. AspectJ is an AO extension to Java, especially in embedded systems the dominant languages are C and C++. AspectC++ is a noble attempt to provide the same functionality for C++ as AspectJ does for Java, but the language and the tools (a plug-in for an MS IDE) are not widely used and can't be considered stable enough to implement critical features in reusable system software.

When trying to achieve the goal of reusability for a family of applications, traditional platforms define extension points where the application developer plugs in application logic in a prescribed way usually through base classes, interfaces and templates. System software defines a contract; applications use its functionality by fulfilling their part of the contract. For typical framework approaches the application has to know how to handle the system software, but not vice versa. The programming model of AspectJ like languages is different. Since the connection between the aspect and application code often requires detailed knowledge of the application code, it is a lot harder to pre-implement generic, reusable system software.

Further, how will the quality of the resulting software be ensured after introducing so many variation points? The original assets – the software that should be augmented by an aspect - are typically not designed to be extended; for example join points are defined only later, independent of the software to be extended.

So other alternatives are needed, as long AO, as the most appropriate way to modularize crosscutting concerns in system software, is not mature enough to get 'picked'.

4 PATTERNS FOR BUILDING EXTENSIBLE ARCHITECTURES

Since AO is still in its infancy, but crosscutting concerns have to be handled properly, we evaluate how patterns, as alternative concepts, can be used to build libraries, frameworks, and component containers, which fulfill the requirements like non-invasiveness, exchangeability, reusability, and modularity for crosscutting concerns. This is an 'architectural approach' to solve crosscutting concern related problems. In a first step, we study the rich pattern literature to find design and architectural patterns that help to address the above mentioned requirements.

The table on the last page shows part of our current state of evaluation of design and architectural patterns regarding their usefulness to capture crosscutting concern related problems. All selected patterns touch the area of extensibility and/or integration of concerns, which were our selection criteria.

For AO it is not relevant if the reason for encapsulating a crosscutting concern is to make the implementation easily exchangeable, to make the encapsulation transparent, or to make the encapsulated concern reusable. With AO mechanisms they are all addressed at once. Investigating design patterns shows, that they are focused on specific problems of separating the concern. But this is not really a problem, since often, only one or a small number of the above mentioned requirements have to be fulfilled at the same time. Applying one or two patterns is often sufficient to solve the specific problem related to a crosscutting concern.

For example Decorator [GOF95] helps to add functionality transparently without changing the decorated class and thus can be used to add part of a crosscutting concern implementation without polluting the original class. Additionally, an Abstract Factory [GOF95] helps to hide the decorated functionality from the client.

If the goal is reuse the crosscutting functionality the above combination of patterns can not be used, since the decorator class has to provide the same interface as the decorated class. The concern implementation therefore needs to be encapsulated, for example by a Strategy [GoF95]. Further, if the concern is resource management specific, one or several patterns of [POSA3], such as Pooling or Caching can be used directly.

Because patterns (can only) address specific forces in encapsulating and localizing crosscutting concerns, they have to be categorized accordingly. This is the intend of the attached table. The table contains the following information:

- Patlet: a short description of the pattern.
- Addressed problem: what is the main problem the pattern solves?
- Modularity: does the pattern help modularize a crosscutting concern; how does it help?

- Uniformity: does the pattern help implement a CCC uniformly throughout a system?
- Non-invasive exchangeability and extensibility: does the pattern help to exchange the crosscutting concern implementation without having to change all the places the concern crosscuts?
- Transparency: does the pattern help to keep a concern implementation and application transparent to the application developer?
- Reusability: does the pattern support the reusability of the concern code and/or of the component code that is crosscut by the concern?
- Improvability with AO (AspectJ): could AO improve the implementation of the pattern? Or does AO make the pattern obsolete?
- Possible solution in AspectJ: describes the AspectJ means we would use to implement the patter

For every pattern one to three "+" say how well it is suited to address a specific problem, a "-" indicates that the pattern is not suited at all to solve the problem. The additional text explains the rating. Several patterns fulfill one or several of these criteria, but none of them fulfills all of them the same way AO does. Also, many patterns that are useful in localizing crosscutting concerns can further benefit from an AO implementation, as can be seen in the last but one column of the table.

Let's take the Strategy pattern [GoF95] as an example. Strategy encapsulates application logic and makes it exchangeable transparently. It pretty well modularizes the logic, but still depends on the state of the entity to be extended (Modularity: ++). It is not meant to be used to solve one crosscutting concern uniformly over a whole application, rather targets one specific task (Uniformity: -) but it keeps exchanging of the contained logic perfectly transparent to clients (Non-invasiveness +++). The implementation of Strategy is not completely transparent to the client, since the client has to hold an instance and trigger the Strategy's functionality (Transparency +). A strategy requires state information and can only be reused if the required state is provided (Reusability +).

We want to continue evaluating patterns for a pattern catalogue that is dedicated to problems related to crosscutting concerns only. The next step after that will be the evaluation of successful product family frameworks to find the best practices for encapsulating crosscutting concerns in design and architecture beyond the currently documented patterns.

By recovering how to capture and localize crosscutting concern in system software by 'traditional' means, we hope to also learn more about how to use AOP for capturing and localizing crosscutting concern in system software in the future.

5 RELATED WORK

Jan Hannemann and Gregor Kiczales implemented all 23 GoF design pattern in AspectJ [Han02] and found out that modularity and reusability were improved with AspectJ [Kicz97] remarkably.

Books like Patterns for Concurrent and Distributed Objects, [POSA2], Patterns for Resource Management [POSA3], or Security Patterns [SPC02], are a few examples for pattern collections and languages that offer solutions to problems that partially stem from crosscutting concerns in specific domains.

The work of Eide et al [ER+02] analyzed patterns regarding their static and dynamic structures. As solution the authors suggest to make participants of pattern implementations easier to exchange, based on the understanding that participants in the pattern literature [GoF] can only be objects.

6 CONCLUSION

In this paper we gave a brief overview over the different layers of system software and how they localize crosscutting concerns. We argued that AO is not yet mature enough to be used in the domains of interest to us. Thus we investigated how patterns can help to build crosscutting concern aware architectures for system software. We started to evaluate design and architectural patterns that can help building frameworks and component containers that solve the problems crosscutting concerns bring up. Doing this we raise the awareness of architects and designer for crosscutting concerns. This not only supports contemporary software development, but also paves the way for AO technologies in the future.

REFERENCES

[ERR+02] E. Eide, A. Reid, J. Regehr, and J. Lepreau, Static and Dynamic Structure in Design Patterns, ICSE 2002, 2002

[FODC00] "The Free On-line Dictionary of Computing" http://www.nightflight.com/foldoc/, 2004

[GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns-Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[Hann02] J. Hannemann, G. Kiczales, Design Pattern Implementation in Java and AspectJ. Proceedings of OOPSLA 2002

[Kicz97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda and C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect Oriented Programming. In Proc. of European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science Vol. 1241, pp. 220-242, 1997.

[PLOPD4] N.B. Harrison, B. Foote, H. Rohnert, “The Role Object Pattern” in Pattern Languages of Program Design 4, p.14-31

[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture – A System of Patterns, John Wiley and Sons, 1996

[POSA2] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, Pattern-Oriented Software Architecture – Patterns for Concurrent and Distributed Objects, John Wiley and Sons, 2000

[POSA3] M. Kircher and P. Jain, Pattern-Oriented Software Architecture – Patterns for Resource Management, John Wiley and Sons, 2004

[SPC02] Security Patterns Community: Security Patterns Homepage. http://www.securitypatterns.de, 2004

Pattern Name	Patlet	Addressed problem, domain	Modularity	Uniformity	Non-invasive exchangeability, extensibility	Transparency	Reusability	Comparison with AO (AspectJ)	Possible solution in AspectJ
Decorator	Attach additional responsibilities to an object dynamically. This provides a flexible alternative to subcalssing for extending functionality	extending existing functionality	+;each decorator encapsulates one concern for a single class, it is not suited to encapsulate concerns that span several classes	-; localized to one class	'+++; a decorator class can be directly exchanged within a chain of decorators, no effect on the decorated class	++ transparent for developer of original class but not at instantiation time	-, decorator has to implement the decorated class's interface	+++; with AspectJ no code changes necessary when inserting a new decorator class into a chain of decorators	comparable to before/around advice using the method arguments as pointcut context
Proxy	Provide a surrogate or placeholder for another object	transparent integration	+++, encapsulates additional functionality, but not meant to encapsulate crosscutting functionality	-, one proxy for several classes not applicable, since signature has to match	++, though proxy classes have to be instantiated instead of original class	+++	-, interfaces have to match	+++, adherence to interface not necessary, therefore reusable for several classes	all kinds of advice
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.	encapsulation of operations on tree structures	+++, Visitor encapsulates additional operations on an existing tree structure	++, only for the tree structure possible	++, extending the tree structure with a new node type requires the adaptation of the new class; new visitors can be added non-invasively	++, every class has to implement an accept method	-, visitor is specific to visited classes' functionality	+++, AO allows to implement the Visitor non-invasively even in case of extending the original class hierarchy	introduction of new method
Strategy	Make application logic exchangeable.	extension of base functionality	++, encapsulates specific code, but depends on the state of the entity to be extended.	-, just locally	+++, it is the main purpose to exchange application logic.	+, the original entity must foresee a hook.	+, if the state of the entity is represented similarly.	++, additional (specific to the extended application logic) state might get weaved in.	Introduction with new methods and all kinds of advices.
Interceptor	Allow functionality to be added transparently to a framework and trigger automatically when certain events occur.	extending functionality in call chains	+++, interceptor implementation can target several classes	+++	**+++**	+++	+++	+++ also execution join points possible	before and after advices with calls join points
Resource Lifecycle Manager	Decouples the management of the lifecycle of resources from their use by introducing a separate Resource Lifecycle Manager, whose sole responsibility is to manage and maintain the resources of an application.	encapsulated lifecycle management; domain: resource management	+, localizes the lifecycle management of one or several resources; transparently provides pooling and caching of resources; manages interdependencies transparently	++, resources are managed uniformly throughout the application	+++, allows to exchange the resource management strategies transparently; to support new types of resources, its interface might need to get extended.	+, resource users need to use the resource lifecycle manager instead of existing resource providers; changes to the strategies are transparent	++, the implementation of the resource lifecycle manager can get reused	-, aspects must have knowledge with regards to when and how resources are acquired or released; pointcuts are hard to define	replace existing acquisition and release calls with around advices
Policy Enforcement Point	Isolate policy enforcement to a discrete component of an information system; ensure that policy enforcement activities are performed in the proper sequence.	consistent enforcement of security policies; domain: security	+++; localizes policy related activities to one point	+++, guarantees uniform policy handling	+++, strategy easily exchangeable	- only transparent with additional framework support	+++	++ policy enforcement could be made transparent	introduction and before advices