Concern Hierarchies

Olaf Spinczyk, Daniel Lohmann, and
Wolfgang Schröder-Preikschat
Friedrich-Alexander University
91058 Erlangen, Germany
{os,dl,wosch}@cs.fau.de

ABSTRACT

About 30 years ago the pioneers of family-based software development invented very useful models. Today we would describe them as models that help software engineers to bridge the gap between variable requirements and the reference architecture of a product line platform. This is one of the key challenges in product line engineering. In this paper we revisit one of these models, namely the *functional hierarchy*. The goal is to derive a new model called a *concern hierarchy* that also takes today's knowledge about crosscutting concerns and aspect-oriented programming into account. The resulting concern hierarchy model facilitates the design of aspect-oriented software product lines by supporting the derivation of class hierarchies, aspects, and their dependency relations more systematically without being overly complex.

1. INTRODUCTION

The design of a software product line is much more challenging than the design of a single application. Many application scenarios (configurations) shall be covered by the same software components. Therefore, components often have to be designed and implemented in a generic way. Furthermore, instead of defining a fixed architecture, a reference architecture has to be developed that can be understood as a set of composition rules for the generic components.

A very important issue in this design process are dependencies between components. If, for instance, a component *A* depends on a component *B*, a composition rule has to be defined that guarantees that no product line variant can be configured that contains *A* but not *B*. Without such composition rule compile time error messages or even runtime errors would be the unpleasant consequence. Even more problematic are cyclic dependencies. If *A* depends on *B*, *B* on *C*, and *C* on *A*, there is almost no room for configuration. Any product variant has to contain either none of these components or all of them.

All these considerations are completely independent of the programming language and even independent of programming paradigms such as object-orientation or functional, imperative, and logical programming. They also do not depend on the actual mechanism that is used for the interaction between the components, such as local function call, remote procedure call, message passing communication, or even macro expansion.

This was the motivation for the program family pioneers from the seventies to abstract from all technical issues, when they designed their systems. The main goal was to get the dependency relations between the logical "functions" right. These models such as Parnas' "uses hierarchies" [16] or Habermann's "functional hierarchies" [12] are still highly influential. Their simplicity makes them attractive.

However, computer science made some steps forward during the last decades. The awareness that crosscutting concerns are a problem for reusability and extensibility as well as the notion of aspects that implement crosscutting concerns in a modular way, came up in the late nineties. Parnas and Habermann did not consider these problems in their work sufficiently. In our opinion it is necessary to revisit and update their work, as aspect-oriented product line engineering can hardly be done without these fundamental models.

The following sections are structured as follows: Section 2 will briefly introduce Habermann's functional hierarchies and discuss our experiences with system design based on this model. Section 3 is the main contribution of this paper. It contains an informal description of the extended functional hierarchy model that we call "concern hierarchy". In section 4 we will discuss how concern hierarchies can be used to derive an aspect-oriented class hierarchy as well as a dependency graph. The paper ends with a discussion of related work in section 5 and our conclusions in section 6.

2. FUNCTIONAL HIERARCHIES

Like many software engineering pioneers Habermann worked on operating systems. The inherent complexity of these systems – even in the seventies – almost automatically made computer scientists think about modularization and configurability in general.

2.1 The FAMOS Structure

Figure 1 illustrates the structure of his FAMOS System[1] as a functional hierarchy. The system is structured in layers. Each layer consists of functions. The term "function" is used in a very general sense and abstracts from the actual implementation and interaction mechanism. Each function knows the functions of its own layer and the functions from the layers below. This acyclic structure allows the hierarchy to be pruned at any layer and, thus, facilitates the

GPCE '06, First Workshop on Aspect-Oriented Product-Line Engineering, October 22, 2006, Portland, Oregon.

[1] Family of Operating Systems

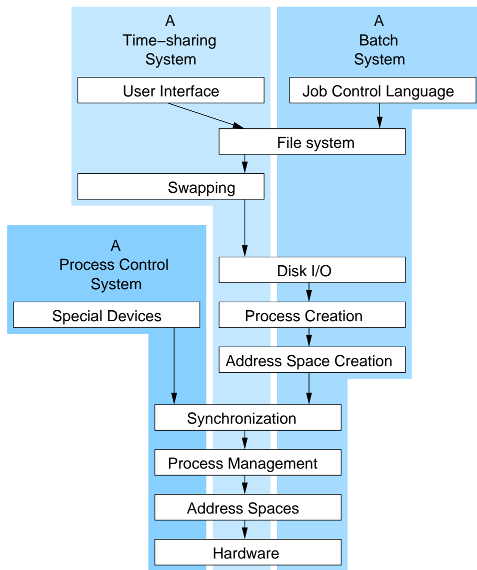

Figure 1: Functional hierarchy of the FAMOS operating system family

derivation of family members. In the case of the FAMOS operating system the lowest function represents the elementary operations provided by the hardware. Based on that are functions which implement *Address Spaces*, *Process Management*, and *Synchronization*. On top of *Synchronization* there is a branch in the hierarchy. With a "minimal extension", i.e. *Special Device* drivers, a *Process Control System* variant can be constructed. The other branch, which starts with dynamic *Address Space Creation*, is the base for the construction of a *Batch System* variant and a *Time-Sharing System* variant.

2.2 The PURE Structure

In the late nineties our research group designed and implemented a highly configurable operating system for the domain of deeply embedded systems. For this purpose we combined the family-based design approach known from FAMOS with a C++ implementation. The result was the PURE operating system family [6, 17].

Figure 2 shows the class diagram of the PURE thread management subsystem. It was derived from a *fine-grained* functional hierarchy in order to achieve a very high degree of configurability and thereby scalability of the memory consumption with the application's requirements. Each class implements a function from this functional hierarchy. Due to the duality of Habermann's incremental design approach and implementation inheritance in OO, it was natural to map the edges of the functional hierarchy to inheritance relations in the class diagram – at least as a rule of thumb. The result was a very deep class hierarchy that looks a bit like the corresponding functional hierarchy rotated by 180°. The static configuration of the system was based on two mechanisms:

1. **Application-Driven Configuration:** Operating systems for deeply embedded systems normally have to support only one specific application. A PURE operating system was used by applications like an ordinary static C++ class library. Hence, we could exploit the C++ compiler and linker for the static system configuration. For example, if the application only instantiated the class *Native*, the code of the classes *Bundle* and *Genius* would not be referenced and, thus, not linked into the final system.

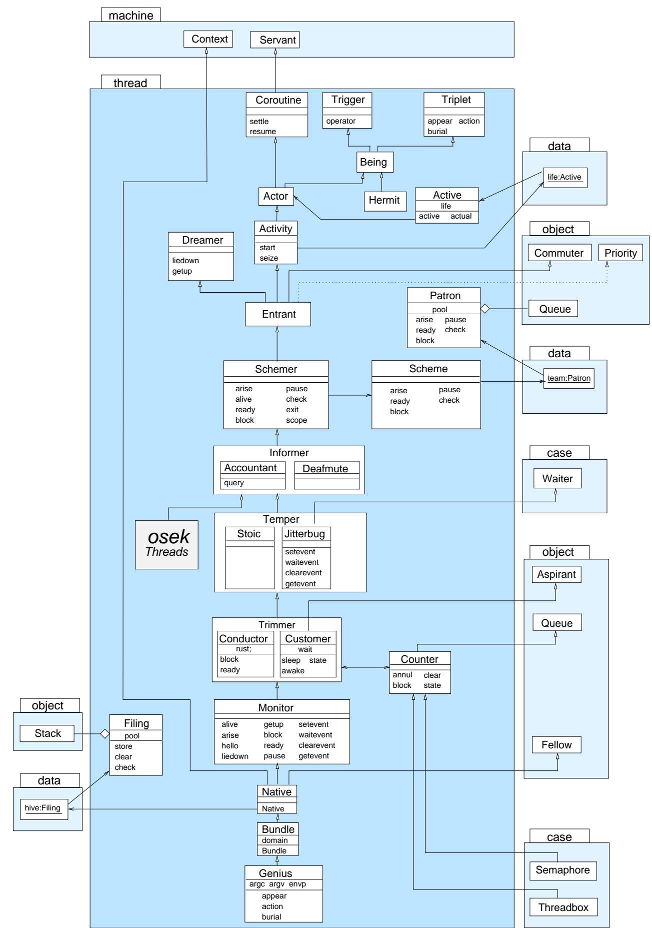

Figure 2: Class diagram of the PURE thread subsystem

2. **Feature-Driven Configuration:** In many cases we experienced the need to statically configure the implementation of a certain layer. For example, each thread object contains some state information that depends on the thread scheduling strategy. While priority-based strategies require a thread priority value, a simple FIFO strategy only requires a pointer to the next thread object. Therefore, a layer was often implemented by a number of classes and a configurable "class alias" that can be used by the next layers to access the code and data members of the configurable layer. Technically, a class alias is a C++ typedef that points to 1 of N classes with alternative implementations of the same abstract function. In order to statically configure these class aliases, the variability was described by a *feature model* [9]. A configuration tool allowed users to select a valid configuration by marking features. The feature selection was used to generate the necessary class aliases. For instance, in the class diagram *Informer* is a class alias that could be configured to be either an *Accountant* or a *Deafmute*. Both classes don't have to be interface compatible. The only requirement is that all members that are *referenced* by the other system layers are provided. By using this technique configurable and optional

layers were implemented in PURE.[2]

Another experience with PURE was that also in operating systems there are *crosscutting concerns* and that it makes sense to implement them as aspects. For example, based on the AspectC++ language [1, 18]we modularized the implementation of interrupt synchronization [15]. The main advantage was that the synchronization strategy could much easier be statically configured than in other systems.

However, it turned out that the step from a variable interrupt synchronization feature to a class hierarchy with aspects was not straightforward, because the functional hierarchy model does not provide any elements to represent crosscutting concerns.

2.3 Lessons Learned

In comparison to other configurable systems such as eCos [2] the PURE operating system family consists of modules that are much better to understand and maintain, because no code is needed within the modules to implement the static configuration. Not a single *#ifdef* pollutes the classes and, due to AOP, code that implements crosscutting concerns is well-separated. For instance, in eCos crosscutting concern implementations contribute about 20% of the whole kernel code [14].

However, on the modeling level we experienced two important problems that were related to functional hierarchies:

1. **no nested hierarchies:** highly configurable systems cannot be represented by a single deep functional hierarchy. The functions of FAMOS were rather course-grained in comparison to the functions of PURE. Therefore, nested class hierarchies would have been necessary to cope with the complexity.

2. **no crosscutting concerns:** on the modeling level functional hierarchies offer no adequate element to describe crosscutting concerns. That makes it very difficult to derive a class diagram with aspects in a systematic way.

These problems were the motivation for us to start thinking about an extension of the functional hierarchy model.

3. THE CONCERN HIERARCHY MODEL

In the functional hierarchy model functions are atomic entities. This makes the static configuration very easy. No feature-driven configuration techniques are necessary. However, for product lines that strive for a high degree of configurability this is not enough. Therefore, our extension does not only model purely functional concerns, but also its *sub-concerns* and *crosscutting concerns*. We call this more general and extended model "concern hierarchy". The following sections will describe the two extensions in detail.

3.1 Sub-Concern Modeling

Sub-concern modeling is needed to support step-wise refinement during the modeling process. A complex function is regarded as a program family within the program family. It is again modeled as a

[2] More details on feature-driven configuration can be found in [5] and [7]

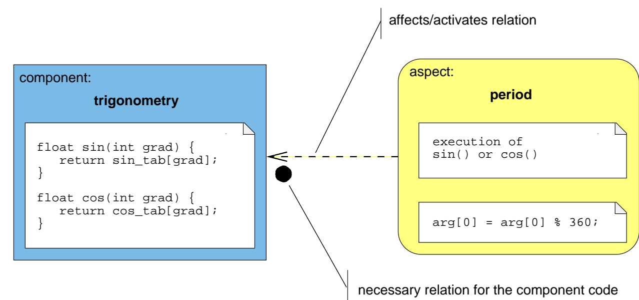

Figure 3: Component code depends on an aspect

concern hierarchy. For convenience the sub-concern hierarchy can either be visualized in-place or as a separate diagram.

As a consequence concern hierarchies don't have atomic entities. Every concern can always be refined. This process is continued until a granularity has been reached that matches the demands on configurability.

3.2 Crosscutting Concern Modeling

Crosscutting concern modeling is much more complicated than sub-concern modeling, because the relations between crosscutting concerns and non-crosscutting concerns as well as the relations among crosscutting concerns are still a field of active research. The following parts will describe the authors' point of view, which is based on experience with aspect-oriented product line development with AspectC++. Our approach is to analyze the relations between aspects and component code as well as the relations among aspects. This knowledge is then used to describe relations on the more abstract concern hierarchy level.

3.2.1 Relations Between Crosscutting Concerns and Ordinary Concerns

An aspect weaver can be regarded as a generic system monitor [10]. Whenever a certain condition becomes true, which is described by an aspect, some specific instructions (*advice code*) are executed. An explicit call is not necessary. There are two possible perspectives on this relationship. On the one hand the aspect code is **activated** by the component code. The activation happens implicitly by reaching a certain state. On the other hand the aspect code **affects** the component code, because it modifies the component code state after activation.

This bidirectional relationship between aspects and component code does not necessarily mean a dependency in the sense of the functional hierarchy. In many cases aspects can exist in a system even though their condition never becomes true and, thus, the aspect code is never activated. At the same time it is often no problem for component code to be unaffected by aspects. This becomes immediately clear if one considers an aspect for the detection of error conditions. For developers of software product lines this property of aspects is very important. It makes it possible to write loosely coupled aspects that work independently of the system configuration, where some or all target components might be missing.

Besides loose coupling there are also tight coupling scenarios. For example, some component code implementation might rely on an affecting aspect. This is illustrated in figure 3. It shows a component that implements trigonometric functions and an aspect that

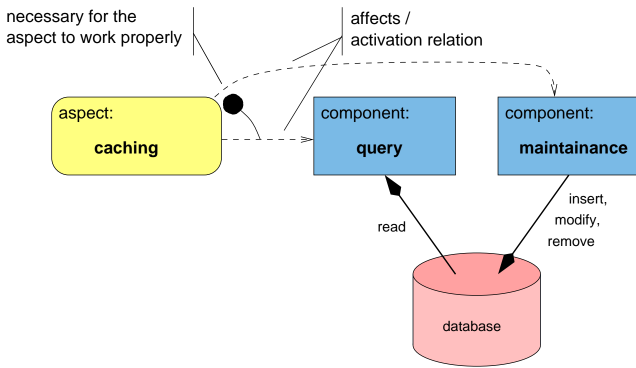

Figure 4: Aspect code depends on activation

makes sure that the argument for *sin* and *cos* is always in the range between 0° and 359°. Without this aspect the component would not conform to its specification.

The dashed line in the figure represents the relation between the component *trigonometry* and the aspect *period*. The filled black circle on the component side means that the component depends on this relation. It is a **necessary relation**.

It is also possible that an aspect depends on the activation in order to work properly. This illustrated in figure 4. The scenario is a database management system that consists of a *maintenance* and a *query* component. An aspect should improve the system's query performance by caching result values. This is implemented by advice for the *query* component. However, in order to guarantee the consistency of the cached results, an additional advice for the *maintenance* component is necessary. It is used to monitor all operations that insert, remove, or modify data.

In this case the mark is on the aspect side. The line that connects the mark with the advice for the *query* component means that the necessary activation would not be necessary without the relation to the *query* component.

As *buffering*, *query*, and *maintenance* can also be regarded as concerns in a concern hierarchy, we conclude that we can and should use the same kinds of relationships for crosscutting and ordinary concerns in concern hierarchies as well. We also use the same graphical notations. The main difference between figures 3 and 4 and the corresponding concern hierarchies is that a crosscutting concern can not always be implemented by an aspect. This depends on programming language features and the nature of the concern.

Another special property of the relationship between crosscutting concerns and ordinary concerns is that crosscutting concerns can affect groups of other concerns. We represent groups in graphical concern hierarchies by areas with a dashed borderline and a group name such as *group 1* and *group 2* in figure 5.

3.2.2 *Relations Among Crosscutting Concerns*

In languages like AspectJ or AspectC++ aspects have ordinary attributes and member functions. They can be regarded as an extension of the class concept. Therefore, an aspect can have the same relations to other aspects as to component code.

More interesting are indirect interactions among crosscutting con-

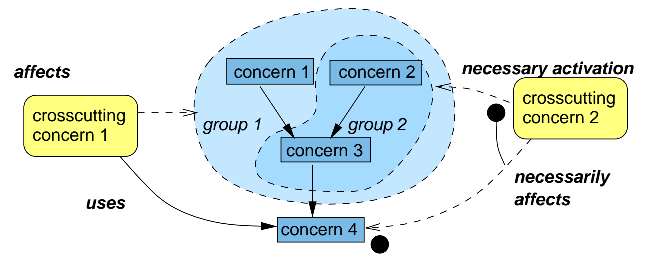

Figure 5: Concern hierarchy with concern groups

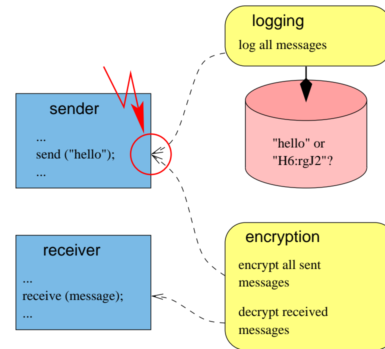

Figure 6: Interactions of crosscutting concerns

cerns. Figure 6 illustrates these interactions with an example. There are two communicating components *sender* and *receiver*. Two aspects affect the components. The first aspect is *logging*. It stores all transmitted messages in a log file. The second aspect is *encryption*. It encrypts all messages on the sender side and decrypts them on the receiver side. Although these two aspects don't have a direct dependency, the order in which the advice is activated is crucial in this scenario. If the *logging* aspect is activated first, the log file will contain unencrypted messages, otherwise encrypted messages. This difference might decide over the whole system's security. Therefore, AspectJ and AspectC++ provide special language elements to control the invocation order of advice code.

Our conclusion is that crosscutting concerns can have **order relations**, in some cases even **necessary order relations**. An example for a necessary order relation is an encryption concern that extends all messages, for instance, by a code that describes the encryption method. If the logging concern relies on this message format, it would not work properly without the encryption concern's advice being executed first. This means that a *necessary* order relation is required, because at least one of the aspects would otherwise not work according to its specification. In the case of a *non-necessary* order relation all aspects work properly, but there is a relevant difference in the system's behavior and, thus, we would like to apply an ordering mechanism.

Figure 7 shows the graphical notation for normal and necessary order relations in concern hierarchies. For order relations a dotted line is used. A necessary relation is again marked by a filled circle.

4. TOWARDS A DOMAIN DESIGN

The previous sections described the relations of crosscutting concerns with other ordinary and crosscutting concerns. These relations shall be used in concern hierarchies. Modeling them ex-

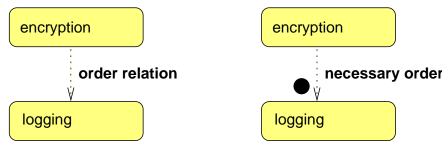

Figure 7: Normal and necessary oder relations

plicitely facilitates the derivation of a detailed design. As concern hierarchies are completely independent of the applied programming paradigm and, of course, also independent of the programming language, they can be used for any project and any design model. As an example, the following sections will discuss how an aspect-oriented class hierarchy can be derived from a concern hierarchy.

4.1 Ordinary Concern Modeling

As described earlier in section 2.2, ordinary concerns can be implemented by classes. The dependency relations between these concerns can be expressed by inheritance. If a concern is too complicated to be implemented by a single class, it has to be refined by sub-concern modeling on the concern hierarchy level. If a concern is not complex enough to be implemented by a class, it often makes sense to group a number of concerns from the same layer of the concern hierarchy into one class[3].

4.2 Crosscutting Concern Modeling

Crosscutting concerns are most naturally implemented by aspects. However, there are cases in which the aspect-oriented programming language is not powerful enough to express the crosscutting concern in a modular way and a scattered implementation is unavoidable. Nevertheless, even in this case the explicit separation of crosscutting concerns in concern hierarchies is beneficial, because the developers are now aware of the problem and can mark the scattered code fragments. This allows them to configure these concerns statically, for instance with text-based transformation tools, or to remove the code automatically as soon as better AOP support is available.

4.3 Sub-Concern Modeling

Concern hierarchy modeling can be applied recursively. Each concern can be described by another, more detailed, sub-concern hierarchy. The classes and aspects from the sub-concern hierarchy can either be grouped by using UML elements like "components" or they can simply be merged with the classes and aspects from the main concern hierarchy.

The motivation for the design of a sub-concern hierarchy is the need to implement a certain concern in a configurable manner. This typically means that a component that uses the resulting configurable components has to be developed against a common interface. A very simple approach to decouple client code from configurable service classes is a "class alias".

Class aliases are the best choice if a sub-concern hierarchy models a family that implements an alternative feature from a feature model and if the feature binding time is *compile time*. It statically connects clients with 1 of N classes.

[3] Note that grouping functions into one class might increase the memory footprint of the system in some configurations if the development tool chain does not support "function-level linking".

If the binding time is *runtime*, a *strategy* design pattern [11] has to be applied. Here the clients use an abstract *strategy* class to access the classes of the sub-concern hierarchy. The project-specific application code then has to connect the client classes with the right instance of a *concrete* strategy implementation. It is important to understand that the abstract and concrete strategy classes belong to the client code and not the sub-concern hierarchy's classes. In contrast to pure OO design, a class hierarchy that is derived from a concern hierarchy never starts with an abstract class. The goal is to avoid dynamic dispatching wherever possible.

4.4 Derivation of a Dependency Model and Tailoring

Concern hierarchies contain various kinds of concern relations that were not known in functional hierarchies. The dependency relations in functional hierarchies are very useful, because they help the developer to find a module structure that can be tailored by using only application-driven configuration mechanisms. Our next step is to discuss the new kinds of concern relations with respect to concern dependencies in order to systematically derive a dependency model for the product line components.

If a crosscutting concern affects an ordinary concern, this relation does not necessarily imply a dependency relation. For example, a *tracing* concern does not depend on the existence of any specific target component, nor do the system components depend on the tracing concern. This kind of relation can be completely ignored in a dependency graph. However, the situation is different with *necessary* relations, which are marked by a filled circle in our graphical notation. In this case the dependency is directed from the marked concern to the other concern. Bidirectional dependencies have to be avoided, because the dependency graph is cycle-free by definition [4].

For order relations the situation is quite similar. Normal order relations are merely implementation guidelines. They do not affect the system's configurability. Once again this is different for *necessary* order relations. The dependency is directed from the marked crosscutting concern to its counterpart.

As an example figure 8 illustrates the dependency model derivation. The relation of *crosscutting concern 1* and *group 1* can be ignored, because it is not a necessary relation. *Concern 2* and *3* have to be affected by *crosscutting concern 3*. Therefore, the dependency model contains an edge from *concern 3* to *crosscutting concern 3*. As *concern 2* also depends on *concern 3*, it is not necessary to explicitly mark its dependency of *crosscutting concern 3*. The dependency relation is transitive. *Crosscutting concern 2* has to affect *group 2*. Otherwise it would not work properly. This means that a dependency edge from *crosscutting concern 2* to *concern 2* is needed.

The result of this mapping is a graph that precisely describes possible system configurations if each concern is implemented by a separate module. As separation of concerns and the modular implementation of crosscutting concerns are the main goals of aspect-oriented programming, the model is an ideal design aid for developers of aspect-oriented software product lines.

[4] Habermann describes a technique called "sandwiching" to get rid of this problem [12].

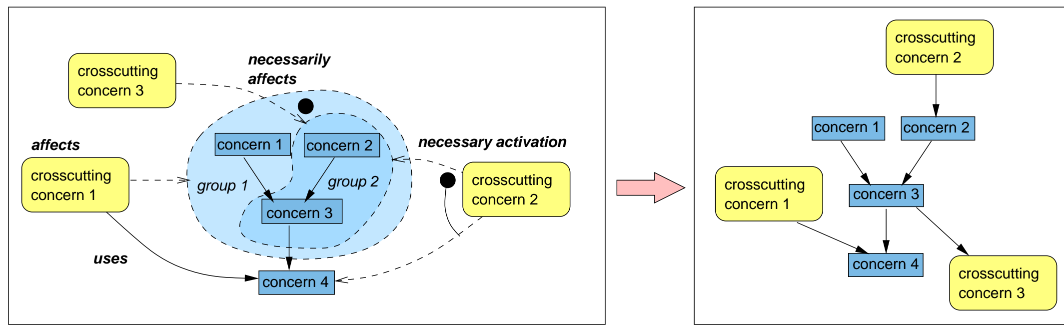

Figure 8: Derivation of the dependency model from a concern hierarchy

5. RELATED WORK

This work is related to all product line engineering methodologies such as FODA [13] or DEMRAL [9]. The unique feature of our approach is that we explicitly try to support developers of aspect-oriented product lines by modeling ordinary and crosscutting concerns during early design steps. In this sense it is similar to theme/UML [8], but the class and dependency model derivation are different. Furthermore, our approach is not based on UML, but extends the model of functional hierarchies. GenVoca architectures and Feature-Oriented Programming [4, 3] are another product line design approach that has its roots in Parnas' and Habermann's work in the seventies. In a GenVoca architecture systems also have a layered structure that are also intended to be implemented as "object-oriented virtual machines". However, crosscutting concerns in the sense of dynamic crosscutting (as supported by AspectJ and AspectC++) have not been considered by this approach, yet.

6. CONCLUSIONS AND FUTURE WORK

In our own ongoing product line development activities concern hierarchies have already been very useful. We use them after describing the variability of the domain with a feature model[9]. While feature models are typically used to describe a *problem space*, concern hierarchies complement feature models, because they are used to design a *solution space*. A concern hierarchy can therefore be regarded as the description of the relations of features in the context of a planned solution space.

Concern hierarchies are a pragmatic extension of functional hierarchies that was necessary to cope with a modern aspect-oriented implementation technology. By modeling crosscutting concerns very early we can systematically derive a detailed design model, e.g. an aspect-oriented class hierarchy, and a module dependency model. We feel that this is a unique and very promising approach, which we should share with other developers in this particular area.

The model description in this paper was informal and based on examples. Our intention was to address a broad audience.

Concerning future work, we plan to analyze and incorporate the feedback on this paper. Our goal is to develop a more precise and formal description of the model and the related development process.

7. REFERENCES

[1] AspectC++ homepage. http://www.aspectc.org/.

[2] eCos homepage. http://ecos.sourceware.org/.

[3] D. Batory. Feature-oriented programming and the AHEAD tool suite. In *26th Int. Conf. on Software Engineering (ICSE '04)*, pages 702–703. IEEE Computer Society, 2004.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *25th Int. Conf. on Software Engineering (ICSE '03)*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[5] D. Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2003. http://www.pure-systems.com/.

[6] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *2nd IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '99)*, pages 45–53, St Malo, France, May 1999.

[7] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Sci. Comput. Program.*, 53(3):333–352, 2004.

[8] S. Clarke and R. J. Walker. Generic aspect-oriented design with Theme/UML. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 425–458. Addison-Wesley, Boston, 2005.

[9] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* AW, May 2000.

[10] R. Douence, P. Fradet, and M. Südholt. Detection and resolution of aspect interactions. Technical Report No. 4435, Institut National de Recherche en Informatique et en Automatique (INRIA), Rennes, France, Apr. 2002.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. AW, 1995.

[12] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *CACM*, 19(5):266–272, 1976.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, Nov. 1990.

[14] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *EuroSys 2006 Conference (EuroSys '06)*, pages 191–204. ACM, Apr. 2006.

[15] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schröder-Preikschat. An aspect-orientied implementation of interrupt synchronization in the PURE operating system family. In *5th ECOOP W'shop on Object Orientation and Operating Systems*, pages 49–54, Malaga, Spain, June 2002.

[16] D. L. Parnas. Some hypothesis about the uses hierarchy for operating systems. Technical report, TH Darmstadt, Fachbereich Informatik, 1976.

[17] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Design rationale of the PURE object-oriented embedded operating system. In *Proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES '98)*, pages 231–240, Paderborn, Germany, Oct. 1998.

[18] O. Spinczyk and D. Lohmann. The design and implementation of AspectC++. In *Journal on Knowledge-Based Systems, Special Issue on Creative Software Design*. Elsevier, 2006. (to appear).