Are Aspects useful for Managing Variability in Software Product Lines? A Case Study

Alexander Nyßen[1] any@cs.rwth-aachen.de
Shmuel Tyszberowicz[2] tyshbe@tau.ac.il
Thomas Weiler[1] thweiler@cs.rwth-aachen.de

[1] RWTH Aachen University, Software Construction Group, Ahornstr. 55, 52074 Aachen, Germany
[2] School of Computer Science, The Academic College of Tel-Aviv Yaffo and Tel-Aviv University, Israel

1 Introduction

Software product line (SPL) based development can help to significantly reduce time-to-market as well as development costs [Don00], by increasing the reuse of all types of documents. To achieve this, SPL development is composed of two primary activities, namely *domain engineering* and *application engineering*, that are interlaced during development. Within domain engineering the common and variable parts of the products, which belong to an application domain, are analyzed and described. The resulting documents of this process form the basis of the product line, the so-called Product Line Platform (PLP). During application engineering concrete products are then derived from this PLP. Therefore the terms *application* and *product* will be used interchangeably. By careful development and intensive tests of the common parts of the SPL, a correctly applied product line based approach thus can increase the quality of the end products .

One main task, which has to be solved during SPL development, is how to represent variabilities adherent to a product line. On the one hand, variability determined during analysis has to be mapped into architectural design; on the other hand, mechanisms are needed to implement these variabilities in source code. Since SPLs consist of a large set of products derived from the PLP, *traceability* of design decisions and variability is crucial for the maintainability and evolution of the SPL. Otherwise return of investment can not be achieved. Thus mechanisms are needed to track concerns identified during the analysis phase throughout the architecture and even the implementation of the SPL.

This paper presents a case study that examines if and how aspects can help to improve *separation of concerns* in the context of SPLs, thereby supporting the handling of the adherent variabilities. As a case study we selected electronic shopping systems (e-shops). During the development of our product line we focused only on domain variability between products, i.e. variability which arises during *domain analysis* and is then resolved during *application analysis*. Variabilities which arise, or are resolved, during later stages of the development were out of scope of this case study.

The paper is organized in 5 sections: In the following section we discuss related work in modeling SPLs. In Section 3 we describe our approach for modeling an e-shop product line. After that, in Section 4, we examine if and how aspects can support separation of concerns at the early development stages, namely analysis and design. This is necessary in order to achieve the aforementioned traceability and to get an SPL which is easier to understand and maintain. Furthermore, we show how aspects can be used to implement variabilities present in our E-shop Product Line. There, we also discuss alternative approaches to implement variability and survey the benefits and drawbacks of the different solutions. In Section 5 we summarize our approach.

2 Separation of Concerns in Software Product Lines

As we already pointed out, a successful product line approach can only be achieved if traceabilty of all design decisions and variabilities is ensured. When using feature modeling in the analysis phase and an object-oriented approach in the design phase — as we do — it is of major importance to trace how the functionality described by a feature is realized by one or possibly multiple classes. The ability to trace such things is especially necessary when dealing with the variabilities defined in cohesion with those features. If, for example, an optional feature is selected to be contained in a product, the implications on the product specific architectural models have to be concise.

To achieve such a traceability, all concerns that are of relevance during the different stages of software lifecycle (e.g. features during the analysis phase, classes during design phase) have to be clearly delimitable and all dependencies and interferences between those concerns have to be tracked. This can only be achieved in combination with a clear separation of those concerns.

The principle of *separation of concerns* has — right from the early days of software engineering — been a major design principle applied to software decomposition, see [Par72]. What it generally refers to is, like stated in [OT00], *the ability to identify, encapsulate, and manipulate only those parts of a software that are relevant to a particular concept, goal, or purpose*. The benefits one attributes to a clean separation of concerns during software decomposition are overwhelming. One can exemplarily list [OT00] reduced complexity and improved comprehensibility, traceability within and across artifacts and throughout the software lifecycle, limitation of the impact of change, facilitation of evolution and non-invasive adoption and customization, facilitation of reuse, and simplification of component integration.

If one generalizes the separation of concerns principle from a mere design principle into a more general principle that is applied during all phases of the software lifecycle, identification and tracing of concerns throughout the different phases becomes more promising. In practice, however, one often faces the problem that the many different kinds of concerns, like use-cases, features, roles, or classes, which are of interest in the different phases and to the different stakeholders, are often interwoven and overlapping.

This is where Ossher and Tarr engage with their approach of *multi-dimensional separation of concerns*. The idea behind the approach is to classify concerns into kinds of concerns (e.g. classes, features, roles, etc.), so called concern dimensions, and to support

- a clear separation of concerns in each concern dimension, and
- a precise definition of the relationships between concerns (of also different dimensions).

Although we do not want to investigate Ossher and Tarr's technical solution (*HyperJ*) further, we want to point out one central problem they address with their approach, namely *the tyranny of the dominant decomposition*. What it refers to is that by the selection of a certain modularization criteria during a certain phase of the development cycle, the concern dimension enmeshed with the chosen modularization criteria becomes the dominating one. In an object-oriented design for example, the *classes concern dimension* becomes dominant. This in turn leads to the problem that concerns of other dimensions (e.g. features, which are modeled during the analysis phase) can not be adequately encapsulated in all cases (features, for example, may be spread over several classes) so that a clean separation of concerns can not always be achieved.

In detail, when switching the modularization criterion from one phase to another (or by changing the perspective between different stakeholders), three phenomena caused by the *tyranny of the dominant decomposition* problem can be observed:

- *tangling*, which refers to the fact that multiple concerns from one dimension are realized by one concern of another dimension, e.g. multiple features of the analysis model may be implemented by a single class in the design phase.
- *scattering*, which means that a concern of one dimension is realized by multiple concerns of another dimension, e.g. a feature of the analysis model may be implemented by multiple classes, realizing disjunctive parts of the feature.
- *cross-cutting*, which means that a concern of one dimension gets realized redundantly by concerns of another dimension, e.g. a feature of the analysis model may be implemented redundantly by multiple classes.

Tangling and scattering seem to be an indicator for a bad decomposition and can be avoided in most cases by applying a different modularization. In contrast, cross-cutting may be caused by the fact that with respect to the chosen modularization criterion, a concern gets *cross-cut* from one phase to another. This is not avoidable in all cases and makes tracing of those concerns difficult.

As aspects can be applied to handle such cross-cut concerns [Lad03], we commenced the herein documented field study to investigate how aspects could help us gain a better separation of concerns and a better traceability, and thus reduce the problems described. In detail, two major questions were of concern:

1. Can aspects be applied to reach a better separation of concerns and a better traceability between the features identified in the analysis phase and the classes of the architectural design model? Or, to be more concrete: does it help to implement variable features, which would become cross-cut when going from feature-based analysis to the object-oriented design as aspects?
2. Can aspects also help to technically deal with the variability that can be modeled on features in the analysis phase?

Both questions will be dealt with in Section 4 after our product line engineering approach is introduced.

3 Designing a Product Line Architecture - The E-Shop Case Study

In this section we describe our approach for modeling a SPL based on a *feature model* [K+90]. As a case study we selected electronic shopping systems (e-shops) as the underlying domain. We used *Feature Modeling* to model domain variability of the e-shop product line because it is a popular and suitable methodology for modeling commonalities and variabilities of software product lines. One problem which arises during feature modeling is how to identify a feature: Since feature modeling is used in different contexts with varying intentions, there is no consensus on what a feature is and is not [LMNW03].

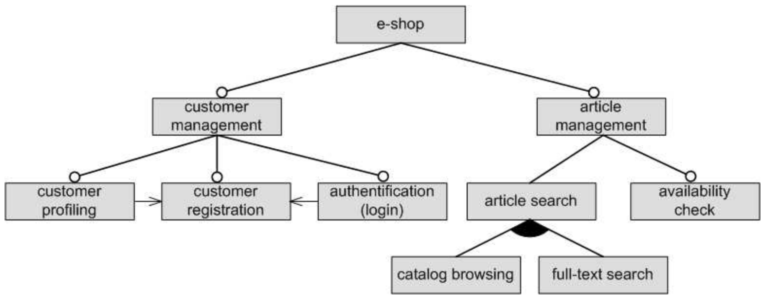

Figure 1: Part of the Feature Model

Our case study showed that it is suitable to focus mainly on functional decomposition of the domain when modeling domain variability, because a user primarily differentiates products of a domain by the offered functionality. To illustrate this, Figure 1 shows part of the feature model of our e-shop product line.

3.1 From Features to Architectural Components

During our case study we tried to elaborate a methodology to design a product line architecture based on a feature model created during the analysis phase. A *good* software architecture not only has to fulfill the functional requirements of a system, but also has to assure that the earlier defined quality attributes are met. *Architectural styles* present one method of achieving this. They improve certain quality attributes but on the other hand are less supportive for other quality attributes, see [Bos00]. The *layered architecture style*, for example, increases flexibility of the system by defining different levels of abstraction, but generally decreases performance of the resulting system. The architectural style chosen defines a categorization of the *building blocks* of the system's architecture.

In our approach we use feature models as the primary input for building an architecture of the SPL. The main advantage of this procedure is that the feature model defines already a *net of terms*, which are identified to be crucial for the system. These terms - the *features* - are thereby organized in a structure (here: a tree or graph) abstracting from the *nebulous cloud* of customers requirements. Thus, feature modeling supplements the output of the requirements process (namely the requirements specification), by providing a more formal and structured input for the following design phase. This aids the software architect finding the structure for the system and identifying architectural components.

When analyzing the feature model to identify architectural component candidates, the categorization defined by the architectural style chosen has to be taken into account. This way all features modeled in the feature model are sieved by some kind of filter stack, defined by the chosen architectural style, see Figure 2. In our case study we used *layers* as the architectural style of our system, consisting of the three layers *presentation*, *business logic* and *data*. The categorization respectively *modularization* defined by the chosen architectural style will rarely map one to one to the features modeled in the feature model. Therefore each feature has to be analyzed to that effect which *parts* of its specification map into which category defined by the architectural style. We call these parts *feature components* (FC).

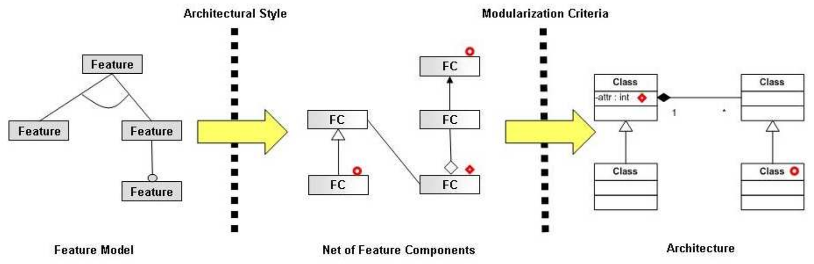

Figure 2: From features to architectural components

We are convinced that features can not be directly mapped to architectural components like classes, aspects, or other technical components because a feature model and thus its features are an abstraction of the requirements of an SPL. Because an architectural design has to be concrete on its components, one has to analyze the features on how they contribute to architectural components. Hence this transformation, in our opinion, can not be automated, but supported by a set of *transformation rules* which help the designer to identify architectural components.

First, all features have been analyzed with respect to the chosen architectural style to find how they contribute to the different categories defined by the architectural style. This way, lists of feature components have been created for the different layers of the architecture. In the next step the relationships between the earlier identified feature components have been analyzed, based on the feature descriptions and a set of transformation rules. The result of this process was a *net of feature components* which served as an input for the next step.

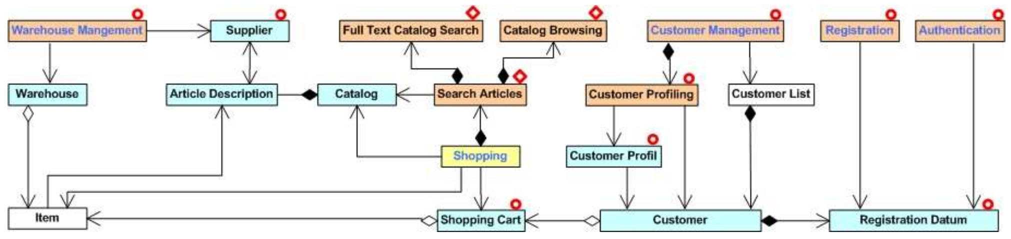

Figure 3: Part of the feature components net of the e-shop product line

A fraction of the resulting net of feature components is shown in Figure 3. The diagram only shows feature components of the business logic layer (shown in ocher) and data layer (shown in blue), because feature modeling was first concentrated on modeling the functionality offered by the products of the given domain and the data processed by it. The presentation layer consists of feature components which serve as an interface between the user and the system and therefore allow to trigger functionality provided by the business logic layer and display data of the data layer. Another step was therefore needed to model these feature components.

The analysis of the feature components was not straightforward, and several iteration steps were needed to obtain the final net of components. In Figure 3, feature components which were not on the list of feature components mentioned before, but were identified during the creation of the net of feature components, are shown in white. The annotations in red depict variability, which is discussed in more detail in Section 3.2.

In the next step we had to form architectural components from the feature components identified before, according to the modularization criteria used at the subsystem level of the architecture. In our case study we used the object-oriented approach for modeling the details of each of the aforementioned three layers. In order to form classes, each component feature of the several layers had to be analyzed to determine whether it denotes a class, a responsibility (method) of a class, or an attribute of a class.

This way we formed a class diagram representing the static structure of our e-shop. The diagram was again revised by adding (technical) classes which did not directly correspond to a feature component, respectively feature. Furthermore, this iteration step revealed some deficiencies and was used to improve the initial feature model as well.

3.2 Variability and Dependencies

One part which has not yet been discussed, is how to model and trace variability modeled in the feature model into the architecture. In our approach, each variability modeled in the feature model is assigned a numerical ID. Variants involved in an alternative e.g. are numerated using the ID of the alternative itself (e.g. 3) and a serial number for the variant (e.g. 3.2).

During identification of the feature components each variability, which is assigned to a feature in the feature model, is analyzed to determine whether it influences one of the feature components derived from the feature. If so, the feature component concerned is also assigned the variability, which is depicted, in Figure 3, by a red circle (denoting an option) or red diamond (denoting an alternative). Therefore, a variability modeled in the feature model can be distributed over a set of feature components which all have the same variability with the same ID assignment. This way, traceability of the domain variability modeled in the feature model is ensured. Dependencies between variabilities in the feature model are handled the same way: A dependency between variability *1* and *2* in the feature model is mapped directly to dependencies between the corresponding variabilities assigned to feature components.

4 Managing Concerns and Variability with Aspects

During our case study we tried to determine if and how aspects could help to achieve *separation of concerns*, thus easing representation, realization and maintenance of domain variability. As pointed out in Section 2, the problem which arises during modularization of a system is that some concerns may become crosscut with respect to the chosen modularization criterion (indeed with respect to the concrete modularization). To solve this problem, one can use aspects to handle these concerns separately from each other.

Within aspect oriented development the use of aspects at the *early stages* is often discussed [RSMA02]. Thereby one use of aspects is to deal with tangling and scattering concerns, meaning functionality which is merged into a single architectural component or distributed over a set of those respectively. However our case study revealed that whenever tangling or scattering concerns were identified, it was caused by bad modularization. That is, those concerns could in all cases be well separated by refactoring the class-based modularization of the system. Nevertheless, cross-cutting concerns can emerge because in general the modularization criterion is switched from analysis to design. Our case study has shown, however, that these crosscutting concerns do not need to be realized as aspects, as we will state in the following.

As not all of the features provided by the e-shop product line are mandatory, many variable features will have to be regarded. As we pointed out in Section 2, dealing with those variable features may be difficult, as phenomena like *tangling* and *scattering* may occur when switching from feature-based analysis models to object-oriented architectural models. This is what makes it difficult to handle the adherent variabilities.

That is why we investigated if and how modeling and implementation of variable cross-cutting features by means of aspect oriented programming could help to decrease the drawbacks experienced. The way aspect-oriented programming tries to solve cross-cutting problems is, simply put, by applying the separation of concerns principle. By implementing cross-cuttings concerns as aspects, a neat encapsulation can be achieved, which enables easy identification and handling of those concerns.

We decided on the customer registration feature to demonstrate and evaluate how a variable feature could be implemented using aspect oriented programming. Customer registration is an optional feature of our e-shop product line, meaning that it can be freely selected to be part of a concrete e-shop product (at least as long as a general customer management capability is included, as can be inferred from Figure 1). It is mainly used to retrieve and persistently store customer details like name, address, credit card and bank account details, which would otherwise have to be provided by the customer in each session. If selected, it may be required in different situations, e.g., when an unregistered customer has to authorize in order to enter an e -shop only accessible to registered customers (i.e., the e-shop has the authentication feature), or when a customer wants to place an order after browsing the article catalog of a freely accessible e-shop (i.e., one without having the authentication feature).

Using for example AspectJ, the registration feature can be implemented as an aspect which contains the functionality of the feature as well as the information regarding where to bind it:

```
public aspect Registration {

   private void register(){
      // functionality realizing registration process here
   }

   pointcut orderPlacement() : call (void OrderSystem.placeOrder(..));

   before() : orderPlacement() {
      register();
   }

   ...
}
```

As we assume that the customer should have the possibility to register when placing an order (in a freely accessible e-shop), a join point `call(* void OrderSystem.placeOrder (..)  )` has to be defined, to specify where the aspect has to be binded. It has to be pointed out that because we are dealing only with domain variability, binding of this feature is always static, i.e. there is no need to change the behavior of the system at run-time.

The simple feature implementation demonstrates one benefit of using aspects, namely the ability to weave the code realizing a variable feature into the source code of the PLP without explicitly changing it or having to be aware that the feature's functionality would be added. This may also be seen as an advantage when new unforeseen features have to be added during evolution of the product line. The aspect solution, however, may be considered bad with respect to software engineering issues, as the aspect contains the whole logic of the feature implementation, as flat global method space is used [MO04]. Another problem is that traceability and readability is not provided, as the interaction of aspects with the classes of the platform is no longer explicit (pointcuts are defined inside the aspects and are not visible from the join points).

As an alternative to aspects one may consider the use of classes, as in a regular object-oriented development. The major difference compared to the aspect solution is in the explicitness of associations: in a class diagram the associations are explicit, while adding aspects makes them implicit. Furthermore, classes use explicit invocation (objects react to messages sent by other objects) while aspects react to *events* generated by objects (implicit invocation). Therefore, a classical object-oriented approach would overcome the problems of traceability and readability, as all communication would be clearly identifiable via method calls. On the other hand, problems of the classical object-oriented solution become immanent when regarding the variabilities adherent to features. Selecting an optional feature as part of a concrete product can cause a lot of code modifications, as all associations and invocations to the feature's class are affected. In our opinion this is only a technical problem which can be solved by tools, which can automatically include/exclude the code concerned as is done, for example, by compilers when using preprocessor instructions. With aspects this inclusion/exclusion is done by the *weaving*.

We conclude that regarding architectural design a classical object-oriented solution would be preferable, as knowledge about association and invocation relationships between the different concepts (here: classes) is absolutely necessary to reach traceability and readability. Regarding the technical implementation, implementing conceptual (i.e. architectural) classes that realize variable features in form of aspects could be reasonable means, as the technology seems to be mature and much experience has been gained in this field. Other tool-supported solutions, however, could be more promising, if they also manage to conserve the traceability from analysis and design towards implementation.

Our case study furthermore has shown that, besides being technically applicable to implementation, aspects are not needed as *modeling elements* in requirements engineering or design. During requirements engineering, no crosscutting concerns arise because one has to deal with only one modularization criterion (here: features). This is why aspects are not needed here. As aspects as well as classes are congenerous technical realizations for architectural components, we also had no need for *early aspects* during design.

5 Summary

In this paper we presented a case study which has been used to elaborate if and how aspects can support separation of concerns, thus easing realization of domain variability present in an SPL. Based on our approach to model an SPL architecture based on a feature model we have shown that, while aspects provide a suitable realization of variability, the aspect based realization of variability complicates the readability and comprehension of an architecture. This is because of the *implicit* communication link between aspects and classes. Thus we prefer realization of variability by classes, which has to be supported by adequate tool support. In this manner, the implicit communication within the aspect based approach becomes explicit, thus improving comprehension of the overall system and further enhancing traceability of the modeled variability.

References

[Bos00] Jan Bosch. *Design & Use of Software Architectures*. Pearson Education Limited, 2000.

[Don00] Patrick Donohoe, editor. *Software Product Lines: Experience and Research Directions*. Kluwer International Series, 2000.

[K$^+$90] K.-C Kang et al. Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical report, CMU/SEI-90-TR-21, Pittsburgh, 1990.

[Lad03] Ramanivas Laddad. *AspectJ in Action*. Manning Publications Co., Grennwich, Conn., 2003.

[LMNW03] Horst Lichter, Thomas von der Maßen, Alexander Nyßen, and Thomas Weiler. Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung (only in german). Technical Report AIB-07-2003, RWTH Aachen, July 2003.

[MO04] Mira Mezini and Klaus Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *ACM Conference on Foundations of Software Engineering (FSE-12)*, 2004.

[OT00] Harold Ossher and Peri Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In *Proceedings Architectures and Component Technology: The State-of-the-Art in Software Development*, 2000.

[Par72] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communcations of the ACM*, 15(12), 1972.

[RSMA02] Awais Rashid, Peter Sawyer, Ana M. D. Moreira, and Jo Araujo. Early aspects: A model for aspect-oriented requirements engineering. In *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 199–202, Washington, DC, USA, 2002. IEEE Computer Society.