Fraunhofer Institut
Experimentelles
Software Engineering

Proceedings of

1. Deutscher Software-Produktlinien Workshop (DSPL-1), Kaiserslautern, November 2000

Editors:
Peter Knauber
Klaus Pohl

Der Workshop wird gefördert im Rahmen des internationalen ESAPS-Projektes, das als Projekt 99005 im EUREKA ∑! 2023 Programm innerhalb der ITEA-Initiative durchgeführt wird.

IESE-Report No. 076.00/E
Version 1.0
November 10, 2000

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competetive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach
Sauerwiesen 6
D-67661 Kaiserslautern

Table of Contents

Frühe Phasen: Scoping und Requirements Engineering

Requirements and Feature Management for Software Product Lines

Andreas Hein [1], **John MacGregor** [2], and **Michael Schlick** [3]

Abstract. As software is used to perform more and more complex tasks and must typically be provided in several variants for different customers, it is commonly accepted that software solutions should be based on solid requirements management. This consensus already holds for producing single applications, but even more so for realizing products in a software product line. A software product-line is a collection of products sharing a common set of features that address the specific needs of a given business area [ESI, 1998]. The product line approach brings more complexity through the abstraction of the commonalities and variabilities of the product line members on the domain level. So as well as a process that includes the domain, adequate tool support is essential to handle the overall methodology in an industrial setting. This paper reports on our experience in managing domain analysis work-products with current tools. In particular, it shows where existing tool support can be used as-is and where adaptations and extensions are needed.

The results presented here are taken from [PRAISE]. The PRAISE project, partly funded by the European Commission under ESPRIT contract 28651 and pursued by Thomson-CSF (France), Robert Bosch GmbH (Germany), and the European Software Institute (Spain), has investigated product-line realization and its assessment in industrial settings. A part of the project was dedicated to the validation and consolidation of proposed product-line technologies in real-scale industrial experiments.

1 INTRODUCTION

Developing a product line requires extra effort for creating domain models and managing them. In particular, domain requirements and features must be managed in addition to the requirements of the single products. Of course this extra effort should finally pay off. A large part of the expected benefit is from using the domain models as starting points for product derivation. That is, the analysis models can be configured and tailored on the domain level to fit the specific needs of a customer at the product level. Therefore requirements modeling for product lines must go beyond traditional requirements modeling for individual products. This implies not only that requirements and feature modeling for software product lines necessitates new concepts, but also extended tool support that realizes these new concepts.

This paper presents an extract of the respective PRAISE results in which Bosch focused on the Car Periphery Supervision (CPS) domain. Bosch's task, which was carried out by Corporate Research in their facilities, was, among other things, to experimentally apply commonality and variability modeling in order to assess its feasibility.

QSS DOORS [i] was used for managing the particular documents for the individual products on the one hand and the product line on the other hand.

The following provides an overview of the work-products that must be managed in the analysis phase, as well as a procedure to abstract and represent commonalities and variabilities on the domain level. After that it will be shown how existing tools, particularly QSS DOORS, can be customized to integrate the abstractions introduced and where there are shortcomings for which there are no practicable workarounds. Then, related work is presented. Finally the major points are summarized and a short outlook is given.

2 DOMAIN ANALYSIS MODELS

During domain analysis, information of the product models is generalized on the domain level so that developers can see what is common to all products, where variations occur and which decisions have to be made to create a special product. As multiple products and the domain level are involved, traceability is of major concern. Traces must be established from the product level to the domain level and between the different models of each level. Furthermore, relationships between the elements within one model may have special semantics, such as 'consists of', 'excludes', etc. These model-specific relationships must also be adequately represented.

Figure 1 shows which models we think are mandatory or at least useful for domain analysis, i. e. which models have to be managed by ideal requirements management or domain analysis tools. The domain analysis models are depicted to the left, the domain design with the architectural model to the right. For both there is a common domain vocabulary.

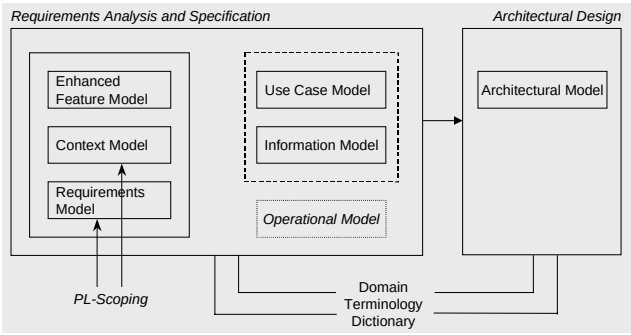


Figure 1. Domain Modeling Overview

In PRAISE we especially considered the domain analysis models that we think are an absolute must: the requirements model, the

[1] Robert Bosch GmbH, Corporate Research and Development – FV/SLD, P.O. Box 90 01 69, D-60441 Frankfurt am Main – email: andreas.hein1@de.bosch.com

[2] Robert Bosch GmbH, Corporate Research and Development – FV/SLD, P.O. Box 90 01 69, D-60441 Frankfurt am Main – email: john.macgregor@de.bosch.com

[3] Robert Bosch GmbH, K8-RU/EPA2, P.O. Box 16 61, D-71226 Leonberg – email: michael.schlick@de.bosch.com

context model, and the feature model. This paper concentrates on the requirements and feature models.

Product line scoping takes place before the domain analysis process. In scoping the requirements on the products that are to be incorporated in the product line are gathered and roughly sorted in advance. The results from scoping are input to the context model, where the domain of concern is separated from other domains and where the interfaces to the other domains are determined. Moreover, the overall result of scoping can also be regarded as a first version of the requirements model which would then be developed iteratively during domain analysis. Finally, the feature model addresses particularly the differences between the diverse products. So, roughly spoken, the feature model appears as a decision tree that represents the possible variations between domain products.

Depending on the field of application, it may be practical to illustrate the requirements through use cases or to complete them by other more detailed models. But this will not be discussed here. Nevertheless, it is important to notice that traceability must be guaranteed between the individual domain analysis models as well as the traceability to the product line architecture.

One goal of domain analysis is to build an abstract model that can be used as a starting point for the derivation of specific products. The feature model is predestined to play this role, as it captures the decisions that must be made to determine the individual characteristics of a system. Application derivation begins by eliminating the domain variability step by step and instantiating a subset of all potential features. Hence we get a complete specification of one single product variant.

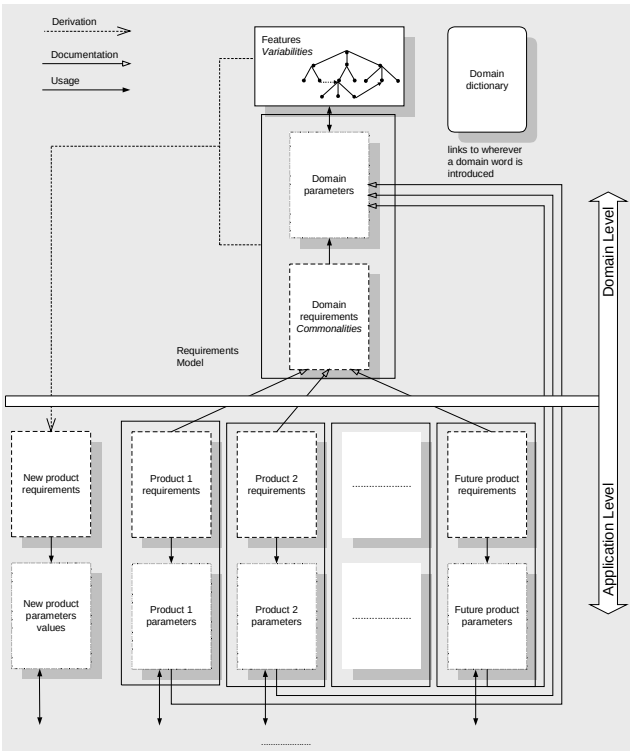


Figure 2. Building Reusable Domain Assets

The overall construction process of the requirements and the feature model, depicted in Figure 2, starts with requirements modeling for every single product used to define the domain. Product requirements may also include variability when they specify multiple variants addressed through parameters. Each requirement that contains parameters can be seen as a requirement template. These templates can be further refined by specifying parameter types and values. The combined requirements texts and parameter definitions of the product or the domain form a unit, the requirements model. The parameter definitions section essentially binds requirements that state commonality and features that state variability.

In the next step, all product requirements are abstracted to build the requirements of the domain. At this point a common domain language emerges. Subsequently, a domain dictionary defining the vocabulary used by experts and needed to understand the domain, related products and domains is created. It is then maintained during all phases of the product-line development process. To be able to understand where the abstractions come from, documentation links are established between corresponding elements on the application and domain levels.

Domain feature modeling usually begins after some requirements modeling. The requirements text structure and parameters are used to get an initial feature model that is then refined. The results from a feature modeling phase are brought back to the requirements model.

Features are modeled as nodes of a tree where basically each node corresponds to a parameter. So a feature tree describes the different possible values that can be assigned to each particular parameter.

Besides the basic tree structure we must introduce additional links between the nodes of the feature model. These links are necessary to express additional constraints between the different features of the tree. The particular constraints must be documented so that their existence can be understood, e. g., why one feature requires or excludes another.

Traceability links are established from each domain requirement to the parameters it contains, and from the parameters to the corresponding nodes in the feature tree. During the derivation process these links in turn are used to generate product specifications by navigating from the nodes of the feature model via the parameters to the requirements.

The domain analysis results (i. e. the models) must be validated through application requirements derivation. It should be possible to satisfactorily specify requirements of the products that have been used as the basis for the domain models. It is unlikely that the derivation will produce exact copies of the product requirements models. Depending on the techniques employed, the requirements text of a derived product is more or less similar to the text in the domain. The associated parameters document contains selected values for all parameters that have been covered during derivation. In order to completely derive an application, all variability must be resolved to final values. Nevertheless, variable products can be derived through partial derivation which leaves some selections open.

3 MANAGING DOMAIN REQUIREMENTS AND FEATURES

Having defined the concepts that are essential for representing and tracing domain requirements and features, adequate tool support for realizing these concepts is needed. It is clear that the concepts that have been introduced are not only new, but also very far-reaching. We knew that no tool or combination of tools would offer a complete solution. Consequently, we were restricted to tools that allow the emulation of at least some essential ideas through workarounds. We chose QSS DOORS as the most promising tool in this respect, especially as it supports

link attributes which we could use to define the link types and traces needed.
In the following, it is shown, on the example of QSS DOORS, where existing tool support has been useful, and where there are basic gaps that cannot simply be overcome by adapting the respective tools, with scripts, for example.
For requirements management in the context of the product line approach, we need support for

- word processing;
- text part identification;
- grouping of text parts;
- linking of text parts and groups;
- flexible filtering and view definition;
- easy link handling and restructuring;
- requirements traceability;
- comfortable navigation facilities;
- automatic generation of specific requirements texts.

From our experience, word processing is a prerequisite for any requirements management tool. It has been shown in the PRAISE experiment that a textual representation is more suitable for discussions with domain experts than more formal models where the information is dispersed. The requirement for word processing is fulfilled by QSS DOORS which provides an own MS Word-like editor for this purpose.
The capability to identify text parts without breaking the requirements document into small objects emerges in domain level modeling and is not supported by either of the tools known to the authors. Text part identification is needed for defining domain words and domain parameters within the requirements. Furthermore, it is useful to be able to group the defined text parts, as e. g. a domain word is likely to occur in the text several times. All occurrences could then be assembled to a group with a unique identifier and be traced through the group to the corresponding definition in the domain dictionary. This already implies that linking individual text parts as well as linking groups must be possible.
Flexible filtering and view definition is even more important in a product line development than in the development of a single product, as more product models and the domain level are involved. Special requests, such as filtering all product requirements that are not connected to the domain, must be supported as well as basic search-and-replace actions. In particular, filters must include link types, as link types are a main criterion for defining views on the complex network of models. QSS DOORS provides very good standard filters for objects (e. g. requirements or features). Nevertheless, standard filters for links are missing, so that it was necessary to define them through scripts.
Furthermore, easy link manipulation and restructuring in general are very important for product line engineering. Principally, the domain and the product level evolve iteratively. E. g., requirements that are first modeled as specific to a single product may later turn out to be commonalities within the product line, so that they must be moved to the domain level. Therefore restructuring the models should be especially supported. QSS DOORS enables restructuring within one model, but objects cannot be moved from one model to another, so that the abstraction process described here is not supported.
QSS DOORS enables requirements traceability within and between all models it manages, and to architecture and design models represented in Rational Rose. The respective functionality is provided through the DOORS Rose Link. This plug-in generates proxies either in DOORS or in Rose to allow the interconnection of the respective models. Moreover, context-sensitive menus let the user navigate along all links that are established.
It is obvious that no existing requirements management tool supports automatic generation of specific requirement texts, as there is no need for such a feature when developing single products without considering variants. Nevertheless, it is regarded very useful in the product line context, where a lot of different products and product variants are to be derived from abstract domain descriptions.
As stated above, introducing the product line concept to software engineering is very far-reaching. One must carefully decide on the extent up to which it makes sense to employ existing tools and adapt them, as it is not practicable to enforce a totally different meta-process through numerous workarounds. The overview is soon lost when workarounds are applied to real-world tasks.
This was the problem that we faced when it came to feature modeling. On the one hand, there is no commercial tool available for this task. On the other hand, dedicated tool support is indispensable for maintaining an overview of a multitude of different feature tree branches that may be arbitrarily deep and connected in complex ways. One idea behind using the requirements management tool QSS DOORS for feature modeling was that we could use its tree presentation capability. Besides, traceability between the requirements and the features was guaranteed as they were represented within one tool, and traceability to the architectural model represented in Rational Rose was guaranteed via QSS DOORS Rose Link. Nevertheless, it must be pointed out that feature modeling introduces a completely new meta-level. Moreover it became clear that a lot of issues concerning the configuration of feature models are far from realization. Here, a solid theoretical foundation is needed. As a first step we determined the expressiveness and the capabilities an ideal feature modeling tool should provide [Schlick and Hein, 2000]. The resulting meta-model includes

- graph structuring;
- object and link types;
- generic objects;
- abstract / concrete objects;
- reasoning and strategies.

FODA [Kang et al., 1990] proposes modeling features in a tree structure, with the composition rules defining the semantics existing between features that are not expressed in the feature diagram. Integrating the composition rules into the feature diagram produces a feature graph rather than a feature tree, however.
Object and link types are needed, e. g., to represent mandatory, optional, and alternative features and requires and excludes relationships that may exist between certain variants.
Support for generic objects is helpful if several instances of a feature are needed within one model. Then, a feature class could be defined that can be reused as-is or extended as needed.
The notion of abstract and concrete objects or features is used to represent (in-)completeness, so that a tool or the user can detect where the domain description is sufficient and where product-specific extensions must be made.
Finally, a feature model representation should enable automatic reasoning and strategies to relieve the user from evaluating composition rules and rationales.
At some points tool support for feature modeling will be quite similar to tool support for requirements management. Again we need

- text editing facilities;
- flexible filtering and view definition;
- easy link handling and restructuring;
- requirements traceability;
- comfortable navigation facilities.

The respective capabilities of QSS DOORS have been discussed above. Additional requirements with respect to feature modeling are

- structure and semantics visualization;
- user guidance for configuration.

During the derivation process, the user should be able to see where various kinds of variations are, i. e. which decisions the user must make to specify a product in the domain. Structure and semantics visualization is not fully supported by QSS DOORS. Especially links and link attributes are not adequately presented in the graphical view. As we used links and link attributes to denote the different composition rules (e. g., "requires" and "mutually exclusive with"), the lack of an adequate presentation was a major inhibitor to understanding the model. At least the tree presentation gave us the chance to capture the feature taxonomy and partonomy of our domain, with the feature names as node names and the feature types ("mandatory", "optional", "alternative") as tool tips.
Of course, a requirements management tool cannot provide any user guidance for configuration. Such guidance should include automatic evaluation and visualization of the composition rules and rationales. To this end, we think that feature models must be formalized so that automatic reasoning and strategy definition become possible.

4 RELATED WORK

The feature-oriented concept of the FODA method [Kang et al., 1990] places special emphasis on the identification and concise representation of commonality and variability in a domain. A feature is understood as "a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems". Features can be related to another by several types of links. Together the features form the feature tree which is used to parameterize all other models. Use case modeling has been used as a high-level functional description from the viewpoint of application families in the Reuse-Driven Software Engineering Business (RSEB) [Jacobson et al., 1997]. The approach introduces the notions of variation points and variants into use cases and analysis types. FeatuRSEB [Griss et al., 1998] and FODAcom [Vici et al., 1998] contribute to domain analysis in that they relate variability management in feature and use case modeling. Our work draws on these approaches and extends them with procedures and experiences from our experiment. Requirements templates, as in FODAcom, are a main point within our work. We use them for requirements derivation starting from feature configurations. An approach to applying feature modeling for controlling code generation is described by [Czarnecki and Eisenecker, 1999]. Over and above this, we are trying to apply feature modeling to software design generation.

5 CONCLUSIONS

The PRAISE experiment has shown that traditional tools are out of their range of applicability on the domain level. Generally, we used tools that were designed to support the analysis, design and implementation of single products, not product families. The additional levels of flexibility needed to manage, analyze and arbitrate among sets of requirements or features related to a specific aspect rather than individual requirements or features were completely lacking. Support for the traceability between and among the domain vocabulary, requirements, and features was lacking in an explicit form, and the adaptations or workarounds that were available were largely unsatisfactory.
At the moment, a tool user is left to his own devices in supporting his needs during domain requirements analysis, during features analysis and while tracing the relationships among these elements. In the end there must be recognized standard concepts for the philosophy underlying these activities.

6 OUTLOOK

From the experiment in the CPS domain we have learned that adequate tools must not only support the development *for* reuse but also the development *with* reuse for the investment in domain modeling to be effective. Therefore the representation of the domain knowledge must be formalized accordingly to enable a partly automated derivation process starting from the feature model. As knowledge representation and reasoning are AI topics, we think that integrating AI solutions into the Product Line Approach is a promising approach at this point.
Insofar as they have not considered the problematic or lack concrete plans about how to support product-line domain engineering, tool suppliers are invited to examine and consider our adaptation of the corresponding meta-processes. They may find indications for the direction of further tool development.

ACKNOWLEDGEMENTS

We would like to thank our colleagues from the Robert Bosch GmbH business units for their contribution to the model realization, and our PRAISE project partners Thomson-CSF/LCR, and ESI for the technical discussions.

REFERENCES

[Czarnecki and Eisenecker, 1999] Czarnecki, K., and Eisenecker, Ulrich W.: Synthesizing objects. In: Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99), Springer, 1999.

[ESI, 1998] European Software Institute: Product-line architectures and technologies to manage them. ESI-1998-RECORD-ED1, 1998.

[Griss et al., 1998] Griss, M. L., Favaro, J., and d'Alessandro, M.: Integrating feature modeling with the RSEB. Proceedings of the 5th International Conference on Software Reuse (ICSR'98), IEEE Computer Society Press, Victoria BC, Canada, June 2-5, 1998, ISBN 0-8186-8377-5.

[Jacobson et al., 1997] Jacobson, I., Griss, M., and Jonsson, P.: Software reuse – architecture, process and organization for business success. Addison Wesley Longman, 1997.

[Kang et al., 1990] Kang, Kyo C., Cohen, Sholom G., Hess, James A., Novak, William E., and Peterson, A. Spencer: Feature-Oriented Domain Analysis (FODA). Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Software Engineering Institute, 1990.

[PRAISE] ESPRIT Project 28651: PRAISE – Product-line Realisation and Assessment in Industrial Settings. IT RTD Project Programme, 1998, http://www.esi.es/Projects/Reuse/Praise/.

[Schlick and Hein, 2000] Schlick, M., and Hein, A.: Knowledge Engineering in Software Product Lines. European Conference on Artificial Intelligence (ECAI 2000), Workshop on Knowledge-Based Systems for Model-Based Engineering, August 22, 2000, Berlin, Germany.

[Vici et al., 1998] Vici, Alessandro Dionisi, Argentieri, Nicola, Mansour, Azza, d'Alessandro, Massimo, and Favaro, John: FODAcom: An Experience with Domain Analysis in the Italian Telecom Industry. Proceedings of the 5th International Conference on Requirements Engineering (ICRE'98), IEEE Computer Society Press, Victoria BC, Canada, June 2-5, 1998, ISBN 0-8186-8377-5.

[i] This paper refers to QSS DOORS 4.1.4, QSS DOORS Rose Link 2.0, and Rational Rose 98i.

Modellbasiertes Requirements Engineering für Produktfamilien

Günter Böckle, Annette Schreiber
Siemens AG, Corporate Technology, D-81730 München

Zusammenfassung

Produktfamilien Engineering ist ein Lösungsansatz, um verschiedene Produkte bzw. Produktvarianten durch Wiederverwendung von sog. Core Assets zu entwickeln, mit dem Ziel, die Time-to-Market und Kosten zu reduzieren.
Requirements Engineering ist ein wesentlicher Bestandteil in der System und Software Entwicklung, für das Produktfamilien Engineering sind jedoch spezielle Anpassungen erforderlich. Requirements Engineering im Kontext des Produktfamilien Engineering erfordert das Nachvollziehen (Tracen) unterschiedlicher Requirements für unterschiedliche Produkte, für unterschiedliche Architekturinstanzen und Komponentenvarianten. Requirements Engineering findet sowohl beim Erstellen der Core Assets (im Domain Engineering) als auch beim Verwenden der Core Assets (im Application Engineering) statt. Ein gutes Requirements Engineering Modell kann alle Teilprozesse unterstützen und erlaubt das Tracing von Requirements zu Architektur, Komponenten und Tests. Der Beitrag stellt unseren Ansatz für ein Requirements Engineering Modell für Produktfamilien vor.

1 Einführung und Motivation

Produktfamilien Engineering unterstützt die systematische Wiederverwendung aller möglichen Artefakte aus dem Entwicklungsprozeß. Im Rahmen des Produktfamilien Engineering Prozesses müssen Maßnahmen zur Wiederverwendung bereits zu Beginn mit Marktanalysen und Requirements Engineering starten. Der Hauptunterschied zwischen Produktfamilien Engineering und dem Engineering eines einzelnen Produkts ist die Unterteilung des Lebenszyklus in zwei Teilprozesse: dem Domain Engineering und dem Application Engineering (siehe Abbildung 1).
Requirements Engineering spielt eine wichtige Rolle in beiden Teilen des Lebenszyklus, wie dargestellt in [Clements99] und verschiedenen anderen Publikationen. Modellbasiertes Requirements Engineering kann die Grundlage für Produktfamilien Engineering bieten. Während im dokumentenbasierten Requirements Engineering Requirements in Dokumenten repräsentiert, gespeichert und geändert werden, werden beim modellbasierten Ansatz Requirements in einem Modell (einer Datenbank) repräsentiert, gespeichert und geändert. Das Requirements Engineering Modell enthält Requirements-Objekte, sowie die Verbindungen zu Design-, Realisierungs- und Test-Artefakten. Requirements-Dokumente sind somit nur Momentaufnahmen zu spezifischen Meilensteinen.
Modellbasiertes Requirements Engineering im Produktfamilien Engineering stellt ein Modell zur Verfügung, auf dem sowohl die Domänenanalyse als auch das Application Requirements Engineering arbeiten und das ein Tracing von der Marktanalyse über Benutzer-Requirements zu Design-Entscheidungen und Designspezifikationen ermöglicht. Es ist erforderlich, ein solches Modell im Produktfamilien Engineering Prozeß einzuführen, um zu ermöglichen, daß alle Beteiligten eine konsistente Sicht auf den Projektstand haben und um zu gewährleisten, daß Inkonsistenzen in Design und Entwicklung sofort entdeckt werden können.
Requirements Engineering wird in diesem Modell sowohl im Domain Engineering (in der Domänenanalyse) sowie im Application Engineering betrieben. Es dient auch als Grundlage

für die Commonality- und Variability-Analyse, die Teil der Domänenanalyse ist. Das Modell beinhaltet als Objekte Requirements, Architekturmodule, Use Cases, Komponentenbeschreibungen etc.

2 Der System- Lebenszyklus-Prozess und seine Anforderungen an ein Requirements Engineering Tool

Der System-Lebenszyklusprozess für Produktfamilien unterscheidet sich vom System-Lebenszyklusprozess für einzelne Produkte vor allem durch das Domain Engineering (siehe Abbildung 1).
Zu Beginn des Prozesses werden Requirements für die gesamte Produktfamilie gesammelt. In einer Commonality und Variability Analyse werden die gemeinsamen Requirements aller Produkte (*Common Requirements*) und die produktspezifischen Requirements (*Discriminating Requirements*) ermittelt. Die Requirements werden verfeinert, es werden neue Requirements abgeleitet.
Bei der Entwicklung eines neuen Produkts als Teil des Application Engineering wird ein neues Produkt-Modell, d.h. ein produktspezifisches Requirements Engineering Modell, angelegt. Alle produktspezifischen *Discriminating Requirements* werden in das Produkt-Modell kopiert.
Das gesamte Requirements Engineering und Teile von Design und Test werden auf dem Modell ausgeführt. Alle Requirements Spezifikationen sowie ein Teil der Design und Test Spezifikationen kann automatisch aus dem Modell generiert werden mittels eines Requirements Engineering Tools. Das Tool gewährleistet eine umfassende und konsistente Sichtweise aller Beteiligten auf das Produktfamilien-Modell und die Produkt-Modelle.

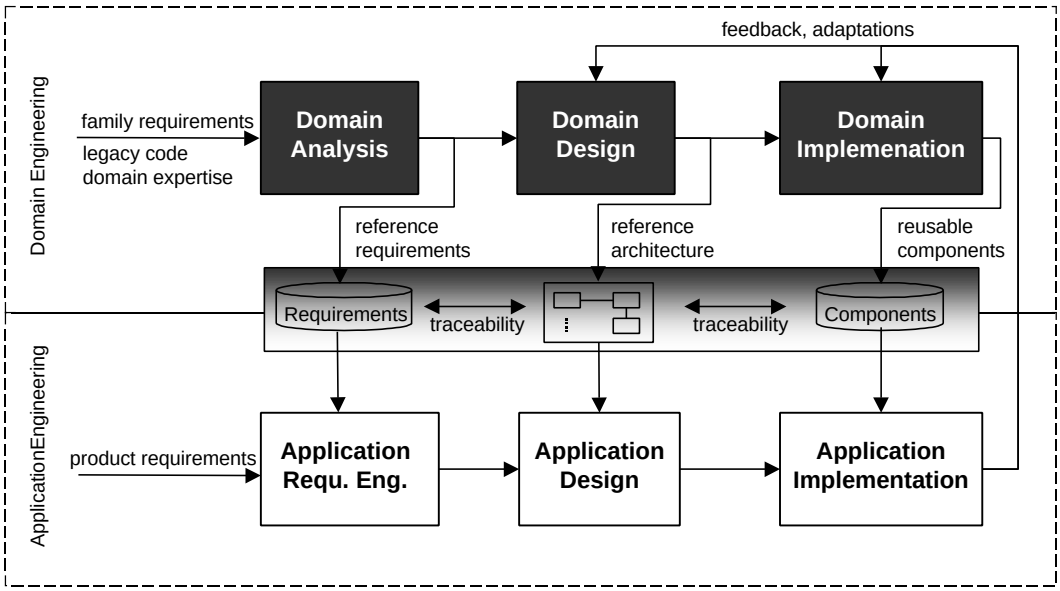


Abbildung 1: Produktfamilien Engineering Prozeß [ESAPS99]

Die wesentlichen Anforderungen für ein Requirements Engineering Tool, das Produktfamilien Engineering unterstützt, sind:

- Das Tool muß Domain Engineering und insbesondere Commonality und Variability Analyse unterstützen.

- Es muß für jedes Produkt der Produktfamilie ein eigenes (Sub-) Modell existieren als Grundlage für die Produktentwicklung.
- Das Modell muß mehrere parallel arbeitende Gruppen unterstützen und Inkonsistenzen der parallelen Arbeit aufzeigen.
- Die Objekte des Modells müssen in allen (Sub-) Modellen – insbesondere auch in denen der Produkte – konsistent gehalten werden.
- Das Modell muß in den Produktfamilien Engineering Prozeß eingebettet sein und diesen widerspiegeln.
- Entscheidungen und Änderungen, die in einem Produktmodell oder im Produktfamilien-Modell ausgeführt werden, müssen in zwei Richtungen nachvollziehbar und aufgezeichnet sein: vom Ursprung zu allen betroffenen Objekten und umgekehrt.

3 Modellstruktur

Die von uns betrachteten Objekte im Requirements Engineering Modell sind attributierte Textfragmente. Die Objekte umfassen Benutzer-Requirements, System-Requirements, Use Cases, Test Requirements, Designspezifikationen, Architekturmodule usw. Die Objekte werden in Mengen zusammengefaßt. Diese Mengen bilden (Sub-) Modelle. Diese können ihrerseits in Submodelle unterteilt werden. Die Objekte des Modells sind miteinander verknüpft, so daß Abhängigkeiten zwischen den Objekten nachvollzogen und aufgezeichnet werden können.
Das in diesem Beitrag vorgestellte Requirements Engineering Modell besteht aus einer Menge von Submodellen und *User Areas*. Jeder Benutzer darf innerhalb seiner *User Area* Objekte hinzufügen, löschen oder verändern. Auf die Objekte im Produktfamilien-Modell und in den Produkt-Modellen können die Benutzer nur lesend zugreifen. Änderungen bzw. Änderungswünsche bzgl. der Objekte dieser Modelle erfolgen nur gemäß einem definierten Change Management Prozeß [Deliv D3.2].
Entsprechend diesem Change Management Prozeß werden akzeptierte Änderungen zunächst auf einer lokalen Kopie des Modells innerhalb der User Area durchgeführt. Erst nach einem Review und einer Freigabe werden die geänderten Objekte aus der *User Area* in das entsprechende Modell kopiert. Das Produktfamilien-Modell wird im Domain Engineering, die Produktmodelle im Application Engineering verwendet. Abbildung 2 zeigt die Struktur der (Sub-) Modelle.
Produktfamilien-Modell und Produkt-Modell bestehen aus drei großen Submodellen:
- dem Dokumentenmodell,
- dem Requirements-Modell und dem
- Systemmodell.

Das *Dokumentenmodell* beinhaltet alle Dokumente, die in das Produkt- Modell oder das Produktfamilien-Modell importiert werden, alle Dokumente, die aus dem Modell erzeugt werden, und alle Templates, die für die Erzeugung der Dokumente verwendet werden.
Für das Projektmanagement werden aus dem Modell Bewertungen/ Auswertungen durchgeführt und die Ergebnisse zusammengefaßt. Diese Auswertungen beinhalten Daten über den Projektstatus, Testabdeckung etc. Requirements- und Designspezifikationen, System- und Software-Architektur-Spezifikationen und werden aus dem Model erzeugt.

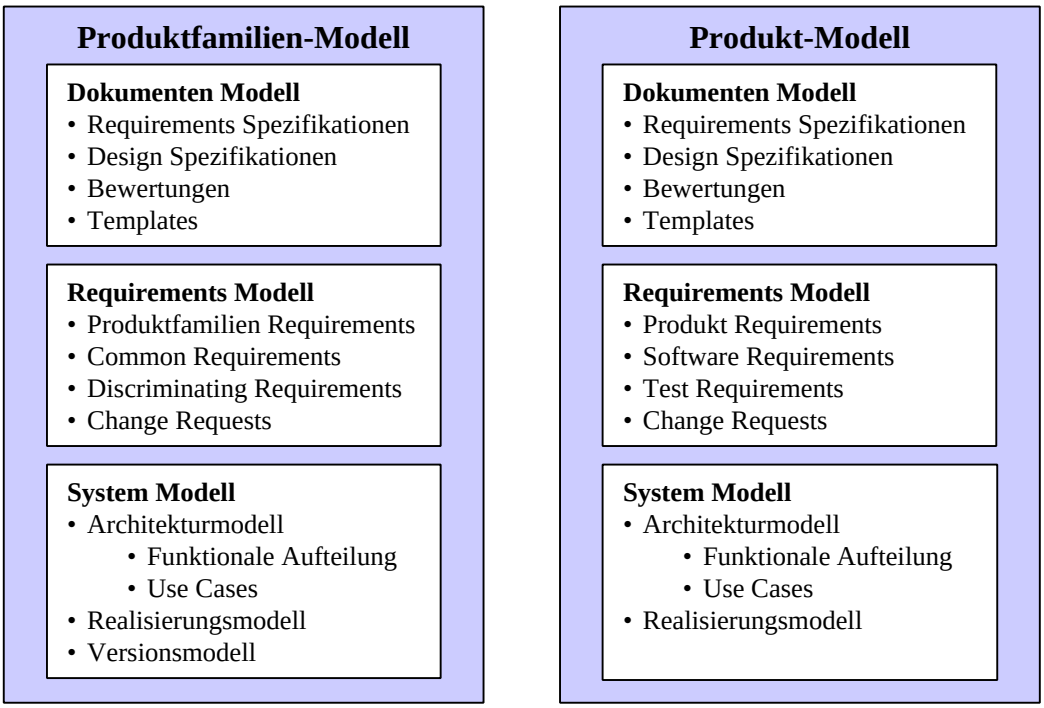


Abbildung 2: Produktfamilien-Modell vs. Produkt-Modell

Das *Requirements Modell* des Produktfamilien-Modells beinhaltet zunächst alle Requirements, die aus Dokumenten extrahiert wurden. Diese Requirements werden in einer ersten Analysephase verfeinert und erweitert. In der Commonality und Variability Analysis werden die Requirements untersucht und differenziert nach Requirements, die allen Produkten gemeinsam sind, die sog. *Common Requirements*, und Requirements, die produktspezifisch sind, die sog. *Discriminating Requirements*. Die Discriminating Requirements eines bestimmten Produkts werden in das Produkt-Modell kopiert und mit dem ursprünglichen Requirement im Produktfamilien-Modell verknüpft. Die Requirements werden im jeweiligen Produkt-Modell verfeinert in Software-, System- und Test-Requirements. Eine spezielle Art von Requirements sind Change Requests, die – wenn sie vom CCB akzeptiert wurden – im Produkt-Modell bzw. im Produktfamilien-Modell aufgenommen werden. Die adressierten Objekte werden entsprechend dem Change Request geändert.

Das *Systemmodell* umfaßt das Architekturmodell, das Versionsmodell, das Realisierungmodell und Use Cases als Bestandteil des Architekturmodells. Das Versionsmodell ist lediglich Bestandteil des Produktfamilien-Modells. Versionierung erfolgt orthogonal zu den Produktvarianten. Eine Version ist spezifisch für die ganze Produktfamilie, aber nicht alle Produkte müssen eine spezifische Version realisieren.
Das Realisierungmodell enthält Verknüpfungen zu Designobjekten in CASE Tools. Das Realisierungmodell enthält außerdem ein Testmodell für die Testrealisierung mit Verknüpfungen zu Testprogrammen.

4 Traceability

Das Modell unterstützt Traceability zwischen Requirements innerhalb eines (Sub-) Modells sowie zwischen Requirements verschiedener (Sub-) Modelle. So unterstützt das Modell Traceability von Benutzer-Requirements zu Design, Komponenten und Testmodulen und

umgekehrt, sowie die Traceability zwischen Produktfamilien Requirements und Produkt Requirements. Dies ermöglicht u.a. auch nachzuvollziehen, wie sich ein Requirement über die Zeit in verschieden Produkten fortentwickelt (history tracing).

Verknüpfungen (Links) zwischen Objekten im Modell sind in dem eingesetzten Tool wiederum typisierte Objekte. Vordefinierte Link-Typen sind beispielsweise *derived*, *causes* und *is-in-version*. Innerhalb eines Requirement Sub-Modells sind die Requirements zu ihren verfeinerten Requirements mittels *causes-Links* verknüpft. Die *causes*-Links spiegeln somit den Design-Fortschritt wieder. *Causes-Links* werden ebenso verwendet zur Verknüpfung von Requirements zu der davon abgeleiteten Architektur.
Die Verknüpfung von Discriminating Requirements des Produktfamilien-Modells zu den Requirements des Produkt-Modell wird über *derived-Links* modelliert. Auf diese Weise können bei einem Change Request für ein Discriminating Requirement automatisch alle davon betroffenen Requirements aufgezeigt werden. Requirements eines Produkts, die mit den Requirements eines anderen Produkts in Beziehung stehen, werden ebenso über *derived*-Links repräsentiert, um auf diese Weise in der Lage zu sein, Auswirkungen bei der Änderung eines Requirements auf die Requirements andere Produkte analysieren zu können.
Requirements können für verschiedene Produktversionen unterschiedlich sein, ein Requirement kann beispielsweise nur für bestimmte Versionen gültig sein. Um eine versions-abhängige Auswertung durchführen zu können, ist jedes Requirement mit seinem Versionsobjekt mittels eines *is-in-version*-Link verknüpft.

5 Schlußbemerkung

Das vorgestellte Requirements Engineering Modell wird in verschieden Siemens Abteilungen verwendet, z.B. bei Siemens Health Services (SHS). Das Modell wurde für den Einsatz in anderen Siemens Abteilungen erweitert, z.B. bei der Abteilung Automation and Drives.
Die Umsetzung des Modells erfolgte mit Hilfe des Requirements Engineering Tools SLATE von TD Technologies. Das vorgestellte Modell kann jedoch auch mit anderen Tools realisiert werden. Falls diese Tools Techniken wie die Erzeugung von Subtypen von Objekten nicht unterstützen – in vorgestellten Modell wurden beispielsweise typisierte Links eingeführt – können die verschiedenen Typen mittels Attributen und entsprechender Wertezuweisung realisiert werden.

6 Literaturverzeichnis

[Clements 99] P.Clements, L.M.Northrop: "A Framework for Software Product Line Practice – Version 2.0", http://www.sei.cmu.edu/plp/framework.html

[ESAPS 99] "ESAPS – Engineering Software Architectures, Processes and Platforms for System Families", ITEA Full Project Proposal, June 1999, http://www.esi.es/esaps.

[Deliv D3.2] J. Plankl, A. Schreiber: "Change Impact Analysis and Propagation Techniques", ESAPS Deliverable D3.1a (international), E1.2f (national), July 002000

Case Study of a Product Line Benefit and Risk Analysis

Klaus Schmid and Isabel John

Fraunhofer Institute for
Experimental Software Engineering (IESE)
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
+49 (0) 6301 707 - 158, +49 (0) 6301 707 - 250
Klaus.Schmid@iese.fhg.de, Isabel.John@iese.fhg.de

ABSTRACT[1]

Embarking on product line development has severe economical and organizational consequences for a company. This holds both in terms of potential benefits the organization is facing as well as risks it may encounter. At that point a disciplined way of evaluating the benefits and risks associated with a certain product line scope is needed.

In this paper we propose such an approach and discuss a case study which has been performed in the context of an industrial cooperation project.

Keywords: Product Lines for small and medium Enterprises (SME), Product Line Planning, Industrial Experiences with Product Lines

1 INTRODUCTION

These days many companies face the problem of developing and maintaining groups of software products, which strongly overlap in terms of functionality; so-called product families or product lines. By using a product line development approach, i.e., a software development approach which aims at developing the whole product line in an integrated manner, companies can, in principle, exploit these commonalities for achieving economic benefits. However, when deciding to start using product line development, a company needs to answer very important questions on the planning level in order to find the appropriate scope for the project:

1. Should we go for a product line development in the context of the particular family of products?
2. If so, should all technical areas be addressed in the same way? (Can we improve our benefit by focusing only on a subset?)
3. If we have the opportunity to invest in several different product lines, which one should we select (e.g., as a pilot)?
4. If we go for a product line development approach, which ones are the hurdles we need to face; what can we do in order to overcome them?

Answering these questions requires an analysis of the benefits and risks linked to the planned product line. As this proved to be a problem we at Fraunhofer IESE did repeatedly encounter when working on transferring product line technology into an industrial environment [1], we developed an evaluation approach, which is specifically tailored to the problem of analyzing benefits and risks of product line development. This approach, which is part of the Product Line Software Engineering framework (PuLSE™)[2] [2], is called PuLSE Benefit and Risk Analysis (PuLSE-B&R).

The core goal of PuLSE-B&R is to provide an overview of the potential benefits and risks of product line development. Several reasons lead us to choose the approach of an assessment for performing this analysis:

- It does not impose any preconditions for applicability onto an organization.
- It is a rather focused approach to evaluation, both requiring on the side of the product line group as well as on behalf of the evaluating organization a clearly limited amount of effort.
- The evaluation can be done within a short time interval (typically 2–3 weeks).

The focus of this approach is to identify and communicate the particular benefits and risks connected to product line development, as opposed to simply making the trade-off decision as is the focus in the only other product line assessment approach we know of at this point in time [4].

In the following section we will discuss in more detail the structure of our approach, namely how it was developed and what steps it does consist of. In Section 3 we will discuss a case study of applying this approach, which we did in an industrial context. Finally, in Section 4 we will present our conclusions on the potential of this approach and future development.

2 STRUCTURE OF THE APPROACH

When developing this approach we made the conscious decision to model this benefit and risk analysis as close as possible after existing work in the area of assessments, in order to take advantage of the existing body of knowledge in the area of software engineering related assessments.

1. This work has been partially funded by the ESAPS project (Eureka Σ! 2023 Programme, ITEA project 99005).

2. PuLSE™ is a registered trademark of Fraunhofer IESE.

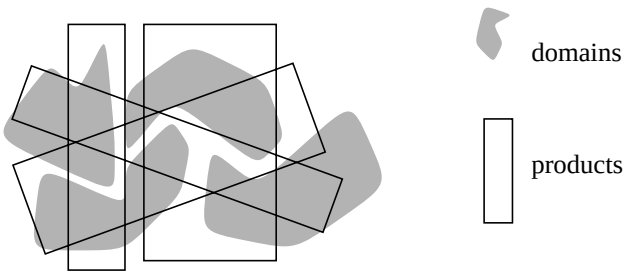


Figure 1. Relationship between products and domains

However, existing knowledge in assessment mostly focuses on process maturity assessments. There are actually some fundamental differences between the problem of process maturity assessments and product line benefit and risk assessments, e.g., with respect to the evaluation framework, or on the process level. However, the severest difference is that in the case of product line assessments, we cannot refer to a standardized framework for characterizing the assessment object. This is due to the fact that reference processes can be defined (as each assessment approach actually does), while there is no set of reference domains that is relevant to each product line in every organization. Instead, we need to develop for each product line a specific description framework. The approach to develop such a framework is called *product line mapping*. The overall process for performing this analysis is derived from the Fraunhofer IESE process assessment method FAME [3]. An overview of the product line assessment process is given in Figure 2.

As described in Figure 2, the process consists of three main phases: preparation, execution, and analysis. During the preparation phase a clarification of the objectives and the participating people is performed. This is followed by the detailed scheduling of the point in time when the various meetings are to happen.

Then the product line mapping is performed. This is a pre-study during which the relevant products and domains are identified. The main difference to existing domain scoping approaches is that in our approach a set of domains relevant to the product line is identified and their interaction is defined, while in the case of existing domain scoping approaches typically the domain defined by the product line as a whole is bounded. Thus, here we actually view the product line as being composed of domains.

The product line mapping is performed with the help of the product experts. It aims at identifying the specific products that are supposed to be part of the product line. It results in a high-level description of the products in the product line and their respective features. Further, the relevant technical domains of the product line are identified and their relationship to the products is described. The main tool for being able to define products and domains and for identifying the relationship between the two is by describing both in terms of features. Then both domains and products are described in terms of the features that are contained in them. Thus, the full (or partial) relevancy of domains to products can be easily identified. Figure 1 depicts this relationship graphically. This identification and description of the domains is key as the domains are the fundamental basis on which reusability is to be evaluated, as reusability can vary considerably across the different domains that are relevant even to a single product. A domain description mainly consists of a description of the functionality of the domain in terms of the features that are part of it and a description of the boundaries of the domain both in terms of functionality as well as data-handling responsibilities. Products differ in the extent to which they make use of the functionality contained in a domain.

In order to perform the analysis an evaluation framework was developed. This is captured in the form of questionnaires. These questionnaires aim at identifying specific benefit and risk situations for product line development. They were developed in a fashion based on the GQM-paradigm [5] and were augmented with information from existing domain assessment concepts [6,7,8] and success factor studies [9]. The different evaluation criteria were organized in seven different dimensions:

- Resource Constraints
- Organizational Constraints
- Market Potential (internal/external)
- Maturity/Stability
- Commonality and Variability
- Coupling and Cohesion
- Existing Assets

Each of these dimensions may point towards risks for product line development. However, not all dimensions will vary for all dimensions, e.g., organizational constraints may sometimes be constant across a subset of the domains, as the same organization is relevant for exploiting them. This is taken into account during the adaptation of the questionnaires.

After this adaptation has been performed we proceed with the actual analysis. This is based on interviews. Both the interviews themselves, as well as the latter evaluation of the results is driven from the questionnaires.

Based on the gathered information an initial analysis is performed and discussed with the interviewees in order to perform clarifications. Finally, the report with the analysis results is developed and discussed with the domain experts.

After we did now discuss the basic evaluation process, we will discuss a case study we did on applying this approach in an industrial environment.

3 THE CASE STUDY

In this section, we will discuss how we applied the approach described above in the context of a specific company.

3.1 The Context

The case study we report on in this paper was performed with MARKET MAKER a leading player in the market of software for stock market investment and information services for non-professionals in Germany.

Until 1999 the company mainly relied on a product line of windows-based charting systems. By the end of 1999 the company started to set up a new product line of online systems. These systems build on knowledge from the previous

systems, but repackage and extend this functionality in a way that is adapted to the web-based operation they are built for. As a result of these additional requirements and the web-based approach taken, these systems were implemented from scratch in Java.

For these new products a new organizational unit with an independent development team and project and product management was built. By August 2000, when this case study was performed, already a first system was launched and several additional systems were under development.

As the company already had some exposure to product line technology due to its cooperation with Fraunhofer and its participation in the ESAPS project, a strong reuse commitment and the application of product line concepts already strongly influenced the development.

3.2 Goal of the Case Study

The overall goals of this case study were:

- Analyze the reuse potential and risks of performing product line development in the context of this product family and suggest core areas for reuse, based on the identified risks and chances and suitable measures for the utilization of the chances or prevention of the risks
- Gather information on weaknesses and strengths of the evaluation approach and prepare the basis for further improvement of the approach

This analysis was performed in the context of a product line that was already under development. Thus, we had the advantage that we could additionally evaluate existing reuse mechanisms to point out possibilities for the improvement of exploiting existing reuse potential.

In order to evaluate our approach and be able to show the added value of this analysis, we asked an expert in advance for an initial rating of the reuse potential of the identified domains.

3.3 Execution of the Case Study

A product line mapping was already performed at an earlier point (January 2000). This was used as a basis for the interviews. In this case, 18 domains were identified. For each domain a domain definition, including a description of the content and the boundary of the domains, was available and as part of the interview set-up, experts for each domain were identified (actually, for all but one domain only one expert was identified). As typically one expert was responsible for more than one technical domain, a total of five different people were interviewed, while the responsibility of an expert ranged from a single domain to ten domains.

In a first interview, the correlation of the evaluation dimensions to the domains was identified. As expected some aspects (resource constraints, organizational constraints, and market potential) proved to be the same for all domains as the whole product line development was contained in a single organizational unit.

Thus, for each of the 18 domains there were four of seven dimensions left to be covered in the interviews with the domain experts and one general cross-domain interview for the above mentioned topics.

The interviews were made in close temporal sequence within one week. The interviews took from 10 to 30 minutes each, depending on size and complexity of the domain and experience of the interviewee with the domain and with the questions. All interviews were performed by two interviewers (the authors of this paper).

For each evaluation dimension there were about ten questions, the answers were given by the interviewees not in one word but in the form of a conversation. Both interviewers ranked the answers on a four-point scale during the interviews.

After the interviews, the results were consolidated and a detailed report was written. This report contains both a rating of the domains according to the various evaluation dimensions and an evaluation of the reuse potential and risks based on these results. Figure 3 through 5 show anonymized examples from the case study. Higher values on the scale (0–3) point to domains being more appropriate for

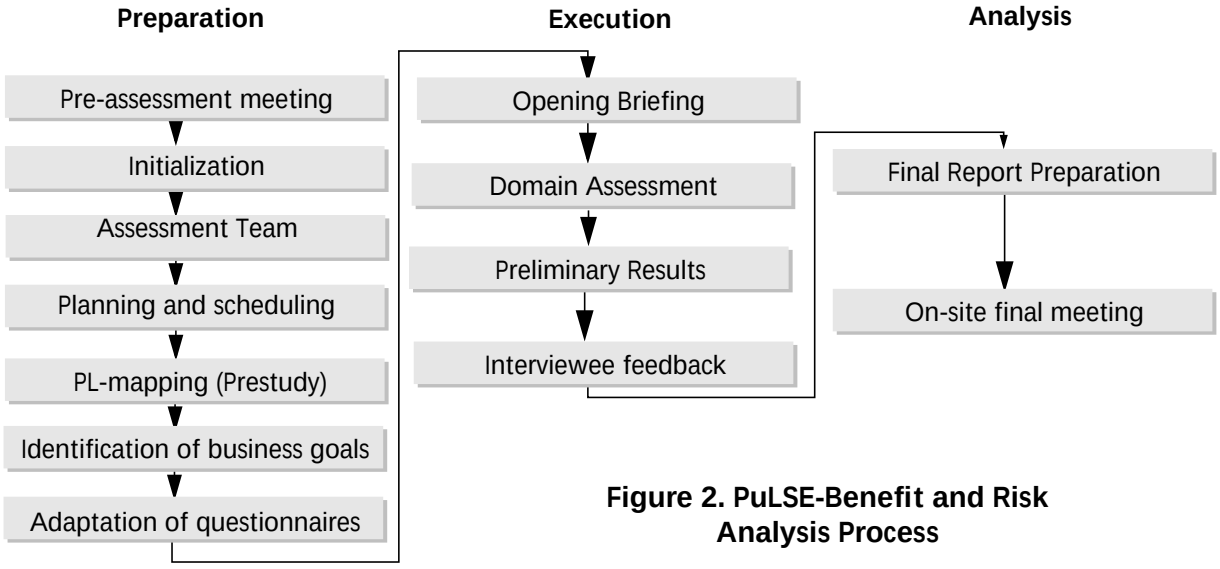


Figure 2. PuLSE-Benefit and Risk Analysis Process

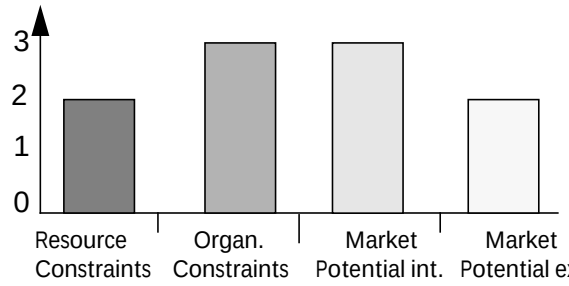


Figure 3. Analysis Results for Organ. Issues

product line reuse. Additionally a list of recommendations is developed as a result of the analysis.

The recommendations focus on possible next steps for an improvement of product line development and a short analysis of those domains with the highest benefits and the lowest risks when exploiting their reuse potential in the context of product line development.

3.4 Evaluation of the Case Study

The evaluation of the organizational aspects (cf. Figure 3) showed that the environment was very positive for product line development. This in combination with the fact that the company was as an IESE-partner and ESAPS project member well aware of the potential and implications of product line development made it clear from the beginning that the potential for improvement of the exploitation of the reuse potential would probably be rather limited. This was largely confirmed by the results of the study, however some interesting recommendations could still be given.

An interesting observation that could be made was that the values of the evaluation varied considerably over the various domains of the product line. This proved an initial assumption of the analysis approach, namely that an evaluation on the basis of the whole product line is too coarse-grained to produce adequate results and that more adequate results can be expected from a per-domain evaluation.

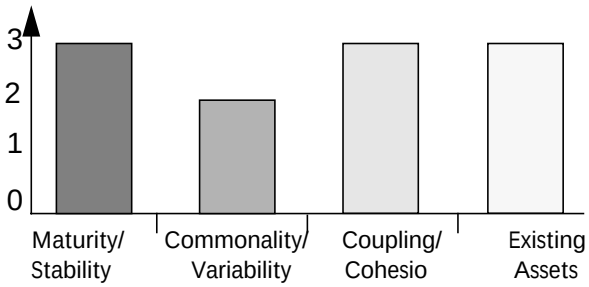


Figure 4. Analysis Result: Domain 1

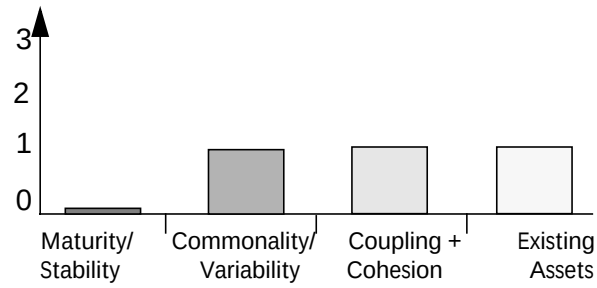


Figure 5. Analysis Results: Domain 2

The evaluation of the quality of the results was performed in three different ways:

- The ratings were performed independently by the different interviewers and only compared later. Their agreement (*interrater agreement*) was used as a basis to evaluate to which extent the evaluation framework was an adequate basis for person-independent ratings [10].
- During the initial interviews prior to the assessment the manager was asked for his initial expectation for the product line scope. After presentation of the results he was again asked to provide this scope expectation. If the assessment provided him with valuable information to make an informed scoping decision it was to be expected that the scope would change towards the proposal given by the assessment (*scope drift*).
- The individual evaluations were presented to the interviewed people and explained. Then they were asked to give their level of agreement to this rating (*evaluation agreement*).
- The agreement with the list of recommendations was evaluated (*recommendation agreement*).

When evaluating the *interrater agreement*, one has to keep in mind that the results were restricted to a four-point scale. Consequently, if the evaluation framework would not provide any guidance to the evaluation one would have to expect a random distribution of the answers over the scale. Given that the ratings of two interviewers were compared one would then expect a (random) agreement on 25% of the evaluations. However, when evaluating the interrater agreement, we came to a value of about 60% identical ratings and less than 1% ratings that differed by more than one level. Further, the disagreements were mostly due to borderline cases. In almost all cases, an agreement on a common value could be found after a short discussion of the background.

We also analyzed the *scope drift*. There, the impact was only limited, we found only six cases were the rating before the evaluation was different from the one after. Out of those six differences in four cases the rating moved into the direction of the assessment and two times the rating moved into the opposite direction. The interview during which these ratings were captured showed that the rationale on which the original evaluation was based was not fully clear. This showed two things: the way the information on this issue was gathered was problematic, probably causing this rather weak tendency, and it supported the hypothesis that a disciplined, traceable scoping method is needed for arriving at an appropriate scope definition.

The *evaluation agreement* was rather strong. So far feedback interviews on 12 domains have been performed. Partially multiple experts were relevant. On the basis of the total evaluation of the results per domain during the feedback, the experts rated these results (on a four point scale “strongly disagree”, “disagree”, “agree” and “strongly agree”) four times an agree and nine times a strongly agree, no disagree or strongly disagree was given. Thus, the evalu-

ation framework supports capturing the information about the domains in an appropriate form, which allows for example the aggregated analysis of this information.

Based on this information recommendations for the future exploitation of the reuse potential of the domain were given. Out of 11 recommendations, *recommendation agreement* on nine was achieved. There were two exceptions in terms of the priority they received due to additional market driven influences that were beyond the scope of the analysis. Thus this analysis also delivered quite favorable results for the analysis approach.

4 CONCLUSIONS

In this paper we presented a case study of a benefit and risk analysis for Product Lines. It was performed within a development unit in a medium sized company. Besides the general goal of the benefit and risk analysis to find out potential benefits and risks for each domain within the planned product line, our second goal was to gather information on weaknesses and strengths of the evaluation approach.

Overall the approach worked fine and gave good insight into the benefits and risks of the domain, for us as well as for the experts we interviewed. Useful recommendations could be developed from an outside perspective, the information gathered partially changed the views of the experts on their domain and so helped to give them a better basis for development with (and for) reuse within a product line.

During and after the interviews we made the following additional observations concerning our case study:

- The execution of the interviews was faster than expected
- The interviews were very interesting for the interviewed experts because it helped them to see their everyday work areas from outside and to get a different view on the domains
- The domains varied fundamentally in size (covered by some LoC up to few 1000), generality (from well know domains like Caching and Security to totally company or product specific areas) and in reuse potential

This case study will help us to improve our benefit and risk analysis in several ways: we will further improve the questions in order to remove remaining ambiguities and we will change the aggregation of some indicators. Additionally, we will use in a future a more fine-grained scale for analysis.

In summary, the analysis showed that the technology is ready and useful to transfer into industrial practice. Thus, performing further applications in industry of this technology is a major goal for us for the future.

5 REFERENCES

[1] Peter Knauber, Dirk Muthig, Klaus Schmid and Tanya Widen. *Applying Product Line Concepts in Small- and Medium-Sized Companies*. IEEE Software, Vol. 17, No. 5, pp. 88–95, 2000.

[2] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen and J.-M. DeBaud. *PuLSE: A Methodology to Develop Software Product Lines*, Proceedings of the Symposium on Software Reusability (SSR'99), pp. 122–131, 1999.

[3] Andrew Beitz and Janne Jarvinnen. *Fame — an approach for software process assessment.* Technical Report 001.00/E, Fraunhofer IESE, 2000.

[4] Sergio Bandinelli and Goiuria Sagardui Mendieta. *Domain Potential Analysis: Getting Serious About Product-Lines.* Third International Workshop on Software Architectures for Product Families, Las Palmas de Gran Canaria, Spain, pp. 75–81, 2000.

[5] Lionel C. Briand, Christiane Differding, and H. Dieter Rombach. *Practical guidelines for measurement-based process improvement.* Software Process Improvement and Practice Journal, 2(3), 1997.

[6] Department of Defense — Software Reuse Initiative, Version 3.1. *Domain Scoping Framework, Volume 2: Technical Description*, 1995.

[7] M. Davis. *Reuse strategy model: Planning aid for reuse-based projects.* Technical Report CDRL 5159, Software Technology for Adaptable, Reliable Systems (STARS), July 1993.

[8] Software Productivity Consortium Services Corporation. *Reuse Adoption Guidebook, Version 02.00.05,* November 1993.

[9] David C. Rine and Robert M. Sonnemann. *Investments in Reusable Software. A Study of Software Reuse Investment Success Factors*, Journal of Systems and Software, Vol. 41, No. 1, pp. 17–32, 1998.

[10] Khaled El Emam and Peter Marshall. *Interrater Agreement in Assessment Ratings*. In SPICE — The Theory and Practice of Software Process Improvement and Capability Determination. Chapter 16, pp. 357–362.

Modellierung von Produktfamilien

Behandlung von Variabilitäten in Produktlinien mit Schwerpunkt Architektur

Stefan Voget[1], Ilario Angilletta, Iris Herbst, Peter Lutz

Zusammenfassung
In diesem Beitrag werden Herausforderungen an die Architekturentwicklung vorgestellt, wie sie sich in einem sich stark wandelnden technologiegetriebenen Geschäftsfeld für die Erstellung einer Produktlinienarchitektur auftreten und es wird ein Ansatz präsentiert, wie damit umgegangen werden kann.

1 Einleitung

Eine Software Produktlinie ist eine Zusammenfassung von software-intensiven Systemen, die einen gemeinsamen Satz an Merkmalen, die den spezifischen Bedürfnissen eines speziellen Marktsegments genügen, teilen [CN99]. Die Entwicklung einer Software nach dem Produktlinienansatz erreicht vordringlich die Wiederverwendung von Arbeitsergebnissen durch die Ausnutzung des in der jeweiligen Organisation vorhandenen Expertenwissens aus ihrem Geschäftsfeld. Redundante Entwicklungstätigkeiten werden durch die Straffung des Entwicklungsprozesses eliminiert.
Der Entwicklungsprozeß nach dem Produktlinienansatz sieht die Aufteilung der Entwicklung in einen Teil mit der Entwicklung gemeinsamer Kernergebnisse, sogenannter Assets, und einer Produktentwicklung, die unter Ausnutzung der Assets erfolgt, vor. Der sogenannte „dual life cycle“, bei dem der erste Teil mit Domänenentwicklung und der zweite Teil mit Applikationsentwicklung bezeichnet wird, bildet das Framework für die Koordinierung dieser Assets [CW00].

1.1 Gemeinsamkeiten und Variabilitäten

In der Domänenentwicklung werden Gemeinsamkeiten von Produkten gesammelt und allen an der Produktlinie beteiligten Produkten in einem sogenannten Assetrepository zur Verfügung gestellt [W00]. Zusätzlich werden aber auch den Unterschieden zwischen Produkten in der Domänenentwicklung eine Schnittstelle zu den Gemeinsamkeiten zur Verfügung gestellt. Diese Schnittstelle ist in unterschiedlichen Produktlinien aus unterschiedlichen Geschäftsfeldern auf die verschiedenste Art und Weise ausgeprägt.
Für das Verständnis der weiteren Ausführungen über die erwähnte Schnittstelle werden zunächst zwei Begriffe definiert:

Definition: Eine Variabilität ist die Möglichkeit, ein System zu ändern oder anzupassen.

Desweiteren wird in diesem Text der Begriff Arbeitsergebnis als Stellvertreter für die im Laufe der Entwicklung entstehenden Dokumente (z.B. Dictionary, Feature Modell, Architekturmodell, Design Modell, ...) verwendet. Eine Variabilität hat Auswirkungen auf eines oder mehrere Arbeitsergebnisse. Die betroffenen Arbeitsergebnisse müssen Techiken zur Verfügung stellen, so dass die Variabilität realisierbar ist.

Definition: Eine Stelle in einem Arbeitsergebnis einer Produktlinie, die durch eine Variabilität betroffen ist und Techniken zu deren Realisierbarkeit zur Verfügung stellt, heißt Variationspunkt.

Die Unterschiede zwischen Produkten einer Produktlinie werden durch die Definition von Variabilitäten innerhalb der Produktlinie explizit hervorgehoben. Die Schnittstelle zwischen Variabilitäten und Gemeinsamkeiten in den Arbeitsergebnissen wird durch die Variationspunkte dargestellt.
Variabilitäten werden in Arbeitsergebnissen durch Variationspunkte gelöst. Der Ausdruck „gelöst“ ist dabei so zu verstehen, dass ein Variationspunkt in dem jeweiligen Arbeitsergebnis diejenigen Techniken zur Verfügung stellt, die notwendig sind, um die mit der Variabilität induzierten Änderungsmöglichkeiten durchführen zu können. Eine Variabilität kann dabei durchaus durch mehrere Variationspunkte in einem Arbeitsergebnis aber auch in mehreren Arbeitsergebnissen gelöst werden.

1.2 Domänenanalyse und Feature Modell

Die Domänenanalyse ist

[1] Dr. Stefan Voget; Robert Bosch GmbH; Zentralbereich Forschung und Vorausentwicklung Softwaretechnologie; Theodor-Heuss-Allee 70, D-60486 Frankfurt am Main, e-mail: stefan.voget@de.bosch.com

„The process of identifying, collecting, organizing, and representing the relevant information in a domain based on the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within the domain.“ [KCHNP90]
Eines der wesentlichen Arbeitsergebnisse der Domänenanalyse ist das Feature Modell. Das Feature Modell faßt die vom Benutzer sichtbaren Aspekte der Produktlinie zusammen. Ein weit verbreiteter Ansatz zur Erstellung eines Feature Modells ist die „Feature Oriented Domain Analysis“ (FODA) Methode vom Software Engineering Institute (SEI) in Pittsburgh [BCK98]. In dem Feature Modell werden neben den gemeinsamen Merkmalen auch grundlegende Variabilitäten festgelegt. Damit erhält die Produktlinie ihre Struktur und eine Abgrenzung zu Änderungsmöglichkeiten, deren Behandlung nicht mehr als Bestandteil der Produktlinie angesehen werden.
Nicht alle Anforderungen entstehen ausschließlich durch Kunden und kommen über Lastenhefte oder sonstige Kundenkontakte in das Feature Modell. Durch den zunehmend sich beschleunigenden technologischen Wandel ist es insbesondere in technologiegetriebenen Geschäftsfeldern für den Erhalt und die Erhöhung von Marktanteilen notwendig, aktiv neue Anforderungen und damit neue Märkte zu schaffen. Diese zu erschließen erfordert insbesondere, zukünftige Anforderungen vorauszuahnen. Die dazu notwendigen Anforderungen entstehen mit der Technologie und damit auch unter anderem mit der Modellierung der Architektur.

1.3 Architektur

In der Architektur werden die ersten Modellierungsentscheidungen in einem Grobdesign zusammengefaßt. Es ist ein zentrales Arbeitsergebnis, in dem wesentliche richtungsweisende Entscheidungen für die Entwicklung aller an der Produktlinie beteiligten Produkte getroffen werden. Ein Ansatz zur Erstellung eines Architekturmodells ist die „Architecture Based Design“ (ABD) Methode vom SEI, die insbesondere einen Fokus auf die explizite Betrachtung von Architekturtreibern und Architekturstilen vorsieht [BBCDP99, CN96].
Die Architektur einer Produktlinie unterstützt die gemeinsamen Anforderungen aller an der Produktlinie beteiligten Produkte und diejenigen Anforderungen, die sich innerhalb der Möglichkeiten einer in der Produktlinie definierten Variabilität unterscheiden.
In einem technologiegetriebenen Umfeld kann die Architektur auch dazu dienen, neue Varianten von Anforderungen zu entwickeln. In einer Produktlinie stellen die Variationspunkte die dafür notwendigen Bedingungen zur Verfügung.

Im folgenden Abschnitt wird ein Ansatz vorgestellt, wie die Schnittstelle zwischen Variabilitäten und Gemeinsamkeiten in dem Arbeitsergebnis Architektur aussehen kann. Dieser Bericht fokussiert auf die Architektur und die Zusammenhänge, die sich zwischen der Architektur und dem Feature Modell bei der Arbeit mit Variabilitäten ergeben. Teile der Betrachtungen lassen sich allerdings ebenso auf weitere Arbeitsergebnisse übertragen, auch wenn sie an dieser Stelle nicht in der allgemeinen Form behandelt werden.

2 Aspekte zur Handhabung von Variabilitäten

In einigen Fällen ist die Rolle der Architektur derart fundamental für die Domäne, dass sämtliche Produkte von einer gemeinsamen Architektur, der Produktlinienarchitektur abgeleitet werden [ESI98]. Die Beherrschung sämtlicher Produkte aus einer gemeinsamen Architektur heraus erfordert ein weitreichendes und präzises Verständnis der Variabilitäten zwischen den Produkten. Wird in der Domänen- und der Applikationsentwicklung mit einem gemeinsamen Architekturmodell gearbeitet, so ist es gegebenenfalls ratsam, die Behandlung der Variabilitäten aus der Architektur herauszuziehen und in einem Entscheidungsmodell zu bündeln. Nicht mehr das Architekturmodell selber zeigt die Unterschiede zwischen einzelnen Produktarchitekturen (die lediglich noch implizit existieren) und der Produktlinienarchitektur zu den Produktarchitekturen. Diese Aufgabe übernimmt das Entscheidungsmodell.
Dieser Abschnitt befaßt sich zunächst mit den Informationen, die in Zusammenhang zu einer Variabilität gesammelt werden müssen und die in dem von der Produktlinienarchitektur separierten Arbeitsergebnis, dem Entscheidungsmodell, zusammengefaßt werden.

2.1 Zustände, die Variabilitäten durchlaufen

Während der Entwicklung der Arbeitsergebnisse durchlaufen auch die Variabilitäten mehrere Zustände:

- Implizit

  Idealer Weise werden die Varianten von Anforderungen im Feature Modell definiert. In technologiegetriebenen Geschäftsfeldern kommen die Ideen zu Variabilitäten nicht ausschließlich von den Kundenanforderungen sondern auch aus der Technologie heraus, deren wesentlicher Repräsentant im Entwicklungsprozeß die Architektur darstellt. Daher werden in der Architektur Variabilitäten gefunden, die im Feature Modell zunächst nicht explizit vorhanden sind.

- Definiert
  Mit der Definition von Variabilitäten im Feature Modell werden
  Richtungen vorgegeben, in die die einzelnen Produkte voneinander abweichen können. Eine klare zielorientierte Definition ist für die Beherrschung der Unterschiede eine notwendige Vorraussetzung.
- Gebunden
  Eine Variabilität induziert eine Entscheidung, die von Produkt zu Produkt unterschiedlich ausfallen kann. Wird eine Entscheidung getroffen, so wird die Variabilität gebunden. Ab dem Entscheidungszeitpunkt ist die Variabilität im Produkt nicht weiter existent. Eine Entscheidung kann im Prinzip jederzeit erfolgen. Es bieten sich allerdings strategisch günstige Zeitpunkte im Entwicklungsprozeß für die Entscheidung von Variabilitäten an. Der nächste Abschnitt befaßt sich mit dieser Fragestellung.

Wenn eine Variabilität auf einer speziellen Abstraktionsebene definiert wird, so ist diese auch auf jeder höheren Abstraktionsebene vorhanden. Insbesondere also in dem Feature Modell und/oder der Architektur. In einem vollständigen Feature Modell und einer vollständig modellierten Architektur sind alle für die Produktlinie relevanten Variabilitäten definiert.

2.2 Bindungszeiten

Eine Variabilität wird durch Variationspunkte in den Arbeitsergebnissen gelöst. Erst die Art der Lösung induziert einen Bindungszeitpunkt. Als Bindungszeitpunkte für eine Variabilität bieten sich markante Meilensteine des jeweils verwendeten Softwareentwicklungsprozesses an. Beispielhaft bieten sich in einem Entwicklungsprozeß die folgenden Zeitpunkte an:

- Design von Komponenten
- Implementierung von Komponenten
- Konfigurierung beim Hersteller
- Konfigurierung beim Kunden
- Laufzeit, durch Aufruf

Aufgrund der vorherigen Ausführungen zur Architektur tauchen in dieser Auflistung die Produkt- bzw. Produktlinienarchitekturen nicht auf. In der Produktlinienarchitektur werden Variationspunkte modelliert aber nicht gebunden, da der vorgestellte Ansatz keine Trennung von Produktlinienarchitektur und Produktarchitektur vorsieht.

Eine Variabilität wird in einigen Fällen durch mehrere Konzepte in den Arbeitsergebnissen unabhängig voneinander gelöst. Mit jeder Lösung kommt eine weitere Bindungszeit hinzu. Dadurch ergeben sich für eine Variabilität durchaus mehrere Bindungszeiten. Jede Lösung hat aber auch Konsequenzen. Liegt z.B. ein Bindungszeitpunkt sehr spät, so ist die Variabilität sehr lange im Entwicklungsprozeß zu unterstützen. Dieses bedeutet unter Umständen erhöhter Implementierungsaufwand, erhöhter Speicherplatzbedarf oder auch erhöhte Kommunikation zwischen Komponenten der Software. Ein Beispiel ist das Konzept der konfigurierbaren Listen. In der Mehrzahl der Variabilitäten ließe sich eine konfigurierbare Liste realisieren und damit ein später Bindungszeitpunkt erreichen. In der Praxis ist es aber nicht realisierbar, da der erhöhte Implementierungsaufwand in den allermeisten Fällen dieses nicht rechtfertigt.

2.3 Kategorisierung

Zur Bindungszeit ist für eine Variabilität eine Entscheidung zu treffen. Welche Entscheidungsoptionen man jeweils hat, läßt sich in einige wenige Kategorien einsortieren.

- **Variabilität umfaßt mehrere Optionen**
  - **(∞) Abstrakt:** Unterspezifikation ist ein Mittel in der Modellierung einer Produktlinienarchitektur, Variabilitäten zuzulassen. Unterspezifikation ermöglicht unendlich viele Varianten und damit dem Designer große Freiheiten. Andererseits wird damit ein Teil der Modellierungsarbeit auf einen späteren Zeitpunkt verschoben. Insbesondere in der Produktlinienarchitektur besteht die Gefahr von auseinandergehenden Entwicklungen in den einzelnen Produktentwicklungen. Das reduziert die Möglichkeiten für eine Wiederverwendung von Arbeitsergebnissen, die durch die Arbeit mit dem Architekturmodell entstehen.
  - **(0..n) Freie Auswahl aus Optionen:** Ebenso wie die vorherige Kategorie erzeugt die Ermöglichung einer freien Auswahl aus den angebotenen Optionen (n=Anzahl der angebotenen Optionen) keine Abhängigkeiten zwischen den Optionen, die die Entscheidung zur Bindungszeit beeinflußt.

  Alle weiteren Kategorien erzeugen Abhängigkeiten zwischen den Optionen, die die Entscheidung zur Bindungszeit beeinflussen. Die Entscheidungsmöglichkeiten werden dadurch eingeschränkt.
  - **(1..n) Auswahl mindestens einer der Optionen:** Die Existenz mindestens einer der vorhandenen Optionen im Produkt ist verpflichtend. Ohne würde das Produkt nicht funktionsfähig sein.
  - **(0..1) Auswahl von nicht mehr als einer der Optionen**
  - **(1) Auswahl von genau einer der Optionen**

Weitere Kategorien, wie zum Beispiel (j..k), 0<j<k<n (n=Anzahl der angebotenen Optionen), sind zwar denkbar, allerdings häufig nicht durch das Architekturmodell begründet. Eine nicht durch die Architektur begründete Einschränkung reduziert unter Umständen überflüssiger Weise die Variantenvielfalt. In einem derartigen Fall ist zu überlegen, ob die Erweiterung der Variabilität (z.B auf (1..n) oder (0..n)) realisierbar und für eine sich in der Zukunft ergebende Anforderung sinnvoll sein könnte.

- **Variabilität umfaßt eine singuläre Option**
  Liegt lediglich eine Option zur Auswahl vor und ist dieses Feature optional, so kann dafür ebenfalls die vorher aufgeführte Kategorie (0..n) verwendet werden. Da in diesem Fall n=1 ist, fällt (0..n) hierbei mit (0..1) zusammen.

Eine Kategorisierung hilft während der Architekturmodellierung bei der Erkennung von relevanten Variabilitäten. Kann eine Variabilität keiner der oben aufgeführten Kategorien zugeordnet werden, so deutet das häufig auf ein nicht ausreichendes Verständnis des eigentlichen Kerns der Variabilität hin. Ein weiterer Modellierungsschritt, der die vermeintlich gefundene Variabilität verfeinert und klärt, kann notwendig sein.

2.4 Abhängigkeiten zwischen Variabilitäten

Variabilitäten können nicht immer derart modelliert werden, dass sie unabhängig voneinander sind. Die Entscheidung einer Variabilität für eine oder mehrere Optionen

- kann eine andere Variabilität überflüssig machen oder
- kann bei einer weiteren Variabilität eine Entscheidung für eine bestimmte Option bzw. für bestimmte Optionen erzwingen.

Beide Möglichkeiten einer Abhängigkeit sind „wenn...dann“ Abhängigkeiten. Die Entscheidung einer Variabilität erzwingt einen Einfluß auf eine zweite Variabilität. Umgekehrt legt die Entscheidung der zweiten Variabilität noch nicht unbedingt die Entscheidung der ersten Variabilität zwingend fest. Desweiteren ist noch eine „genau dann, wenn ...“ Abhängigkeit, also einer wechselseitigen Erzwingung zweier Entscheidungen denkbar. In einem derartigen Fall ist allerdings zu überlegen, ob nicht das Verhältnis beider Variabilitäten derart eng aneinander geknüpft ist, dass die Zusammenlegung beider Variabilitäten zu einer einzigen Variabilität nicht zu einer Erhöhung der Klarheit führt.
Abhängigkeiten erschweren die Verständlichkeit für die notwendigen Entscheidungsschritte und sind während der Modellierung nach Möglichkeit zu vermeiden.

2.5 Tätigkeiten, die einer Entscheidung folgen

Wird eine Entscheidung getroffen, so bedingt dieses Folgetätigkeiten. Diese sind je nachdem, welches Arbeitsergebnis betroffen ist, unterschiedlich. Als Beispiel sei die logische Sicht (gemäß dem 4+1 Sichtenmodell aus [K95]) der Architektur betrachtet. Eine Entscheidung für eine Option kann im Architekturmodell die Auswahl einer spezifischen Komponente aus einer Reihe von optionalen Komponenten sein. In diesem Fall sind die nicht ausgewählten Komponenten zu löschen und in der weiteren Implementierungsarbeit nicht weiter zu berücksichtigen.

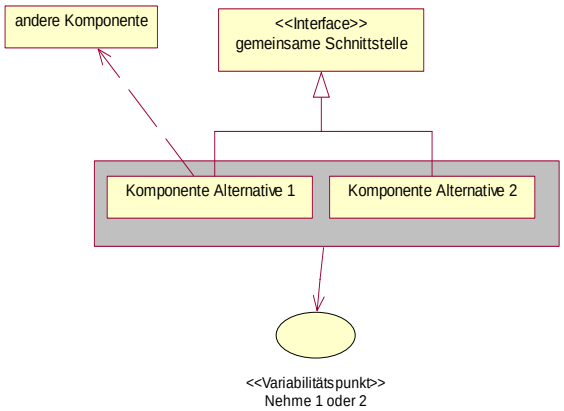


Abbildung 1: Auswahl einer Komponente aus zwei Optionen

In dem Beispiel aus Abbildung 1 wird eine Variabilität vom Typ (1) dargestellt. Der Variationspunkt wird durch eine Ellipse an der durch die Variabilität betroffenen Stelle in der Architektur hervorgehoben. Wird die Entscheidung für „Komponente Alternative 2“ getroffen, so ist die Alternative 1 zu löschen und darüber hinaus aber auch die Beziehung zu der „andere Komponente“. Die Entfernung der Beziehung kann unter Umständen weitere Auswirkungen auf das Schnittstellendesign der „anderen Komponente“ haben.

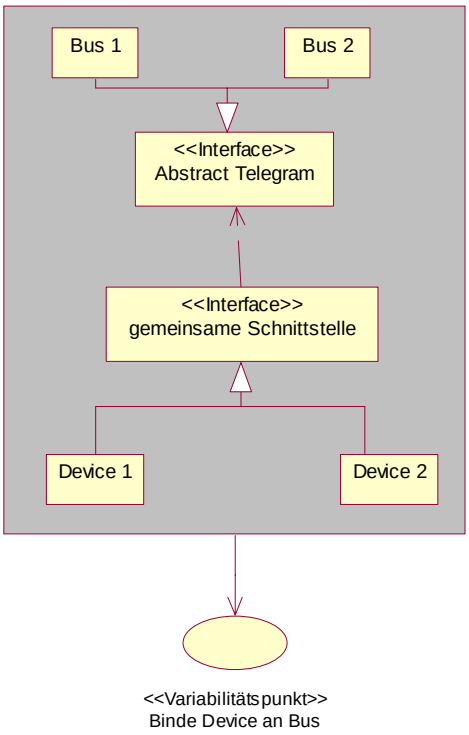


Abbildung 2: Binde einen Device an genau einen Bus

In der Abbildung 2 wird eine weitere durch einen Variationspunkt betroffene Stelle in einer Produktlinienarchitektur dargestellt. Ein Gerät kann über verschiedene Bussysteme an einen zentralen Rechner angeschlossen werden. Aus Gründen der Performance ist spätestens zur Implementierung eines Produkts eine Entscheidung zu treffen, um welches Bussystem es sich handelt. Hierbei handelt es sich ebenfalls um eine Variabilität vom Typ (1). Für die Architektur bedeutet die Entscheidung, dass die Abhängigkeitsbeziehung, die in der Produktlinienarchitektur zwischen den beiden Interfaces angesiedelt wird, für ein spezifisches Projekt direkt zwischen einer Komponente Device und einer Komponente Bus umzulinken ist. Zusätzlich wird durch die Nebenbedingung (die Performance ist von entscheidender Bedeutung) eine Bindungszeit vor der Implementierung festgelegt.
Wie die beiden Beispiele zeigen, sind die Tätigkeiten, die aus der Entscheidung folgen, unterschiedlich und können nicht einfach generalisiert werden.

2.6 Das Entscheidungsmodell

In den bisherigen Ausführungen wurden eine Reihe von Konzepten eingeführt, die für die Behandlung von Variabilitäten notwendig sind. Neben den Informationen, die durch das Feature Modell dargestellt werden, kommen Informationen hinzu, die für die eigentliche Verwendung des Feature Modells nicht benötigt werden und dieses überladen würden. Daher ist eine Auftrennung in ein Feature Modell und ein separates Entscheidungsmodell in dem sämtliche Informationen gesammelt werden, die für den Modellierungs- und Entscheidungsprozeß notwendig sind, sinnvoll. Das Entscheidungsmodell beinhaltet:

- Definition und Beschreibung der Variabilität
- Links zu den Optionen (die Features sind) im Feature Modell.
- Links zu den Variationspunkten und modellierte Lösungen in den Arbeitsergebnissen
- Beschreibung von Entscheidungsbeschränkungen
- Festlegung und Begründung der Kategorie
- Festlegung und Beschreibung der Bindungszeiten
- Die zur Disposition stehende Entscheidung
- Bedienungsanleitung für die Tätigkeiten, die nach der Entscheidung zur Bindungszeit durchzuführen sind.

3 Diskussion und Ausblick

In [EP98] und [GFA98] werden Entscheidungsmodell und Feature Modell nicht getrennt. Die in diesem Beitrag vorgestellte explizite Abtrennung eines Entscheidungsmodells aus dem Feature Modell und der

Architektur hat deutliche Vorteile für die Modellierung und Handhabung von Variabilitäten und für die Modellierung der Arbeitsergebnisse. Die Beherrschbarkeit der Variabilitäten ist Voraussetzung für die Beherrschbarkeit der Produktlinie. Hinzu kommen aber noch weitere Vorteile. Die Existenz von einem einzigen Architekturmodell, dem Produktlinienarchitekturmodell, vermeidet organisatorische Probleme, die sich durch die Existenz von mehreren Architekturmodellen (Produktlinienarchitektur und mehrere Produktarchitekturen) ergeben. Dazu gehören unter anderem die Verfolgbarkeit von Änderungen in einem der Modelle. Zudem existiert zu dieser Art von Problemen derzeit keine für Produktlinien geeignete Toolunterstützung [EP98, KCJPO00, SH00]. Sowohl für das Feature Modell als auch für die Architektur fehlen gegenwärtig noch geeignete Werkzeuge, um diejenigen Abhängigkeiten, die Änderungen in einem Modell auf die übrigen Modelle haben, handhaben zu können. Durch die Reduzierung auf ein Architekturmodell werden diese Probleme umgangen.
Die in diesem Beitrag beschriebenen Vorgehensweisen zur Behandlung von Variabilitäten werden gegenwärtig im Bereich der Architekturmodellierung eingesetzt. In einem nächsten Schritt wird versucht werden, die sich aus einer Entscheidung ergebenden Tätigkeiten zu sammeln, zu generalisieren und zu kategorisieren (Ansätze sind z.B. zu finden bei [SB00, SGB00]). Außerdem wird noch die Betrachtung von Variationspunkten im Design und der Implementierung notwendig sein.

4 Literatur

[CN96] P. Clements, L.M. Northrop
„Software Architecture: An Executive Overview“. Technical Report, Software Engineering Institute Pittsburgh, CMU/SEI-96-TR-003, 1996.

[CN99] P. Clements, L..M. Northrop
„A Framework for Software Product Line Practice“. Technical Report, Software Engineering Institute Pittsburgh, 1999.

[CW00] M. Coriat, F. Waeber
„Product Line Process Framework: The Wheels process“. ICSE 2000.

[BCK98] L. Bass, P. Clements, R. Kazman
„Software Architecture in Practice“. Addison Wesley, 1998, ISBN 0-201-19930-0.

[BBCD99] F. Bachmann, L. Bass, G. Chastek, P. Donohoe, F. Peruzzi
„The Architecture Based Design Method“. Technical Repot CMU/SEI-99-TR-XXX, Software Engineering Institute Pittsburgh, 1999.

[EP98] ESPRIT Project 28651: PRAISE
„Product-line Realisation and Assessment in Industrial Settings. IT RTD Project Programme, 1998.

[ESI98] „Product-line architectures and technologies to manage them“. Technical Repot ESI-1998,RECORD-ED1, European Software Institute, 1998.

[GFA98] M.L. Griss, J. Fararo, M. d‘Alessandro
„Integrating Feature Modeling with the RSEB“. 5‘th International Conference on Software Reuse (ICSE5), 1998

[K95] P. Kruchten
„Architectural Blueprints – The 4+1 View Model of Software Architecture“. IEEE Software 12(6), 1995.

[KCHNP90] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson
„Feature-Oriented Domain Analysis (FODA) Feasibility Study“. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute Pittsburgh, 1990.

[KCJPO00] W.EL Kaim, S. Cherki, P. Josset, F. Paris, J.-C. Ollagnon
„Applied technology for designing a PL architecture of a pilot training system“. ICSE 2000

[KM99] B. Keepence, M. Mannion
„Using Patterns to Model Variability in Product Families“. IEEE Software, 1999

[SB00] M. Svahnberg, J. Bosch
„Issues Concerning Variability in Software Product Lines“. Technical Report, University of Karlskrona/Ronneby, 1999.

[SGB00] M. Svahnberg, J. van Gurp, J. Bosch
„On the Notion of Variability in Software Product Lines“. Technical Report, University of Karlskrona/Ronneby, 2000.

[SH00] M. Schlick, A. Hein
„Knowledge Engineering in Software Product Lines“. European Conference on Artificial Intelligence (ECAI 2000), 2000.

[W00] D. M. Weiss
„Defining Families: The Commonality Analysis“. Technical Report, Lucent Technologies Bell Laboratories, 2000.

Erfahrungen bei der objektorientierten Modellierung von Produktlinien mit FeatuRSEB

Kai Böllert Ilka Philippow

Technische Universität Ilmenau
Postfach 100565
98684 Ilmenau
{kai.boellert,ilka.philippow}@theoinf.tu-ilmenau.de

Zusammenfassung

Software-Systeme mit ähnlicher Funktionalität lassen sich in einer Produktlinie zusammenfassen und so besonders wirtschaftlich entwickeln. Eine Methode zur objektorientierten Entwicklung von Produktlinien ist FeatuRSEB. Dieser Beitrag bewertet diese Methode auf Basis von Erfahrungen, die in einem Projekt zur Modellierung einer Produktlinie von Bibliothekssystemen gesammelt wurden.

1 Einführung

Die heutige Softwareentwicklung bringt zwei Arten von Software hervor: Individualsoftware entsteht in Einzelfertigung und ist maßgeschneidert auf die Anforderungen genau eines Kunden. Standardsoftware hingegen deckt die (vermeintlichen) Anforderungen vieler Kunden gleichzeitig ab. Einmal entwickelt läuft die Produktion von Standardsoftware als Massenfertigung durch Vervielfältigung der Datenträger ab. Die Konsequenzen: Individualsoftware verursacht hohe Kosten durch lange Entwicklungszeiten. Standardsoftware ist preiswerter, entspricht aber selten genau den Anforderungen der Kunden.

Als Lösungsweg bietet sich die Serienfertigung von Software-Systemen an: Ausgehend von den Kunden-Anforderungen setzt ein Generator automatisch passende Systeme aus vorgefertigten Komponenten zusammen [4]. Die Vorteile dieses Verfahrens sind u.a.: kurze Produktionszeiten, Systeme sind preiswerter als Individualsoftware, im Vergleich zu Standardsoftware stimmen Kunden-Anforderungen und Funktionalität der Systeme besser überein, dadurch sinkender Einarbeitungsaufwand beim Anwender sowie geringerer Ressourcenbedarf der Systeme.

Den wirtschaftlichen und organisatorischen Rahmen für eine Serienfertigung bildet eine Produktlinie. Eine Produktlinie beschreibt eine „Gruppe von Produkten, die in enger Beziehung zueinander stehen, da sie eine ähnliche Funktion erfüllen“ [10, S. 680]. Investitionen in eine *Software*-Produktlinie amortisieren sich am schnellsten, wenn die Produktlinie als Systemfamilie gebaut wird, d.h. auf einem „common set of core assets“ basiert [3, S. 3].

Eine wichtige Aufgabe bei der Entwicklung von Systemfamilien ist der Entwurf des Systemfamilien-Modells. Dieser Beitrag berichtet über ein laufendes Projekt, in dem eine Familie von Bibliothekssystemen objektorientiert nach der Methode FeatuRSEB [6] modelliert wird. Der Beitrag gliedert sich wie folgt: Abschnitt 2 stellt das Projekt kurz vor. Abschnitt 3 beschreibt die wesentlichen Modelle von FeatuRSEB und illustriert ihre Verwendung im Projekt. Schließlich bewertet Abschnitt 4 die FeatuRSEB-Methode auf Basis der bisher im Projekt gewonnenen Erfahrungen.

2 Das Projekt

Ausgangspunkt für das Systemfamilien-Projekt war ein Software-System für eine Hochschulbibliothek, das im Rahmen einer Lehrveranstaltung entwickelt worden ist. Für die Modellierung nutzten die Studenten die UML (Unified Modeling Language): u.a. entstand ein Use-Case-Modell, verschiede-

ne Aktivitätsdiagramme und ein Objektmodell. Die Implementierung erfolgte mit Java.

Als potentielle Kunden für ein Bibliothekssystem kommen jedoch nicht nur Hochschulbibliotheken in Frage sondern auch Stadtbüchereien und ihre ländlichen Ableger, die Fahrbüchereien. Die prinzipielle Anforderung nach einer Aufzeichnung des Leihverkehrs ist bei allen drei Kundentypen gegeben. Aber es gibt auch Unterschiede in den Anforderungen: Eine Hochschulbibliothek möchte Benutzerkategorien wie Student und Hochschulmitarbeiter einrichten und an diese bestimmte Ausleihbedingungen knüpfen. Eine Fahrbücherei wiederum ist am täglichen Abgleich mit den Stamm- und Bewegungsdaten der Zentrale interessiert.

Um nicht viele getrennte Bibliothekssysteme entwickeln und warten zu müssen, wurde beschlossen, das vorhandene System zu einer Familie auszubauen, d.h. einen gemeinsamen, für alle Familienmitglieder gleichen Kern zu bilden und um diesen die Unterschiede zwischen den Systemen (sogenannte *Variabilität*) zu gruppieren. Der Aufbau der Systemfamilie findet in mehreren Evolutionsschritten statt [11].

3 FeatuRSEB

Als Entwicklungsmethode kommt im Systemfamilien-Projekt FeatuRSEB zum Einsatz. FeatuRSEB [6] ist der Nachfolger des Reuse-Driven Software Engineering Business (RSEB) [8]. RSEB ist eine an die OOSE-Methode [7] angelehnte Vorgehensweise zur Entwicklung von Systemfamilien. FeatuRSEB integriert Konzepte aus der Feature-Oriented Domain Analysis (FODA) [9] in RSEB.

Eine weitere Methode, die für das Projekt in Betracht gezogen wurde, ist KobrA [1], eine Instanz der Product-Line Software Engineering Methodology (PuLSE) [2]. PuLSE besteht aus einem Vorgehensmodell und einer Reihe abstrakter technischer Komponenten zur Entwicklung von Produktlinien. KobrA übernimmt von PuLSE das Vorgehensmodell und verfeinert die abstrakten Komponenten, so daß Produktlinien mit objektorientierten Techniken entwickelt werden können.

Die Entscheidung fiel auf FeatuRSEB, weil der zum Projektstart verfügbare Dokumentationsumfang bei KobrA sehr viel geringer ausfiel als bei FeatuRSEB. Ein weiterer Pluspunkt für FeatuRSEB war dessen Abstammung von der OOSE-Methode, die viele Projektmitglieder bereits kannten, so daß sich die Einarbeitungszeit verkürzte.

3.1 Überblick

FeatuRSEB beinhaltet:

1. Beschreibung der Entwicklungsprozesse
2. Methode zum Finden von Variabilität
3. Notation zur Modellierung von Variabilität

Aus Platzgründen beschreibt der Rest dieses Abschnitts nur den Modellierungsaspekt von FeatuRSEB näher. Zur Illustration dienen vereinfachte Beispiele aus dem Systemfamilien-Projekt.

3.2 Merkmalsmodell

Das zentrale Modell in FeatuRSEB ist das aus FODA übernommene Merkmalsmodell, das gemeinsame und unterschiedliche Merkmale (Funktionalität, technische Eigenschaften) aller Mitglieder der Systemfamilie spezifiziert. Merkmale werden hierarchisch in einem Baum organisiert und wie folgt klassifiziert:

- *notwendig:* alle Familienmitglieder besitzen das Merkmal, sofern sie das übergeordnete Merkmal ebenfalls besitzen
- *optional:* nur bestimmte Mitglieder bieten das Merkmal an
- *alternativ:* jedes Mitglied wählt eine der gegebenen Ausprägungen des Merkmals

Desweiteren legt das Merkmalsmodell Regeln fest, welche Merkmale in einem System nicht zusammen auftreten dürfen und welche Merkmale noch zusätzlich die Auswahl weiterer Merkmale erfordern.

Abbildung 1 zeigt einen Ausschnitt des Merkmalsmodells der Bibliotheks-Systemfamilie. Die gemeinsamen Merkmale aller in der Familie modellierten Systeme sind demnach: Leihverkehr (Ausleihe, Rückgabe, Verlängerung, Vorbestellung), Benutzer- und Medienverwaltung sowie die Identifikation der Benutzer im Leihverkehr. Wie sich die Benutzer identifizieren, ob per Chipkarte oder Biometrie, ist von System zu System verschieden.

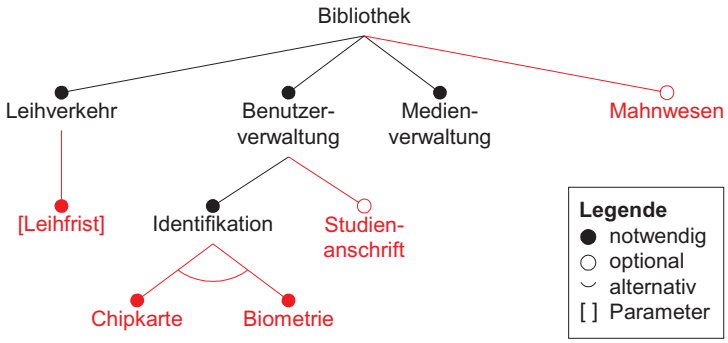


Abbildung 1: Merkmalsmodell

Weitere Unterschiede zwischen den Bibliothekssystemen sind: Leihfrist (z.B. zwei oder vier Wochen), Speicherung der Studienanschrift von Benutzern sowie ein automatisches Mahnwesen.

Das Merkmalsmodell bietet eine wiederverwendungsorientierte Sicht auf die Systemfamilie: Welche Funktionalität steht zur Verfügung und welche Kombinationen sind möglich? Wie die einzelnen Funktionen bzw. Anwendungsfälle ablaufen, spezifiziert das Use-Case-Modell.

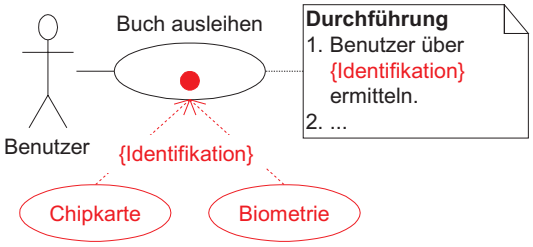


Abbildung 2: Use-Case-Modell

3.3 Use-Case-Modell

Das Use-Case-Modell von FeatuRSEB dokumentiert die Anwendungsfälle und Akteure aller Mitglieder der Systemfamilie. Unterschiede zwischen den Familienmitgliedern, die im Merkmalsmodell als optionale und alternative Merkmale modelliert sind, zeigen sich im Use-Case-Modell durch:

- Vererbungsbeziehungen zwischen Akteuren oder Use-Cases
- «extend»-Beziehungen zwischen Use-Cases

Ein Beispiel für den letztgenannten Fall zeigt Abbildung 2. Aus dem notwendigen (bei allen Bibliothekssystemen der Familie vorhandenen) Merkmal Leihverkehr ist der Use-Case „Buch ausleihen“ hervorgegangen. Bevor Benutzer Bücher ausleihen können, müssen sie sich identifizieren. Wie dies geschehen soll, hängt von der für das jeweilige System gewählten Ausprägung des alternativen Merkmals Identifikation ab: Chipkarte oder Biometrie. Die beiden möglichen Abläufe beschreiben zwei weitere Use-Cases.

Um jetzt im Modell die Verbindung zwischen der Ausleihe und den Varianten zur Benutzeridentifikation herzustellen, definiert der Use-Case „Buch ausleihen“ den Variationspunkt {Identifikation} und delegiert an diesen die Durchführung der Identifikation. Die Use-Cases zur Identifikation hängen sich in diesen Variationspunkt ein. Im Diagramm werden Use-Cases mit Variationspunkten zusätzlich mit einem gefüllten Kreis dargestellt.

3.4 Objektmodell

Variationspunkte und Varianten spiegeln sich auch im Objektmodell und in der Implementierung wider. Hier setzt FeatuRSEB auf bekannte Techniken:

- abstrakte Klassen
- Templates
- Entwurfsmuster

Abbildung 3 demonstriert die Anwendung des Entwurfsmusters Strategie [5, S. 315ff] auf die Identifikation von Benutzern im Leihverkehr.

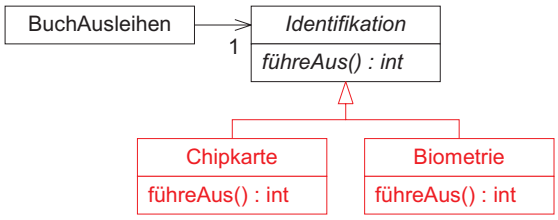


Abbildung 3: Objektmodell

3.5 Verknüpfung der Modelle

Die vorgestellten Modelle von FeatuRSEB sind über «trace»-Beziehungen miteinander verknüpft. Eine solche Beziehung besteht beispielsweise zwischen dem Merkmal Chipkarte, dem gleichnamigen Use-Case und der konkreten Strategie-Klasse.

Der Einsatz von «trace»-Beziehungen erleichtert nicht nur Entwurf und Wartung des Systemfamilien-Modells, sondern ist auch Voraussetzung für die erfolgreiche Produkt-Fertigung, die Generierung der Familienmitglieder. Der erste Schritt im Fertigungsprozeß ist die Zusammenstellung der Produktkonfiguration durch den Kunden. Dazu wählt der Kunde aus der Liste der angebotenen Merkmale die gewünschte Funktionalität aus. Der Generator verarbeitet diese Konfiguration und setzt das Produkt aus denjenigen Klassen zusammen, die er ausgehend von den gewählten Merkmalen über die «trace»-Beziehungen ermittelt hat. Dadurch ist gewährleistet, daß das Produkt nur die wirklich benötigte Funktionalität implementiert.

4 Bewertung von FeatuRSEB

FeatuRSEB hat sich im Systemfamilien-Projekt bisher nur bedingt bewährt. Während die vorgeschlagenen Entwicklungsprozesse sich als praktikabel erweisen, stellte sich das methodische Vorgehen zum Finden von Variabilität als zu vage und unkonkret heraus. Vermißt wurde insbesondere eine Zusammenstellung sogenannter *Best Practices*.

Hauptkritikpunkt an FeatuRSEB ist jedoch der in Abschnitt 3 beschriebene Modellierungsaspekt, der viele Schwächen und Lücken aufweist. Einzig die Integration des Merkmalsmodell ist als gut zu bewerten – sieht man einmal davon ab, daß keines der gängigen CASE-Werkzeuge hierfür Unterstützung anbietet. Die folgenden Absätze gehen detaillierter auf einige der gefundenen Schwächen und Lücken ein.

Vorkehrungen für die Produkt-Fertigung. Der Generator fertigt ein Produkt, indem er nur die Klassen in das Produkt aufnimmt, die über «trace»-Beziehungen mit den vom Kunden gewählten Merkmalen verbunden sind. In der Praxis treten häufig Fälle auf, in denen Merkmale am Ende der Verweiskette nicht auf vollständig neue Klassen verweisen. Oft fügen sie lediglich weitere Attribute, Operationen oder Assoziationen zu bestehenden Klassen hinzu. Ein Beispiel ist das optionale Merkmal Studienanschrift, das die Klasse Benutzer um ein entsprechendes Attribut erweitert.

Das Problem liegt darin, daß «trace»-Beziehungen nur auf Klassen als Ganzes verweisen, aber keine Erweiterungssemantik besitzen. Im Beispiel beziehen sich die Merkmale Benutzerverwaltung und Studienanschrift immer auf die gesamte Klasse Benutzer, inklusive dem Attribut Studienanschrift. Für die Produkt-Fertigung bedeutet dies: Wählt der Kunde das Merkmal Studienanschrift *nicht* aus, speichern Benutzer-Objekte im Produkt trotzdem die Studienanschrift, da der Generator nur anhand der «trace»-Beziehungen das Attribut nicht aus der Klasse entfernen konnte. Einer der eingangs genannten Vorteile der Serienfertigung von Software-Systemen, der geringere Ressourcenbedarf der Systeme, tritt somit nicht in vollem Umfang ein.

Ein möglicher Ausweg ist der vermehrte Einsatz von Entwurfsmustern. Der dadurch ausgelöste Anstieg der Komplexität des Objektmodells wirkt sich aber wiederum nachteilig auf die Wartbarkeit der Systemfamilie aus [4, S. 292ff].

Modellierung variablen Verhaltens. Der grundlegende Programmablauf aller Systemfamiliemitglieder ist durch den gemeinsamen Kern vorgegeben. Optionale und alternative Merkmale können den Ablauf erweitern oder abändern. Die Modellierung variablen Verhaltens geschieht im Use-Case-Modell über Variationspunkte, die den

variablen Teil der Durchführung an Use-Case-Varianten delegieren. Um variables Verhalten auf Ebene der Objekte umzusetzen, sind Aktivitäts- und Sequenzdiagramme zu erstellen. Wie Variabilität in diesen beiden Diagrammen zu modellieren ist, darüber trifft FeatuRSEB keine Aussage.

Dynamische Aktivierung von Merkmalen. Merkmale können zu verschiedenen Zeitpunkten in einem Produkt aktiviert (ausgewählt, zugeschaltet) werden [4, S. 100ff]:

- *statisch:* vor der Fertigung
- *dynamisch:* beim Start des Systems oder während seiner Laufzeit

FeatuRSEB berücksichtigt nur die statische Auswahl von Merkmalen vor der Fertigung des Produktes. Zur Modellierung (und Implementierung) dynamisch aktivierbarer Merkmale finden sich keine Angaben.

5 Schlußfolgerungen und Ausblick

FeatuRSEB ist die zur Zeit am besten dokumentierte, öffentlich verfügbare Methode zur objektorientierten Entwicklung von Systemfamilien. Sie wird deshalb in einem Projekt zum Aufbau einer Familie von Bibliothekssystemen eingesetzt. Die bisherigen Erfahrungen mit der Methode im Bereich Modellierung sind ernüchternd: Die Modelle unterstützen die spätere Fertigung von Systemen nur unzureichend, eine Modellierung variablen Objektverhaltens ist genauso wenig vorgesehen wie die dynamische Aktivierung von Funktionalität zum Start oder während der Laufzeit der Systeme.

Ein zukünftiger Schwerpunkt der Arbeiten im Projekt ist folglich die Verbesserung der FeatuRSEB-Methode. Eine Alternative wäre der Einsatz einer anderen Methode, z.B. KobrA, sobald sich hier die Dokumentationssituation bessert.

Literatur

[1] Colin Atkinson, Joachim Bayer und Dirk Muthig. Component-Based Product Line Development: The KobrA Approach. In *Proceedings of the 1st Software Product Line Conference* (2000).

[2] Joachim Bayer et al. PuLSE: A Methodology to Develop Software Product Lines. In *Proceedings of the 5th Symposium on Software Reusability*, Seiten 122–131 (1999).

[3] Paul Clements und Linda Northrop. A Framework for Software Product Line Practice, Version 2.7 (Juli 1999).

[4] Krzysztof Czarnecki und Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley (2000).

[5] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley (1995).

[6] Martin L. Griss, John Favaro und Massimo d'Alessandro. Integrating Feature Modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse*, Seiten 76–85. IEEE Computer Society Press (1998).

[7] Ivar Jacobson et al. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley (1992).

[8] Ivar Jacobson, Martin Griss und Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley (1997).

[9] Kyo Kang et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Bericht CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (November 1990).

[10] Philip Kotler und Friedhelm Bliemel. *Marketing-Management: Analyse, Planung, Umsetzung und Steuerung*. Schäffer-Poeschel, 9. Auflage (1999).

[11] Matthias Riebisch und Bogdan Franczyk. Evolutionary Development of Frameworks – from Projects to System Families. In *Proceedings of the Conference on Design and Process Technology (IDPT '99)* (Juni 2000).

Feature Modeling Using Design Spaces*

Lars Geyer
System Software Research Group
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
geyer@informatik.uni-kl.de

Abstract

In the context of software product-lines the configuration knowledge describing how requirements are mapped onto elements of the reuse framework is essential for the success of the product-line. The starting point of this mapping process can be a feature model which describes the functionality to be found in the domain on an abstract level. In this paper we present an approach for the description of feature models which offers a tool-supported selection of features and a connection of this information to the further steps of application development.

1 Introduction

The highly competitive and dynamic field of software development implies the need to continuously increase the efficiency of development processes. Software reuse and especially the principle of product-lines are commonly recognized approaches to address this problem. The product-line process can thereby be seen as a two-staged process. In the first stage during domain engineering the product-line is initialized by the definition of the scope of the domain, i.e. by defining the real world aspects which are part of the product-line. In a later step a reuse framework is built which comprises requirements models, and a reference architecture as the basic structure for all systems in the product-line, as well as components which are reused during the implementation of applications in the domain.

The second stage contains the actual application engineering process which is performed for every application of the product-line. In this process the reuse framework built during domain engineering is used as a skeleton that is extended to the needs of a specific application. E.g., the requirements models developed during the domain analysis are extended by specific requirements in order to complete the specification. This specification is later used to instantiate the given reference architecture.

An important part of the requirements models is a description of the feature model. The feature model is an abstract representation of functionality found in the domain. Each feature thereby is a relevant characteristic of the domain. The feature model is used during domain engineering in order to obtain an abstract view on this functionality which can be verified against the needs raised by the domain. The primary purpose of feature modeling, however, is the configuration of the product-line during the application engineering process. Since the reusable infrastructure already exists, it is possible to describe which parts of the reference architecture implement a single feature. With this knowledge in mind it is possible to trace decisions on the level of the feature model down to relevant components of the architecture. In contrast, it is also possible to exclude parts of the architecture based on the decision not to include certain features.

This paper describes an approach for the formalization of feature models which is based on the design space technique. This technique is well suited for the tool supported selection of features as well as the deployment of the feature selection in further steps of the development process. The paper is structured as follows. The next section gives a short overview of feature modeling as it is currently seen in the research community. Section 3 contains a description of the design space technique whose deployment for feature modeling in the context of our software development process is explained in section 4. In section 5, short usage scenarios are given which describe the further application of the feature selection during the configuration of the product-line. Finally, section 6 gives some concluding remarks and an outlook on further work.

2 Feature Modeling

The description of feature models goes back to the late eighties when, e.g., the FODA [6] approach for domain analysis was introduced. In FODA, features are typically arranged in an hierarchical structure that spans a tree. Fig. 1 gives a small example of a feature model of the home automation domain in the notation introduced by the FODA approach

The root node of a feature tree always represents the domain whose features are modeled. The remaining nodes represent features which are classified into three types:

- Mandatory features are always part of the system if their parent feature is part of the system. E.g. the *heating unit* feature in fig. 1 is a mandatory feature which is indicated by a solid circle on the edge leading to the feature.
- Optional features may be part of the system if their parent feature is already in the system. The decision whether an optional feature is part of the system or not can be made independently from the selection of other features. E.g., the *temperature sensor* feature in fig. 1 is optional which is indicated by the transparent circle at the edge leading to the feature.
- Alternative features are connected via an *exclusive or* relationship, i.e. exactly one feature out of a set is part of the system if the parent feature is part of the system. The two features *convector* and *radiator* in fig. 1 form a typical alternative feature set which is indicated by an arc connecting the edges leading to the alternative features.

*This research was supported by the Deutsche Forschungsgemeinschaft as part of the Special Research Project 501.

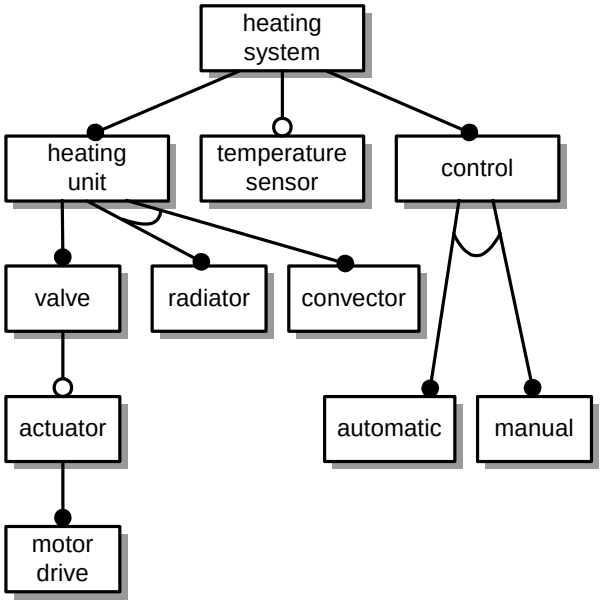


Figure 1. An example feature model of a heating system in a home automation domain.

In addition, Features in a domain are typically distinguished between common and variable. Common features are always part of a system in the regarded domain. Variable features are only part of some systems. The classification of a feature is determined by its type, and by its position in the feature tree. Common features are always mandatory. Another prerequisite is that there are only mandatory features in the path from the root node to the common feature. E.g., the *valve* feature is a common feature in fig. 1, while the *motor drive* feature is not. Optional and alternative features are always variable.

The feature tree is the basic description of a feature model. It defines a hierarchical structure over the set of features of a domain, thereby defining the parent-child relationship between different features. But typically there are more relationships between features. One relationship is called *Or-Features* by Czarnecki and Eisenecker [5]. This relationship connects a set of optional features with a common parent feature. The meaning of the relationship is that whenever the parent feature is part of a system, at least one of the optional features in the set has to be part of the system. Czarnecki and Eisenecker extended the FODA notation so that this relationship can be expressed in the feature tree.

Other types of relationships which cannot be expressed with the feature tree notation are the *requires* and the *excludes* relationships. The requires relationship connects two variable features expressing that if one of the features is chosen to be part of the system, the other feature has to be chosen, too. In the feature model in fig. 1, e.g., if the *automatic control* feature is chosen, then the *temperature sensor* feature and the *actuator* feature have to be chosen as well. The excludes relationship on the other side states that only one out of a set of features can be part of the system.

These relationships and several other types like default features or feature combination recommendations cannot be expressed in the tree notation. Typically they have to be defined in an external representation.

Feature models are used for several purposes during the product-line life-cycle. In the domain engineering process, the feature model is an abstract description of the features which are part of the domain. The abstract view allows a verification of the functionality which helps to avoid two serious problems [5]:

- Relevant features are not included in the reusable software framework
- Features which do not offer enough reuse potential are defined to be part of the domain.

Besides these effects, the feature model is the starting point for the traceability information within the reuse framework of a product-line. During domain engineering, the information how features are implemented by the product-line reference architecture has to be collected and described in a way that can be reused during application engineering. The feature model thereby is the starting point of the configuration. During application engineering the feature selection is mapped onto the components of the reference architecture which implement the needed functionality, allowing a instantiation of the reference architecture.

In order to represent the feature model in an operational way and support the ensuing configuration steps we use the design space technique. In the next section we give a short overview of the design space technique. Afterwards, we explain how the technique is used for the description of feature models.

3 Design Spaces

The design space technique was originally presented in [7][8] as a semi-formal way to assess user-interface requirements and to study their effects on the choice of implementation architectures. However, our extended notion of design spaces has shown a general suitability of the representation of decisions in the area of requirements as well as of design choices [2][1].

A *design space* (DS) is a multidimensional space representing requirements and design choices. It is spanned by a set of *dimensions* identifying relevant criteria for characterizing artifacts in a specific domain – components, subsystems, or complete systems. In the home automation domain, e.g., such criteria might relate to functional aspects such as the type of a heating unit, or to non-functional aspects such as the timing behavior of specific services. Design spaces may comprise two types of dimensions. Discrete dimensions enumerate possible alternatives, so-called *categories*. Categories of the heating unit type dimension, for instance, are the available unit types such as radiators, convectors, in-floor heating units and so on. To enrich the expressiveness of the design space concept, we added continuous dimensions that represent either integer or real values. With these dimensions, it is, e.g., possible to describe the amount of water which has to be heated up in order to fill the heating units of a building.

A design space represents a taxonomy for variable aspects of systems in a domain. A selection in the design space – a so-called *design space profile* – characterizes a concrete system along this taxonomy. The process of the selection of dimensions is called *profiling*.

Beyond the expressiveness of faceted classification schemes [10][9][12], design spaces allow to represent favorable and unfavorable combinations of design choices. To this end, *correlations* between categories or between categories and dimensions can be specified. A correlation is a relationship between two sets of categories or dimensions, as illustrated in figure 2. Both sets of a correlation are combined by either a boolean term or by a functional expression. We distinguish between *weak* and

hard correlations: In weak correlations, the relationship between the two sets is expressed by a weighted factor on a continuous scale from -1 (choices should not be combined) to +1 (categories should favorably be chosen in combination). In hard correlations, the relationship is expressed by a boolean value representing a positively or negatively dependent decision; the assignment of one set of the correlation implies an assignment of the affected dimensions of the other set of the correlation. In the case of positively dependent decisions, the terms on both sides have to be true; for negatively dependent decisions, only one of the terms may be true.

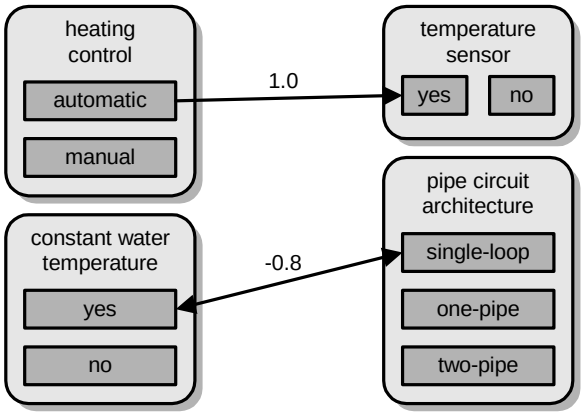


Figure 2. Excerpt from a Design Space describing home automation systems.

Based upon our initial experiences, we have made some further extensions in order to ease the handling of design spaces. An important mechanism we introduced to the concept are *groups* which allow to group related dimensions together. Since groups may be members of other groups as well, this mechanism allows the hierarchical structuring of a design space. Another mechanism are the so-called *hierarchies* which allow selections in one dimension to hide or to disclose other dimensions. Finally, *sequences* express a preferred order in which selections in a design space should be made.

To take full advantage of the design space technique, tool support is indispensable. On the one hand, the considerable size of design spaces (40-120 dimensions in design spaces recently developed) calls for tools to provide context sensitive views on the profiles and to guide the developer along sequences of dimensions. On the other hand, correlations can favorably be evaluated by tools, thereby laying the basis for an automated processing of the represented knowledge.

To this end, we have developed two interactive tools. The so-called *Designer* supports the development of design spaces in terms of dimensions, categories and correlations. Regarding correlations, the full expressiveness of the extended design spaces concept is provided to ease the efficient description of dependencies. The concepts of groups, hierarchies and sequences can further be applied to structure the information included in an design space in an intuitive manner.

While the Designer is rather used in the domain engineering phase, the so-called *Configurator* is intended to support the interactive creation of design space profiles and to automatically evaluate correlations. The Configurator uses the structuring mechanisms like subspaces, groups, or hierarchies to present the relevant parts of the design space in dependence of the current profile and the current state of the development process. It furthermore provides the possibility to automatically select values in dimensions by evaluating correlations.

4 Feature modeling using design spaces

In this section we describe how the design space technique can be applied for feature modeling. As we will show in this section, design spaces offer advantages especially for the description of relationships between features.

As mentioned in section 2 a feature model comprises mandatory, optional, and alternative features. For each of these types there is a representation in the design space technique. Fig. 3 shows the example of section 2 in a graphical representation of the design space model.

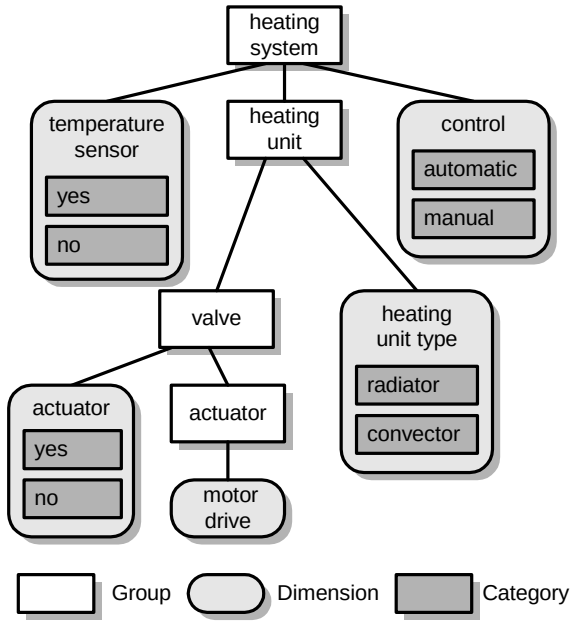


Figure 3. The feature model of a heating system in the home automation domain described in a graphical representation of the design space model.

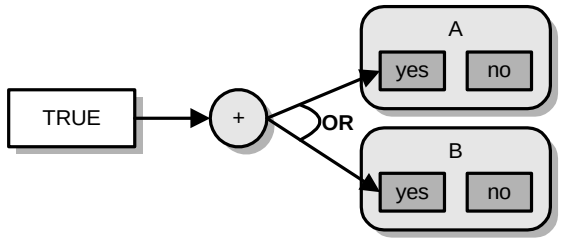
Mandatory features are not relevant for decision making. A mandatory feature is always part of the system if the parent feature is selected for the application. As a consequence it does not make sense to describe a mandatory feature as a dimension in the feature design space. Instead, we use the group concept to emphasize the structural meaning of mandatory features. E.g. in fig. 3, the *heating unit* feature is modeled by a group in the design space. It contains the features *valve* and *heating unit type*. A special case are mandatory features which are leafs in the feature tree. This type of features only has an informational character in order to make the description of the feature model complete. They are irrelevant during the configuration process since the decision whether they are part of a system or not is made elsewhere. So if the feature model is only used for configuration purposes, they can be left out of the model. If completeness of the model is desired the features can be described by an empty dimension. In fig. 3 the *motor drive* feature is an example for this special case.

Alternative features can be distinctly mapped to a dimension of the feature design space. The overall concept is modeled as the dimension, and the alternative features are modeled as categories within this dimension. In comparison to the feature tree notation described in section 2, it may be necessary to insert a supplementary node in order to define a node in the tree which only has the alternative features as child nodes. E.g., the *heating unit type* dimension in fig. 3 has to be inserted into the model because the *heating unit* feature has — besides the heating unit types — a child feature concerning the *valve* of the

heating unit. In other cases like the *control* feature, no extra node is necessary because the child features are all part of the alternative feature set.

Optional features can be selected to be part of the system independently from other features. This type of feature is modeled by a dimension with two categories, *yes* and *no*. Examples for this type are the *actuator* and the *temperature sensor* features in fig. 3. The *actuator* feature shows an special case in the modeling. Along with the optional decision whether to include a feature in a system or not there are other features associated which are only relevant if the optional feature is part of the system. This is modeled by an normal optional feature, e.g., *actuator*, and a group which contains all dependant features, e.g., group *actuator* in figure 3.

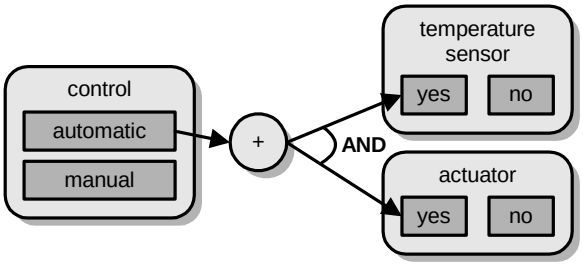
As described in section 2 there are various types of relationships between features which cannot be expressed in the tree representation. The or-features can be modeled in two different ways in the design space concept. The first possibility is the definition of a correlation. This correlation would only be fulfilled if one of the optional features is chosen to *yes*. E.g. if there are two dimensions A and B which represent optional features with a common parent feature and these two features have the or-feature relationship, a correlation would be defined with the correlation term:



This correlation expresses that there has always (TRUE) to be chosen *yes* either in dimension A or in B. A tool can ensure that this condition is always held during the profiling of the feature design space.

The second possibility to model an or-feature relationship to represent the whole feature set as one alternative dimension. Since the design space concept allows to select a set of categories as the profile of one dimension, all consistent selections of the relevant features could be selected with this dimension.

The requires and the excludes relationships are modeled using hard correlations. An advantage of the design space concept is the possibility to combine several dimensions with the expressiveness of boolean expressions. In this way complex relationships between a whole set of features can be described. E.g., the requires relationship between the *automatic control* feature, the *temperature sensor*, and the *actuator* feature could be expressed in one correlation:



This correlation expresses that whenever *automatic* is chosen in the *control* dimension, the category *yes* has to be selected in the *actuator* as well as the *temperature sensor* dimension. This consistency constraint can also be ensured by a tool which supports the profiling process.

The design space concept allows to express other types of relationships as well. Feature combination recommendations, for example, can be modeled by weak correlations. They describe hints used by a developer as guidance for the selection of several features. Unlike the requires relationship, these recommendations only characterize combinations which are favorable but not necessary. Default features can be defined quite easily by using a default profile as starting point for the feature selection process. This default profile contains a basic set of selections for the design space which is extended during the profiling.

All of the mentioned relationship types can thus be expressed within the design space technique. This reliefs the designer of a feature model from the error-prone checks for consistency between several techniques used for a complete description of a feature model. Another advantage of the design space concept is the possibility for dimensions and groups to be member of more than one group. This offers the possibility to disrupt the bonds of the tree-structure allowing features to turn up in several places of the feature model. This can be used, e.g., during the profiling of the feature design space in order to present features in all places where their selection may have influence on the selection of other features.

The structuring concepts of the design space technique can be favorably deployed during the development of a consistent feature selection. The *hierarchies* are used to mask out parts of the design space which are not relevant for a special system. E.g. if an optional feature is chosen not to be part of the system, all child features of this feature do not have to be regarded. In figure 3, the *actuator* dimension and group describe this type of situation. In the design space a hierarchy is defined which masks out the *actuator* group until *yes* is chosen in the *actuator* dimension. The concept of *sequences* can be used to define configuration strategies, i.e. it is possible to define a favorable path through the feature model. This technique allows to ease the decision making. For example, in the feature model in figure 3 it is favorable to select the *control* feature before the *temperature sensor* and the *actuator* feature because of the requires relationship between these features. If *automatic control* is chosen, the selection of the other two features can be made automatically, as stated in the respective correlation presented above. Therefore it is advantageous to define a sequence which recommends a selection of the features in the described order.

5 Usage of DS profiles

The feature selection for a specific application is essential in the further steps of system development. So there have to be ways in which the configuration data can be processed in the further development steps. We currently use two scenarios for the processing of the profiles of feature design spaces.

The first scenario is described in [3]. In this scenario the feature design space represents only one level of an overall development design space. The information gathered in the feature design space is broken down onto design space profiles for specific component types describing the requirements for this component type. In the opposite, components which implement a specific type are described with a design space profile specifying the provided features of the component. The requirements profile is afterwards compared to the provided profiles of

the components of a component pool allowing the selection of components which meet the requirements of a specific application.

The second scenario comprises the completion of templates used for the documentation or implementation of product-line-based applications. Special filters can be attached to a design space profile which transform the information in the profile into appropriate fragments which complete these templates. E.g., we have realized a configuration design space for the QNX operation system [11]. In [4] we describe the configuration process which uses a filter that extracts the configuration information out of a design space profile and transforms it into a configuration file for the QNX operating system.

Currently we are working on an XML filter for design spaces and design space profiles. This XML filter will serve as a basis for the configuration of UML diagrams with the information gathered in a design space.

6 Conclusions and further work

In this paper, we have presented an approach for the representation of feature models using the design space technique. We have shown how traditional feature trees can be transformed into a design space description. One important advantage of the design space technique in this area is the tool supported feature selection during the application engineering process. It is possible to present the features to a developer in a structured way, while simultaneously ensuring the consistency of the feature selection by automatic evaluation of the correlations. Other advantages are the possible description of all relationships between features within one technique, and the possibility to describe the feature model in a more complex structure as compared to a tree. The information gathered during feature selection can be processed at least semi-automatically in further steps of the application engineering process. We described two brief scenarios which illustrated the possible approaches.

As experienced in various projects, one problem of larger feature models is their complexity. The design space technique offers possibilities to handle this complexity. We are currently examining strategies in which the structuring concepts of the design space technique are deployed best. In order to implement these strategies in a design space the tool support has to be enhanced. We are currently integrating the experiences made in this area in recent modeling projects into our tools.

In addition, we are working on possible extensions to the design space concept such as substructure templates which allow to easily replicate parts of a feature design space in several places. Other extensions would be a tabular representation of parts of a feature model, and vice versa the automatic generation of parts of a feature model out of a tabular representation.

Another aspect in our further work is the evaluation of the mapping between design space profiles and other description techniques. As mentioned above, we are currently developing a filter which allows the completion of UML diagram templates. Additionally we are planning to further integrate other description techniques as needed in our research.

7 References

[1] Baum, L.: *Towards Generating Customized Run-time Platforms from Generic Components*, Proc. of the 11th Conference on Advanced Information Systems Engineering (CAISE'99), 6th DC, Heidelberg, Germany, June 1999

[2] Baum, L.; Geyer, L.; Molter, G.; Rothkugel, S.; Sturm, P.: *Architecture-centric Software Development Based on Extended Design Spaces*, Proc. of the 2nd ARES Workshop (Esprit 20477), Las Palmas de Gran Canaria, February 1998

[3] Baum, L.; Becker, M.; Geyer, L.; Molter, G.: *Mapping Requirements to Reusable Components using Design Spaces*, Proc. of the IEEE Int'l Conference on Requirements Engineering (ICRE-2000), Schaumburg/Chicago, USA, June 19-23 2000

[4] Baum, L.; Becker, M.; Geyer, L.; Gilbert, A.; Molter, G.; Tamara, V.: *Supporting Component-Based Software Development Using Domain Knowledge*, Proc. of the SCI 2000 Conference, Orlando, Florida, USA, July, 2000

[5] Czarnecki, K.; Eisenecker, U.: *Generative Programming - Methods, Tools, and Applications*, Addison-Wesley, 2000

[6] Kang, K.; Cohen, S.; Hess, J.; Nowak, W.; Peterson, S.: *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, November 1990

[7] Lane, T. G.: *Guidance for User-Interface Architectures*, in: Garlan. D., Shaw, M.: Software Architecture – Perspectives on an Emerging Discipline, Prentice Hall, 1996

[8] Lane, T.G.: *Studying Software Architecture Through Design Spaces and Rules*, Technical Report CMU/SEI-90-TR-18, Carnegie Mellon Univ., 1990

[9] Ossher, H.; Tarr, P.: *Multi-Dimensional Separation of Concerns using Hyperspaces*, IBM Research Report 21452, April 1999

[10] Prieto-Diaz, R.: *Classifying Software for Reusability*, IEEE Software, January 1987

[11] QNX Software Systems Ltd.: *QNX System Architecture*, System Documentation, QNX Software Systems, Canada, 1993

[12] Tarr, P.; Ossher, H.; Harrison, W.; Sutton, S. M.: *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, Proc. of the Int'l Conference on Software Engineering (ICSE'99), May 1999

Management und Wartung von Produktfamilien-Assets

Maintenance Aspects of Software Product Lines

Joachim Bayer and Dirk Muthig
Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
+49 6301 707 161
{bayer, muthig}@iese.fhg.de

ABSTRACT
Software product lines have various benefits. Among them are an increased level of reuse, high quality of the resulting systems, as well as reduced development costs and effort. Another very important benefit of software product lines is the possibility to share maintenance effort among the systems developed using the product line.

Though the importance of the maintenance and evolution of a product line infrastructure has been widely recognized, these activities are not sufficiently addressed in literature and their implications have not been fully investigated. In this paper, we try to open a field for discussion by presenting our view on maintenance aspects of software product lines.

1 Introduction

The core of a product line is a product line infrastructure, a reuse infrastructure, that supports the location, the evaluation, and the adaptation of reusable assets, as well as a more accurate planning of development projects. Product line engineering is usually split into three phases: the construction, the usage, and the maintenance of a product line infrastructure.

The initial step of the construction of the product line infrastructure is to determine its scope. That is, based on the stakeholders' business objectives and the planned products, the range of characteristics that systems in the product line should cover is identified. Then, subsequently, the requirements for the product line are documented in a product line model, a generic reference (or domain-specific) architecture is developed, and its components or modules are partially designed and implemented.

After the construction phase, the product line infrastructure consist of assets used during application engineering, when a single product line member is created. The creation encompasses the instantiation of the product line model and the reference architecture, the creation and/or reuse of assets that constitute the instance, and the validation of the resulting product. Additionally, reusable assets that are needed, that have not been created yet, are developed and put into the reusable asset base.

Because a product line infrastructure is a long-term investment and concepts within the domain or other requirements on the product line change over time, the infrastructure must be continuously maintained.

A product line encompasses numerous products that are supposedly used over a long period of time. The long-term investment in an infrastructure that allows the derivation of product line members can only pay off when, over the whole life-cycle, all product line members are based on the same common asset base, the product line infrastructure. To achieve that, the maintenance activities for the different product line members that are in production and the maintenance activities for the product line infrastructure must be coordinated. Being a long-term investment, maintaining a product line requires the infrastructure to be kept up-to-date. This means adapting it to changed needs of customers, changes in the application domain, or evolving it to enter new market segments. The evolved assets can potentially be used to update product line members in production.

The complexity of these activities originates from the required coordination with update cycles of delivered systems and application engineering activities that create new product line members, which typically produce additional requirements on the product line infrastructure.

Though the importance of the maintenance and evolution of a product line infrastructure has been widely recognized, these activities are not sufficiently addressed in literature and their implications have not been investigated. In this paper, we try to open a field for discussion by presenting some maintenance aspects of software product lines.

We do this by first discussing product line infrastructures in section2 and then products that are derived from these infrastructures in section3. Maintenance activities for product line infrastructures and derived product line members are presented in section4. Section 5 finally discusses configuration management aspects of product line maintenance.

Throughout this paper, we use an example for illustration. It is based on a case study that is being performed at Fraunhofer IESE. The case study is a library system product line.

2 Product Line Infrastructure

A product line infrastructure typically consists of two parts: firstly, a set of product line assets that are built for (re-)use during the application engineering process, and secondly, a decision model that represents the knowledge needed to build applications with the infrastructure.

Product Line Assets

A product line asset is any kind of asset that captures information about the systems of a product line. The assets

are built to be reused when systems in the product line are developed. There are two types of information captured: information that is valid for all systems in the family (i.e., commonalities) and information that varies from system to system (i.e., variabilities). With respect to these information types, there are three general categories of product line assets:

1. Assets that capture only commonalities and, thus, are completely relevant for each system in the product line. Examples are public standard definitions or system modules that provide basic services needed by all systems in the product line.
2. Assets that consist of information units that do not contain any variability but that may not be relevant for all systems in the product line. An example is a glossary, which also contains definition of terms that describe optional application domain concepts.
3. Assets that capture variabilities explicitly in the form of variation points, such as C-Preprocessor constructs or alternative classes expressed by the strategy pattern in a class diagram.

The third category does only exist in the context of system families. The variation points of assets that fall in the third category must be resolved when a particular product line member is specified. Resolving variation points means that parts of the asset are specialized, included, or excluded. Many assets used to capture variable information are not able to capture points of variation explicitly and cleanly. The reason for that is that the variation of system characteristics is at a higher level of abstraction than the system characteristics themselves. This higher abstraction level is not explicitly supported by most of the commonly used assets and notations. At least, when the number of varying characteristics is large, which is true when a whole system family is modeled, a clean and explicit way for modeling varying characteristics must be defined. Hence, new asset elements are defined to enable the explicit integration of variabilities into an asset. This can be done by extending the meta model of an asset.

In figure 1, some product line assets of the library system family are shown. At the left side, the process of reserving an item in a library is modeled using a UML activity diagram. In the middle of figure 1, a simplified class diagram of a library system is shown. The UML notation has been extended, the grey-shaded classes and operations are optional (They realize the optional reservation and suggestion capabilities). With respect to reservation capabilities, the classes `ReservationList` and `Reservation`, as well as the library system operations concerning these capabilities, are optional. The KobrA method developed at IESE models product lines in the described way [1].

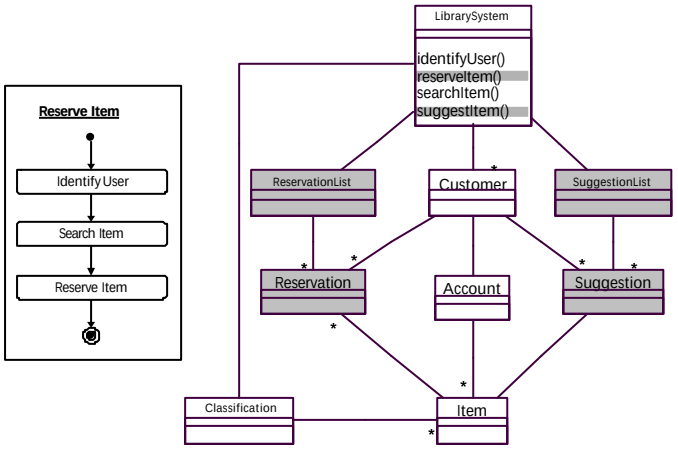
Decision Model

There is typically a large number of variation points in the assets of a complete product line infrastructure. Consequently, it is, even for experts, hard to control the rationales for each variation point, as well as the complex interrelationships and dependencies among them. To support the intellectual control, typically, a decision model, which captures this kind of domain knowledge, is built on top of the variation points.

A decision model consists of a decision hierarchy, which is grounded on simple decisions that capture the rationale and the possible choices for a single point of variation. Dependencies among decisions are explicitly captured by constraints. Usually, additional decisions are introduced. These decisions are not directly related to a point of variation of an asset but they represent domain variability at a higher level of abstraction. This higher abstraction level is related to sets of interdependent variation points.

An ideal decision model provides decisions at the top level of its decision hierarchy that are understood by domain experts or customers and integrate assets from all stages of the software life cycle. Then, the decision model guides the systematic employment of the product line infrastructure across all stages of software development - from the requirements level down to the code level.

On the right side of figure 1, a part of the library system decision model is shown. The decisions check whether



#	Question	Resolution	Diagram	Effect
1	Are customers allowed to reserve items?	Yes		
		No	Activity Diagram „ReserveItem“	Diagram is excluded.
			Class Diagram	Remove classes „ReservationList“ and „Reservation”
				Remove operation LibrarySystem.reserveItem
2	Are customers allowed to suggest items?	Yes		
		No	Activity Diagram „SuggestItem“	Diagram is excluded.
			Class Diagram	Remove classes „SuggestionList” and „Suggestion“
				Remove operation LibrarySystem.suggestionItem

Figure 1: Library system product line assets and a part of the corresponding decision model

reservation capabilities and suggestion capabilities are required. The decision's default resolutions are printed in bold.

3 Product Line Instance

A product line instance is a system that is developed with reuse of product line assets for a particular customer. It is the output of application engineering, the process of developing specific product line members. A full description of our view on application engineering is described in [2]. In this section, we consider only the tailoring of a product line asset while reusing it: first, its instantiation in the space of supported variabilities and, second, its extension with aspects that are not supported by the product line asset but that are required by a particular customer.

Decision Model Resolution

The application engineering process is driven by the decision model, that is, while traversing the decision hierarchy, decision by decision is resolved. When a decision is resolved that constrains variation points of a product line asset, the asset is instantiated accordingly, that is, the variation point is removed and replaced by the concrete realization that corresponds to the selected resolution. The resolution process stops when all simple decisions are resolved. The resulting asset instance is the variant of the variants supported by the product line infrastructure that is closest to what is required in the specific context. Often, even the closest variant is not exactly what is required and, thus, further modifications are necessary. The made resolutions are stored in a decision model instance.

Customer-Specific Changes

When none of the variants supported by the infrastructure is acceptable for a customer, additional requirements must be fulfilled either via infrastructure extensions or customer-specific changes - an important, strategic decision. The asset instance, which is the closest to the required one, is input for the extension process. Hence, the required asset instance is expressed in differences to the already supported instance. This simplifies the later integration generally with the infrastructure or the maintenance activities, which we will describe in more detail in the next section.

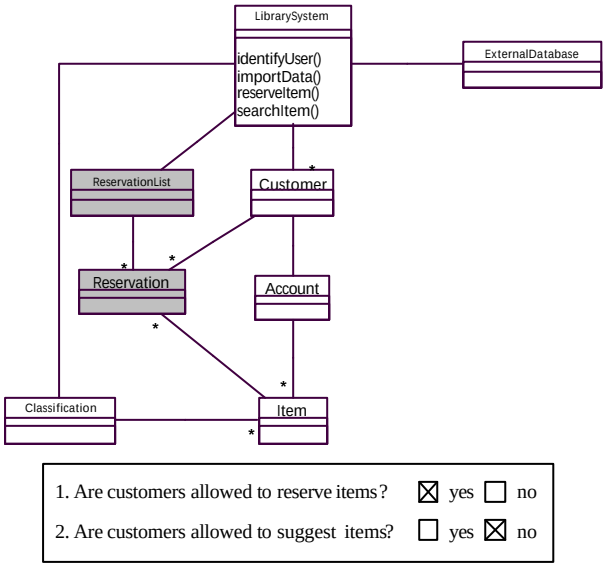


Figure 2: Class diagram and the decision model instance of the archive library system

In our example, we consider an archive library, where it is possible to reserve but not to suggest items. The decisions shown in figure 1 have been resolved with 'yes' and 'no', respectively. Consequently, the instantiated class diagram contains the classes `ReservationList` and `Reservation`, as well as the operation `reserveItem`. The suggestion capability has been completely excluded. The result is shown in figure 2. In the archive library, a system-specific requirement concerns the import of data about the registered items that do exist in an external database. Such a functionality is not supported by the infrastructure. Therefore, an interface to an external database has been modeled into the instantiated class diagram as customer-specific extension.

4 Maintenance

Once the product line has been constructed, it enters its maintenance phase. The product line infrastructure is the basis for the different product line members that are shipped and put in production. Maintenance requests usually arise at the customers' sites and are concerned with one product line member. The product line member might be maintained in isolation. Information about the request for maintenance and the changes made are then passed to product line maintenance. The goal of product line maintenance is that — over the whole life cycle — the products that have been built based on a product line are still based on the same, common product line infrastructure. In the case that maintenance changes a product line member drastically, it might have to be excluded from the product line. Nevertheless, the goal of maintenance is to avoid these cases. Consequently, the product line maintenance activities and local product line member maintenance activities have to be coordinated

Various policies exist for dealing with the changes that are results of maintenance activities, such as forcing all product line members to adapt immediately, allowing for product line members to adapt only on their subsequent release, or providing patches to current versions. We do not impose any specific policy, yet we strongly recommend keeping the number of concurrent versions of the assets in the infrastructure to a minimum.

In general, there are three types of maintenance [4]:

- Corrective Maintenance: corrects errors.
- Adaptive Maintenance: adapts systems to environmental changes (e.g., hardware or operating system changes).
- Perfective Maintenance: improves the capabilities or functionality of a system.

In the literature sometimes a forth type of maintenance is identified [3]: preventive maintenance that aims at anticipating future problems or trends in the application domain and adapting the systems in order to be ready in time. For product line maintenance, perfective and preventive maintenance have the same consequences. They

alter the capabilities of the product line as a whole and evolve it. Therefore, we do not discuss them separately.

Maintenance requests have their origins usually at a customer's site as a maintenance requests for the respective product line member. Maintenance request can also arise in the context of a product line evolution when the product line as a whole is further developed or adapted to changed needs.

For corrective maintenance, the diagnosis is done locally (i.e., in the context of the system in production). If a correction would impact product line assets, the maintenance is passed to product line maintenance together with a diagnosis report. Corrections that only impact customer-specifics are performed locally.

If adaptive or perfective maintenance activities are requested, application engineering is re-entered. If the product line infrastructure already covers the requested changes, simply a new application is built to handle the maintenance request. Otherwise, the maintenance request is passed to product line maintenance.

The maintenance of the product line infrastructure and its product line members is done in three steps: infrastructure revision, reinstantiation, and customer-specific reintegration. These steps are described in more detail in the following sections.

Infrastructure Revision

The first activity to maintain a product line is to analyze the impact of the changes that are needed to satisfy a maintenance request. The possible impacts range from changes in the product line scope (e.g., when the scope is extended by adding new features to the product line) to changes in a code module (e.g., when an error is corrected). In all cases, the development step, which created the affected asset(s) has to be revisited, which leads to a new revision of the respective asset(s). Then, the infrastructure is revised accordingly. This means that all assets that are affected by the direct change are also changed. In this way, the change is propagated through the product line infrastructure. Traceability within the complete product line infrastructure is essential in order to determine this propagation. If the changes add variability to the changed assets and, therefore, require new decisions, the decision model must be updated.

The result of such an infrastructure revision is a number of product line assets that have been changed (i.e., they exist in a new revision, too) and a new revision of the decision model.

A maintenance request that could arise in the library system example, is the collection of a reservation fee. One library using a library system wants to add that feature and requests a respective perfective change to its product line member. The request is investigated and it is decided to add the feature to the product line, because more libraries that take fees for reservation are expected as customers. After the product line scope has been changed accordingly by adding reservation fees, at some point the assets shown in figure 1 are changed. The result of these changes are shown in figure 3 where support for reservation fees has been added.

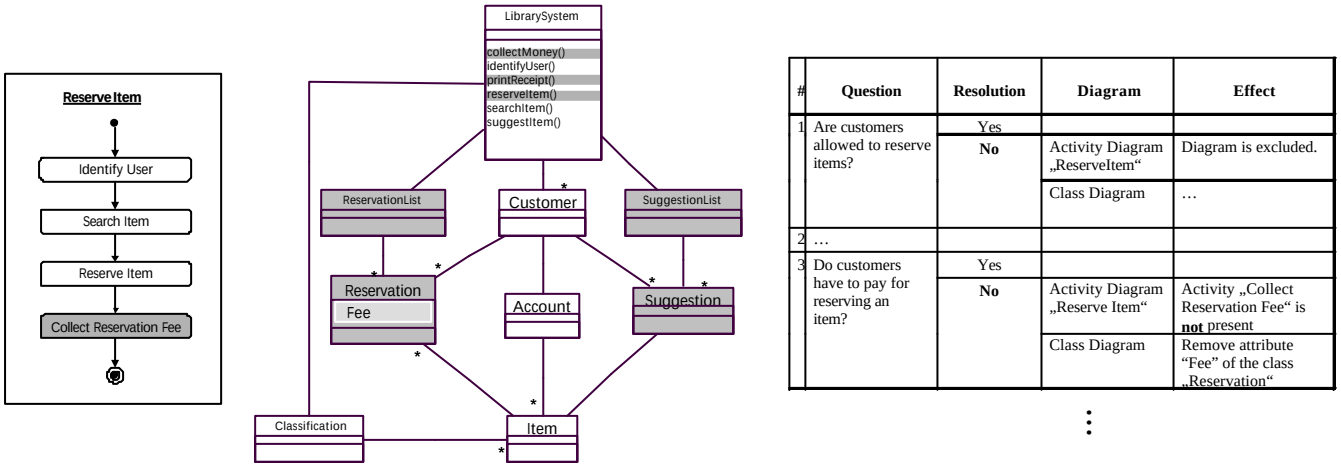
Reinstantiation

After the product line infrastructure has been revised, the product line members that are affected by the changes in the revision can be updated. As mentioned before, there are several policies to handle the update of product line members. When the product line member is to be updated to follow the changes in the product line infrastructure, it is reinstantiated. This is done by reusing the decision model instance. The decisions that were added to the decision model have to be added to the decision model instance and resolved. Whenever possible, the default resolutions of new decisions should correspond to the case before the decisions has been added. Using the augmented decision model instance, the product line member is rebuilt.

In the example, all libraries using a product line member are informed about the new availability of handling reservation fees. The libraries that need this new feature can then get an update to their library system.

Customer-Specifics Reintegration

If a product line member that is updated after the



#	Question	Resolution	Diagram	Effect
1	Are customers allowed to reserve items?	Yes		
		No	Activity Diagram „ReserveItem“	Diagram is excluded.
			Class Diagram	…
2	…			
3	Do customers have to pay for reserving an item?	Yes		
		No	Activity Diagram „Reserve Item“	Activity „Collect Reservation Fee“ is **not** present
			Class Diagram	Remove attribute “Fee” of the class „Reservation“

Figure 3: Revised library system product line assets and decision model

infrastructure has been revised has customer-specific characteristics, the customer-specifics have to be re-integrated with the reinstantiated assets.

Reusing the decision model instance of the archive system, the instantiation of the revised product line infrastructure shown in figure 3 results again in the models in figure 2. For the new, third decision the default resolution has been selected and the system-specific requirement of an external database has been reintegrated.

When the archive system should be upgraded to support a registration fee, only the new decision's resolution must be changed. The resulting model for the archive system with registration fee support is shown in figure 4.

5 Configuration Management (CM)

In the previous sections, we have illustrated our view on the product-related maintenance aspects of software product lines. In a product line, the numbers of product line assets, product line members, and customer-specific changes is typically large. Consequently, tool support is needed to control all the existing product configurations, variants, and revisions. In single system development, these tasks are done by a configuration management system (CMS) but for software product lines no CMS is available on the market today. In this section we will derive requirements on a CMS for product lines from the description given before.

CM of the Infrastructure Assets

The product line infrastructure consists of product line assets and a decision model. Except for the semantic consideration of variabilities, there is no difference between a product line infrastructure and the asset set of a single system. In both cases, each single asset is controlled by the CMS and a configuration of the infrastructure or, respectively, of the system, consists of valid set of the controlled assets in a particular revision. Consequently, the product line infrastructure can be controlled by a standard CMS.

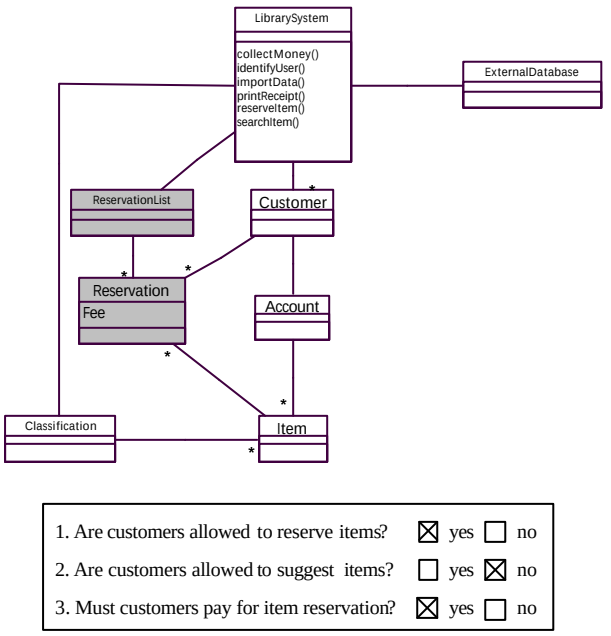


Figure 4: Re-integrated class diagram of the archive library system with reservation fee support

CM of Instances

The instance itself is represented by a decision model instance, the resolutions of the decisions. Decision model instances can also be controlled by a standard CMS. However, there is no support for the transformation of the resolutions of a previous decision model revision into the resolutions of a new revision of this decision model. This is needed in cases where the product line infrastructure including its decision model has been revised and therefore an instance must be generated by re-instantiating the revised infrastructure.

The transformation can partially be supported by the decision model itself when the default resolutions of new decisions are defined in a way that they generate the instances that have been generated by former infrastructure revisions like the exclusion of the reservation fee support in the example. If this is not possible, for example, when a decision has been changed, the maintainer must manually intervene with the re-instantiation process. In any case, a manual validation of the transformation is required but its focus can be minimized by the supporting tool.

CM of Customer-Specifics

Customer-specifics are integrated into instances of product line assets. When the generated instances are checked in a standard CMS, the CMS can track the customer-specific changes. The special requirement on the CMS for product lines is the capability of re-applying the changes to re-instantiated assets when the delivered system is updated in the context of maintenance activities. The re-application itself is supported by standard CMSs but not the analysis whether the resulting assets are valid. This analysis can be realized by extending the merge capabilities of standard CMSs.

6 Conclusions

In this paper, we have presented our view on maintenance aspects of software product lines. Maintaining and evolving a product line are key activities, which must be supported by tools to be performed efficiently. Unfortunately, there is no configuration management system for product lines available although only some extensions are necessary to cover also the maintenance aspects of software product lines.

REFERENCES

1. Atkinson, C., Bayer, J., Muthig, D., Component-Based Product Line Development: The KobrA Approach. Accepted by the First Software Product Line Conference, August 2000.
2. Bayer, J., Gacek, C., Muthig, D., and Widen, T. PuLSE-I: Deriving Instances from a Product Line Infrastructure, in the Proceedings of the *7th Annual IEEE Conference on the Engineering of Computer-Based Systems (ECBS)*, Edinburgh, 2000.
3. Pressman R. S., Software Engineering; *A Practitioner's Approach*, McGraw-Hill Book Company, 1996.
4. Sommerville, I., *Software Engineering*, Addison-Wesley, 1996.

Configuration Management for Software Product Lines

Roland Laqua and Peter Knauber

Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
+49 6301 707 161
{laqua, knauber}@iese.fhg.de

ABSTRACT
This position paper proposes a new approach to support management of the assets used for software product line development. Examples for these assets are the domain (or product line) model, reference architecture, design, and code. New configuration management concepts provide means to create, maintain, and evolve these assets efficiently and consistently, including not only their common parts but product-specific parts as well.

1 Introduction

The basic concept behind software development in product lines is goal-oriented development for and with reuse. Reuse is achieved through synergistic effects during development and maintenance of common system parts. In order to plan and develop these common parts to be usable by future members of the product line, it is necessary to consider the main variabilities among the future products during modeling, architecting, design, and implementation. The commonalities are represented together with the main variabilities in generic assets.

There are two reasons for this separate treatment:

- First, tools to support the manipulation of generic models including model instantiation[1] are basically not available. Thus, in traditional product line approaches, tools for single system development are adapted or extended in such a way that they provide some support for genericity, at least the respective notation.
- Second, keeping them separate from each other allows for the parallel development and maintenance of product line members.

Two general kinds of processes have to be considered for generic product line assets:

- feed forward processes to derive them and to add new or change existing functionality and
- feed back processes to (re-)integrate changes or additions to product-specific assets into the common core in order to reuse them in other products as well.

Feed back processes, especially, are difficult and error prone because traditionally, product-specifics are developed (and typically also stored) separately from the generic assets representing the common core. Thus, it is difficult to decide if and which product-specific aspects should be integrated into the core and it is even more difficult to perform the integration.

The lack of appropriate tools makes integration of product specifics into the common core extremely difficult: there is basically no automated support for checking constraints on existing models if these constraints apply to model variations, or to even represent them explicitly.

Even traditional configuration management tools do not provide enough support for these particular problems (e.g., constraint checking on system variants): They allow representation and checking of constraints but do not support product variants (e.g., alternatives or options).

The position we present in this paper can be briefly summarized as follows:

> In order to support management of product line assets appropriately, the definition of the way generic asset data are structured has to be reflected in the supporting configuration management system. This system has to be set-oriented, attribute-oriented, unification-oriented, and structure-oriented.

This position will be explained in the remaining sections of this paper. Section 2 presents the basic assets of a product line infrastructure, in section 3 the configuration management elements on which these product line assets have to be mapped are explained. Problems of traditional configuration management during maintenance and evolution of a product line are shown in section 4. Finally, the proposed configuration management prerequisites for the maintenance of a product line infrastructure are presented in section 5.

2 Software Product Line Assets

The traditional software product line (SPL) infrastructure consists of

- assets representing the product line (which are the smallest units from a configuration management point of view),
- the relations between the assets, which are captured by the product line configurations.

1.The term “instantiation” here refers to hiding model parts that do not belonging to a certain product and show the parts belong only to this product while checking existing constraints.

Product line assets can be considered as intermediate software products that are needed to produce applications in a certain domain. They can be subdivided into domain assets that capture the knowledge of the business domain, and system assets representing the information specific for certain systems of the product line, which are needed to derive (e.g., parametrize, generate, instrument) the assets for these specific systems from the (generic) domain assets.

Domain assets can be categorized into three groups: *common assets* (holding information that is identical for all members of the product line), *generic assets* (for capturing information about variant parts of the software product line in one asset), and *configuration assets* (representing the information needed to configure systems in the product line).

- **Common assets** only hold information that is common for all members of the product line and contain no information specific to a certain system. These assets should not vary because they are used in the same way by all members of the product line.
  Examples are database access libraries, which very often can not be changed by the developers, or trace and log services, which can be shared by most systems of a product line.
- **Generic assets** capture information about different variants of a product line asset in one common asset in order to increase its reusability and adaptability for different systems of a product line. The description of the different variants strongly depends on the SPL method used, very often it is achieved by means of the introduction of variation points via conditional blocks or by feature modeling methods (see [1],[2] pp.82).
  An example can be seen in [3], [5] p.8. There the optional and alternative sub-components are decorated with conditional elements that are used to decide whether or not the sub-component will be part of the specific product.
- **Configuration assets** contain the information on how to derive a specific asset from a generic one.
  An example is the decision model that describes the relation of possible decisions ([4]) in the business domain and variation points of a certain asset and how they can be resolved.

System assets as mentioned above represent the information specific to a certain system of the product line and the knowledge needed to build a certain system of the product line.

- **Specific assets** describe information that is specific to a certain system or sub-group of systems in the product line. Typically they contain no variabilities and share less common parts with other assets.
  Examples are the description of the compilation parameters for a certain asset or GUI forms, which are very often system specific.
- **Production assets** define a certain system of the product line. They configure the different parts that belong to a system and resolve the decisions that are necessary to derive specific assets from generic ones.
  Examples are tuples of decisions that resolve variation points in generic assets or makefiles that contain the dependencies between assets belonging to a certain system.

Product Line System
A product line system defines all the domain and system assets necessary to build a specific application from the product line infrastructure. This means that all decision tuples that are needed to resolve the generic assets together with the appropriate decision models will be described as part of a product line system. Moreover, all application specific assets used are defined there.

3 Configuration Management Elements

The main objective of configuration management (CM) in SPL is to support maintenance of the product line infrastructure. This means to support management of the life cycle of the product line assets, their different customer specific variants and revisions, their configuration of the assets in specific systems, and their change over time.

In order to meet these requirements, CM systems must be able to map SPL elements onto their own structures and to support the different SPL processes like derivation of specific assets and reintegration of changes. The basic elements of CM systems to implement these requirements are *items* (which are the elementary units), *configurations* (which define the relations between items), *revisions* (for describing the change of items and configurations over time), and *variants* (which define different shapes of the same item or configuration).

Items
Items are elementary units of a software product that must be uniquely identified. They can be instantiated for different versions (see “Version” on page 2). This definition is open to any kind of software product because no type restriction is made.

Items can be categorized in two groups:

- **plain items** that are not derived from any other items (but nevertheless can depend on others) and
- **derived items** that are derived from one or more other items (which again can be plain or derived).

This distinction is needed because the versions of derived items also need the version information about the plain items they stem from.

Version
A version is an instance of an item. Two instances of the same item that are developed in parallel are related as variants, whereas a version that follows by a change of another version of the same item is related to this version as a revision. The variant relation defines alternative forms, whereas a revision expresses the change of an item over time.

Configuration
A configuration defines a composition of items and possibly further configurations. Configurations have to obey the following rules: First, a configuration can only be assigned to one version of a given item, second, a version of an item

may be a member of several configurations, third, a configuration can be recursive, and fourth, a configuration version can be assigned to another configuration.

Mapping SPL to CM elements

The mapping of SPL elements to CM elements creates a certain net of plain and derived items of generic, common, specific, production, and configuration assets. The possible relations between SPL and CM elements, as well as the source assets they are derived from, are depicted in Table 1.

SPL Asset	Corresponding CM Asset	Source of Derivation
Generic Asset	Plain Item	Plain Item
	Derived Item	Generic Asset + Configuration Asset + Production Asset
Common Asset	Plain Item	
	Derived Item	Common Asset + Production Asset
Configuration Asset	Plain Item	
Specific Asset	Plain Item	
	Derived Item	Specific Asset + Production Asset
		Generic Asset + Configuration Asset + Production Asset
Production Asset	Plain Item	
System	Configuration	Subset of Generic Assets + Subset of Common Assets + Subset of Specific Assets + Production Assets

Table 1: SPL - CM element map

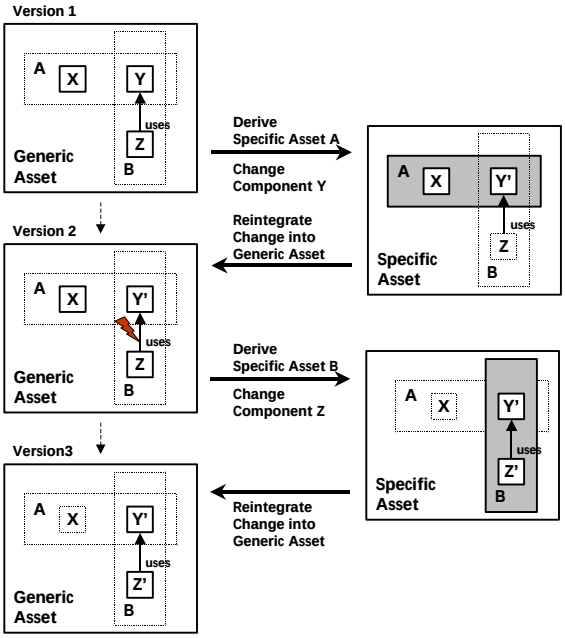


Figure 1: Consistency Problem

4 Problems of Traditional CM for SPL

The traditional approach suffers from certain problems that arise in the different activities of the change processes and that are related to the fact that the product line assets are treated as self-contained items in a CM system without considering the inner structure of an asset. Typical problems are the *reconstruction problem*, the *change propagation*, and the *consistency problem* (which are related to the correctness of the product line and which emerge during the change process), and the *set problem* (which describes difficulties in treating a set of systems instead of a single system).

Consistency Problem: The reintegration of changes made in specific assets can lead to inconsistencies in another specific asset that is derived from the same generic asset. Therefore, the reintegration raises the need to adapt parts of the other specific asset.

This happens, for example, if a certain part of a specific asset B (figure 1) uses a common part that is shared by a specific asset A and that was changed in the specific asset A. The derivation of B from version 2 of the generic asset would then lead to an inconsistent specific product B. This implies that before the generic asset can be used for all specific assets that can be derived from it, all change effects have to be determined and the changes have to be integrated for reaching consistency again.

Reconstruction Problem: The parallel change of specific assets derived from the same generic asset can lead to specific assets that are not reconstructible. Figure 2 shows an example, where two specific assets A and B are changed at the same time and these changes are then reintegrated into the original generic asset at different points in time (e.g., A before B). This implies that the changed specific asset B can not be reconstructed from any of the three generic asset versions because in version 1 and version 2 the part Z', and

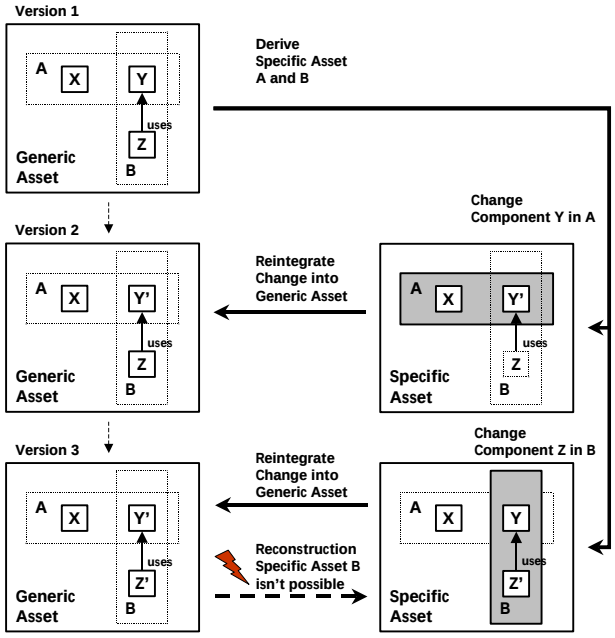


Figure 2: Reconstruction Problem

in version 3 the part Y are missing.

Set Problem: The simultaneous change of common parts of specific asset variants derived from the same generic asset is impossible or hard to achieve. Changes in the traditional CM only allow working with one instance of the product line in parallel. This is because each instance is treated as a certain variant of the product line instead of handling all its variants as a set.
Furthermore, the identification and selection of common parts is not supported by traditional CM systems. Figure 3 illustrates the problem and a possible solution through a respective specification of the affected asset parts with an example.

Change Propagation Problem: Traditional CM systems are not able to represent revisions and variants of a product line in one common model. Thus, there is no possibility to propagate changes for more than one variant at once. This is depicted in figure 4, where a change in a specific asset derived by generic asset version 1.0 has to be reintegrated twice, once in the generic asset version 1.1 and then in the generic asset version 2.1 of the generic asset variant version 2.0.

5 Proposed CM Concept for an SPL Infrastructure

In order to overcome these problems, we propose a CM concept that has the following four major properties:

Set-orientation: The CM concept supports manipulating consistent sets of items and configurations. For this it provides operations to identify and select them according to their properties as well as to manipulate them at once.

Attribute-orientation: The CM concept supports

- identification and selection of plain, derived and composed items that are tagged with attributes,
- propagation of attributes among the items, and
- design of relations among the items in order to enable attribute propagation.

Unification-orientation: The CM concept supports

- unification of an attribute model for all items of the product line and
- identification and selection schemes for assets.

In the most CM systems a strong identification scheme comes along with a weak selection scheme or vice versa. For example, the selection of variants can be realized by the C preprocessor that provides a strong variant identification mechanism via arithmetic expressions. On the other hand, the selection of versions through the conjunction of attributes is quite complex using that tool. In order to overcome this problem different steps are necessary:

- Firstly, items (configurations) and SPL assets (SPL Infrastructure) are unified. This means that they are the same entity inside of the repository (see Figure 5, Figure 6 melting). Thus, the structural model of an SPL asset can be used to build up CM approaches for versioning, change propagation and a unified selection and identification scheme using of the repository meta model.
- Secondly, items and configurations in the repository are mapped on a common feature logic attribute model. Thus, the selection and identification scheme of items and configurations can now rely on the same expression model for both.
- Thirdly, the items (configurations) are modeled and interpreted as features themselves. This means that additional features decorating items (configurations) are treated in the same way as the items (configuration). This way, the identification and selection scheme of items and their attributes can be unified in one expression.

Structure-orientation: The CM concept supports the definition of optional and alternative assets and asset parts for product lines.

Mapping of product line assets to CM items
Each product line asset has to have a dual nature (figure 5): on the one hand it is handled as a CM item, and, on the other hand, it fulfils its purpose as a product line asset.

To reach this goal the data structure of each asset has to be modeled on the data structure level of the CM repository. Thus, it will be possible to decorate each entity (figure 5) of the data structure with features like the ones in feature logic ([5]). There, a feature is not only the decoration of an entity but also the entity itself. Even the changes of an entity can be

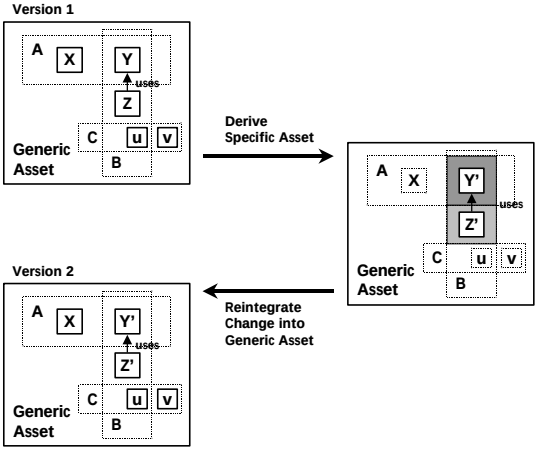


Figure 3: Set Problem

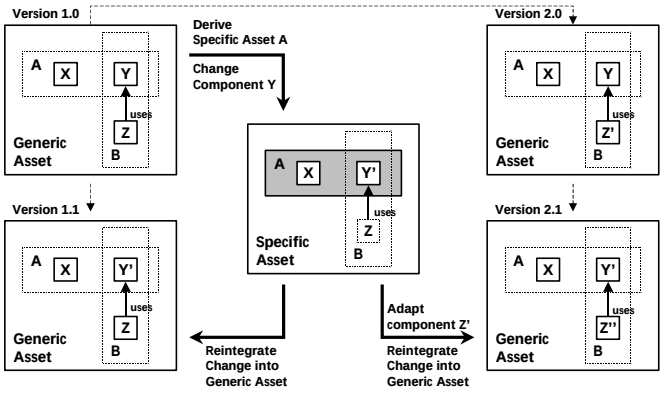


Figure 4: Change Propagation Problem

seen as a feature, enabling the unification of variants and revisions in one common model. Thus, product line assets and CM items can be modeled and treated on the same level.

Advantages of the proposed concept
The proposed CM concept solves the problems listed in section 4:

- **Consistency Problem**: The set-orientation and the structure-orientation allow to define sets of specific assets. That allows to change dependent parts of different specific assets derived from the same generic asset in parallel.
- **Reconstruction Problem**: The attribute-orientation allows to unify parts of different revisions that were developed in parallel into one common configuration item. Therefore, the reconstruction of certain specific assets created during development is always possible.
- **Set Problem**: The set-orientation allows to define subsets of arbitrary parts of specific assets derived from generic assets. Changes to these sets affect all selected assets at the same time.
- **Change Propagation Problem**: Set and attribute-orientation enable to group parts of different variants of specific assets that were developed in parallel into a common set and to change them at one time. This makes multiple reintegration superfluous.

The realization of such an SPL infrastructure requires anchoring the product line asset and configuration models in a version-set-oriented repository ([6]). This is necessary to interpret a certain CM item as a set of different variants depending on the properties on the data structure level, as must be done for generic product line assets. Additionally, it is necessary to explicitly express the relations among different assets. This will be done by the feature logic approach (see [5]) based on the product line assets' data structure. This allows to design the dependencies and attributation on a finer grained level than with traditional CM concepts.

6 Conclusion

In this paper we present a new concept for configuration management to support management of software product line assets. Typical problems of traditional approaches in a product line context are described and their solution using the new concept is sketched. Future work will cover the refinement and, later on, the implementation of the concept using existing tools (if possible).

REFERENCES

[1] K.C. Kang, S.G. Cohen, W.E. Novak, A.S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. Report CMU/SEI-90-TR-21, Software Engineering Institute (SEI), November 1990

[2] K. Czarnecki, U.W. Eisenecker, Generative Programming, Addison-Wesley, May 2000

[3] D.M. Weiss, Defining Families: The Commonality Analysis, Lucent Technologies Bell Laboratories, 1997

[4] C. Atkinson, J. Bayer and D. Muthig, Component-Based Product Line Development: The KobrA Approach, 1st International Software Product Line Conference, Pittsburgh, August 2000.

[5] A. Zeller, Configuration Management with Version Sets, A Unified Software Versioning Model and its Applications, PhD thesis, http://www.cs.tu-bs.de/softech/papers/zeller-phd

[6] C. Reichenberger, Konzepte und Verfahren für die Software-Versionsverwaltung, Universitätsverlag Rudolf Trauner, Linz 1994

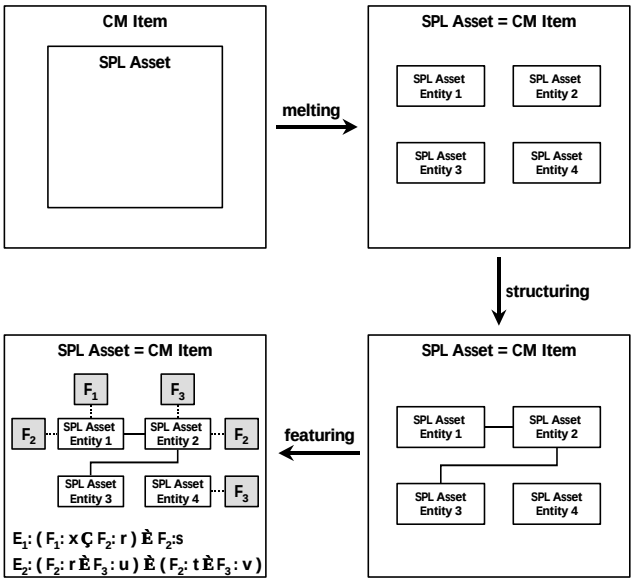


Figure 5: Feature Set Items

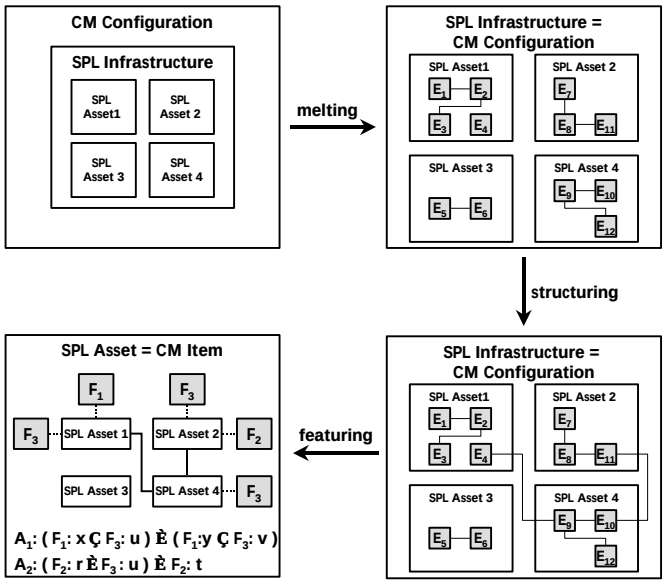


Figure 6: Feature Set Configurations

Komponenten-Auswahl für Software-Produktfamilien[1]

Klaus Pohl, Andreas Reuys
Lehrstuhl für Software Systems Engineering
Universität GH Essen
{pohl, reuys}@informatik.uni-essen.de

1 Einleitung

Geeignete Wiederverwendung von Softwarekomponenten führt zu einer effektiveren Softwareentwicklung. Neben einer kürzeren Entwicklungszeit („time to market“) und Kosteneinsparungen birgt die Verwendung von „Commercials of the shelf“ natürlich auch Risiken (z.B. Sicherheit oder Fehlerkorrekturen, siehe [Voas98]). Component-Based Software Engineering (CBSE) ist eine Richtung im Software Engineering, die sich mit den Fragestellungen der Wiederverwendung von Softwarefragmente bei der Entwicklung großer Systeme beschäftigt. Die Integration von Komponenten wird von vielen Entwickler als essentielle Technik des CBSE angesehen und vorangetrieben (siehe [DeVi97] oder [BrWa96]). Ein Erfolg der Integrationsbemühungen ist eine partielle Aufhebung des „Architectural Mismatch“ [GAOc95], wie [SuKn96] berichten.

Während die technische Integration kommerzieller Komponenten in den letzten Jahren vielfach untersucht und erheblich weiterentwickelt und durch die Standardisierungen vereinfacht wurde, fehlten noch strukturierte Ansätze zur Auswahl der Komponenten und dies obwohl die effektive Komponentenauswahl immer mehr an Bedeutung gewinnt. Bis vor wenigen Jahren wurde die Komponentenauswahl relativ unsystematisch betrieben. Eine Verbesserung der Auswahl-Prozesse wurde kaum beachtet und war aufgrund der unsystematischen Vorgehensweise auch kaum möglich. Wie in [Kont96] berichtet, wird der Auswahl-Prozess sehr oft von ungeschultem Personal durchgeführt. Erste Ansätze für systematische Komponentenauswahl werden in [Kont96] und [MaNc98] vorgestellt.

Product Familiy Engineering behandelt die methodische Entwicklung von einer Menge von Anwendungen, der Produktfamilie. Diese Anwendungen zeichnet aus, dass sie einen Software-Kern besitzen, der allen Anwendungen der Produktfamilie gemeinsam ist. Die Entwicklung dieses Kerns ist Bestandteil des Domain Engineerings (DE). Das DE umfasst neben der Analyse der Zieldomäne, der Erstellung des Designs des Kerns und der Implementierung gemeinsamer Code-Fragmente auch die Analyse der unterschiedlichen Applikationen. Die Ergebnisse aus Analyse, Design und Implementierung im Domain Engineering werden als generische Assets bezeichnet.

Die Entwicklung einer kundenspezifischen Anwendung wird als Application Engineering (AE) bezeichnet. Eine kundenspezifische Anwendung ergänzt diese generischen Assets durch entsprechende kundenspezifischen Erweiterungen und Anpassungen. Zunächst werden die kundenspezifischen Anforderungen der Anwendung erfasst und mit dem Model der Domänenanalyse ergänzt. Das Design der Anwendung ergibt sich aus dem Design der Domäne und anwendungsspezifischen Anpassungen. Die Implementierung der Anwendung entwickelt sich aus der Verwendung des im DE entwickelten Codes und dem anwendungsspezifischen Design (siehe den „Product Line Process“ des PRAISE-Projektes, via http://www.esi.es/Projects/Reuse/Praise/publications.html verfügbar).

[CzEi00] unterscheiden zwischen vertikalen Domänen (Domänen von Systemen, z.B. Flugbuchungs-Systeme) und horizontalen Domänen (Teile eines Systems, nach Funktionalitäten geordnet, z.B. GUI-Bibliotheken). Das im Folgenden dargestellte Verfahren zielt auf die Selektion von COTS für erkannte horizontale Domänen ab.

Ziel dieses Papiers ist es, die wesentlichen Bestandteile von Komponentenauswahl-Prozessen herauszustellen und die Bedeutungen, Auswirkungen und Aufgaben für den Software-Produktfamilien Kontext aufzuzeigen. Zu diesem Zweck wird ein Rahmenwerk vorgestellt, welches auf der Basis vorhandener Auswahl-Prozesse entwickelt wurde (Kap. 2). Die Anwendung der Komponentenauswahl im Product Family Engineering und die sich daraus ergebenden Auswirkung wird in Kap. 3 beschrieben. Kap. 4 fasst das Papier zusammen und zeigt die künftigen Aktivitäten auf.

Klärung der verwendeten Begriffe

Der Begriff *COTS* (von engl: Components Of The Shelf oder Commercials Of The Shelf) wird in dem Kontext dieses Papiers nicht eingeschränkt, sondern vielmehr in dem breitest-möglichen Sinn verstanden. Dies impliziert, dass der Begriff unabhängig von dem Ursprung der Komponente und der Möglichkeit zur Code-Änderung verstanden wird. Eine ausführliche Diskussion der COTS-Begriffserläuterung ist in [CaLo00] zu finden.

Ebenso sind keine Einschränkungen in dem Begriff *Anforderung/Requirement* impliziert. In dem Kontext dieses Papiers kann eine Anforderung funktionaler, nicht-funktionaler, qualitativer, technischer oder einer weiteren beliebigen Natur sein (z.B. spezielle Anforderungen an die Architektur oder bestimmter Kosten).

[1] Diese Arbeit wurde gefördert durch das Projekt ESAPS des BMBF, Förderkennzeichen 01 IS 902 C und durch das europäische ITEA Programm, Projektnummer 99005.

2 COTS-Auswahl für Produktentwicklungen

In dem folgenden Kapitel werden zunächst die Probleme und Möglichkeiten der COTS-Auswahl in der Software-Entwicklung erläutert (2.1). Ein Rahmenwerk zur Auswahl von Komponenten wird zunächst motiviert (2.2), anschließend wird ein entsprechendes Rahmenwerk definiert und erläutert (2.3).

2.1 Probleme und Möglichkeiten der COTS-Auswahl

Bevor die Integration von Komponenten in die Architektur erfolgen kann, müssen die Komponenten zunächst anhand von Funktionalitäten, Kosten etc., d.h. anforderungsbasiert ausgewählt werden. Zu diesem Zweck müssen die Anforderungen an die Komponente –so noch nicht vorhanden- ermittelt und spezifiziert werden. Basierend auf den Anforderungen und dem Grad, wie gut vorhandene Komponenten diese Anforderungen erfüllen, wird eine Entscheidung getroffen, welche Komponente in das Softwaresystem integriert wird.[2]

Der Umgang mit COTS birgt eine Menge von Möglichkeiten und Risiken. Die Auswahl falscher Komponenten kann von einem nahezu vollständigen Projektfehlschlag (z.B. durch eine unvollständige oder falsche Anforderungserfassung oder unvollständiges Testen der Anforderungen) bis zu einem erheblichen Mehraufwand im Projekt führen, falls zur Erfüllung der Anforderungen ein umfangreiches Anpassen der Komponente in das System erforderlich ist.

Auf der anderen Seite bieten sich neben den Reuse-Effekten Möglichkeiten, wie sie traditionellen Entwicklungsprozessen nicht möglich sind: Die Komponenten sind bereits vorhanden, so dass die Erfassung der Anforderungen direkt an den Komponenten erfolgen kann (z.B. IKIWISI -„I Know It When I've Seen It"- die Erfassung von Anforderungen geschieht anhand ähnlicher Produkte). Moderne, use case-basierte Ansätze zur Erfassung von Anforderungen müssen nicht in andere Modelle (ER-, OO-) überführt werden, wenn das Ziel, eine realisierte Anforderung, bereits vorliegt. Ferner kann ein zentraler Bestandteil use case-basierter Ansätze, die Szenarien, in den Komponenten-Tests wiederverwendet werden.

Durch die genannten Vor- und Nachteile man kann erkennen, dass der Einsatz von COTS durchaus lohnenswert ist. Es gilt jedoch stets zu beachten, dass eine kosteneffektive Lösung mit COTS nur dann gewährleistet ist, wenn die ausgewählten Komponenten die kundenspezifischen Anforderungen realisieren oder sich mit verträglichen Aufwand entsprechend erweitern lassen.

2.2 Motivation der Erstellung eines Rahmenwerks zur COTS-Auswahl

Um eine strukturierte Durchführung zu ermöglichen und die Ergebnisse überprüfbar und nachvollziehbar zu machen, muss die Auswahl von Komponenten anhand eines definierten Prozesses durchgeführt werden. Innerhalb dieses Prozesses müssen Methodiken für die Auswahl von Komponenten in Beziehung gesetzt werden. Diese Methodiken beginnen mit der Erfassung der Anforderungen, beinhalten die Abläufe und Metriken für die Ausführung von Komponenten-Tests und enden bei der letztendlichen Komponentenauswahl.

Die bisherigen Auswahl-Prozesse haben eine initiale Methodik zur Auswahl von Komponenten in Softwareentwicklungen geliefert. Durch die Definition dieser Prozesse wurden individuelle Prozess-Abläufe und Methodiken in Beziehung gesetzt, was einen Vergleich der Ansätze und Techniken erschwert. Das in Abschnitt 2.3 vorgestellte Rahmenwerk soll daher dazu dienen

- einen Vergleich existierender Ansätze unter Verwendung einer zusätzlichen Abstraktionsebene zu ermöglichen,
- das für die COTS-Auswahl benötigte Domänenwissen zu vermitteln und damit einen Rahmen für zukünftige Prozessdefinitionen zur Auswahl von COTS zu liefern,
- durch die explizite Herausstellung der RE-Phasen die Einsatzmöglichkeiten weiterer RE-Techniken (z.B. Szenariennutzung zur Komplexitätsreduktion und Kommunikation der verschiedenen Aktoren) unterstützen,
- Ansatzpunkte zur Nachvollziehbarkeit (Traceability) geben, indem die Arbeitsprodukte der Prozessphasen definiert und damit zu anderen Arbeitsprodukten in Bezug gesetzt werden können und daher
- zu einem verbesserten Vorgehen in Komponentenauswahl-Prozessen führen.

Die Instanziierung eines Prozesses, der insbesondere den Aspekt der Nachvollziehbarkeit berücksichtigt, ist Gegenstand aktueller Forschungsarbeiten.

---

[2] Sollte der Grad der Anforderungserfüllung unter einer bestimmten Marke liegen, kann die Entscheidung fallen, dass die Komponente neu entwickelt und implementiert wird („make-or-buy" decision). Im folgenden wird ein Verfahren beschrieben, welches weder „make-or-buy" Entscheidungen, noch sonstige Entscheidungen die zu einem Abbruch des Verfahren führen, explizit beinhaltet. Die Definition solcher Abbruchkriterien war nicht Bestandteil dieser Forschungsarbeiten, sie sollten aber problemlos an die entsprechenden Stellen des Prozesses eingefügt werden können.

2.3 Ein Rahmenwerk zur anforderungsbasierten COTS-Auswahl

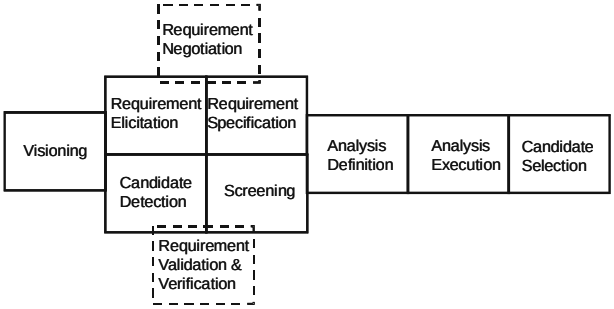


Abbildung 1: Die Phasen des Rahmenwerks zur anforderungsbasierten COTS-Auswahl

Basierend auf der Analyse uns bekannter COTS-Auswahl-Prozesse ([Kont96], [MaNc98]) und der Techniken in CBSE-Prozessen (z.B. [BrWa96], [DeVi97]) definierten wir ein Rahmenwerk (siehe Abbildung 1). Dieses Rahmenwerk ist eine Abstraktion der identifizierten Phasen der analysierten Auswahl-Prozesse. Ein Schwerpunkt unseres Ansatzes besteht aus der expliziten Modellierung der vier Phasen des Requirements Engineering (siehe [Pohl96]), welche häufig nur implizit Verwendung finden.

Im folgenden wird das Rahmenwerk zur Auswahl von COTS anhand der Aktivitäten der einzelnen Phasen beschrieben:

Visioning: Das Ziel der Visioning-Phase ist das Propagieren des Software-Ziels dieser Komponente an die Requirements Engineer oder das Entwicklungsteam. Dieses Team muss so viel Vorgaben wie möglich erhalten, um die nächsten Phasen (Requirements Elicitation und Candidate Detection) selbstständig durchführen zu können.

Requirement Elicitation: In dem Kontext der Komponentenauswahl dient die Elicitation-Phase dazu, die Anforderungen an die Komponente so weit zu bestimmen, dass sowohl eine Marktanalyse als auch eine spätere Spezifikation der Anforderungen stattfinden kann.

Candidate Detection: Mittels der Durchführung einer Marktstudie wird eine Liste möglicher Komponenten Kandidaten erstellt. Die Angaben zu jedem Kandidaten sollten in einem projektspezifischen Template festgehalten werden, welches sowohl allgemeine (Hersteller, Vertrieb, Support, Kontaktadressen, Firmengeschichte, erfolgreiche Projekte, Kompatibilität, etc) als auch projektspezifische Informationen beinhaltet. Ferner sollten die Komponenten in einer Testversion verfügbar gemacht werden.

Requirement Specification: Die Spezifikation der Anforderungen wird basierend auf der Requirements Elicitation-Phase und ersten Rückläufen der Marktstudie erstellt. Art und Umfang der Notation sollten an das aktuelle Projekt und den involvierten Personen angepasst werden. In Hinblick auf die Reduktion von Komplexität (vgl. [WPJH98]) und den späteren Komponenten Test-Phasen, bieten Use-Case-basierte oder Szenario-basierte Vorgehen Vorteile: Die Definition einer Aktionenfolge ist Szenarien und Testläufen gemeinsam, daher besteht die Möglichkeit zur Wiederverwendung bei den später definierten Testfällen. Für die am Ende erfolgende Auswahl müssen die Anforderungen gewichtet werden, um eine differenzierte Bewertung der Komponenten zu ermöglichen.

Requirement Validation & Verification: Das Abgleichen der spezifizierten Anforderungen mit den Kundenwünschen und dem auszuliefernden Produkt ist Bestandteil dieser Phase. Da die Komponenten vorhanden sind, können Techniken einer „Hands-On Evaluation“ oder einer Demonstration eingesetzt werden. Diese Phase geht einher mit der später beschriebenen Screening-Phase und stellt sicher, dass die Kandidaten die Anforderungen im Wesentlichen erfüllen (z.B. durch die Zusicherung eines „Vision Holders“). Die qualitative Validierung der Komponenten findet jedoch erst in den Analyse-Phasen statt.

Screening: Das Ergebnis dieser Filterphase ist eine Vorauswahl der Kandidaten. Unter der Annahme, dass die Spezifikation recht umfangreich und die Komponenten komplex sind, ist der Aufwand ausführlicher Tests auf alle möglichen Kandidaten nicht zu rechtfertigen. Basierend auf einen projektspezifischen Fragebogen sollte das Ergebnis dieser Vorauswahl zwei bis fünf (Erfahrungswerte der veröffentlichten Prozesse) Kandidaten umfassen.

Requirement Negotiation: Die Erkennung und Auflösung von Anforderungskonflikten ist das Ziel dieser Phase. Da diese Phase in den meisten Fällen kein eigenes Ergebnis, sondern eine Verfeinerung der Elicitation oder der Spezifikation ist, wurde diese Phase gestrichelt dargestellt.

Analysis Definition: Basierend auf der Anforderungsspezifikation werden Testfälle spezifiziert, d.h. einzelne oder Gruppen von Anforderungen sollen in bestimmten Testfällen überprüft werden. Zu diesem Zweck muss das Entwicklerteam Testabläufe definieren, die anschließend in der Analysis Execution Phase durchgeführt werden. Bei der Definition von Testabläufen können die Szenarien in der Anforderungsspezifikation wiederbenutzt werden. Die

Testabläufe sollten so erstellt werden, dass nicht nur eine Überprüfung, sondern auch eine Bewertung (Grad der Erfüllung oder definierte Punkte-Skala) vorgenommen werden kann, um die letztendliche Entscheidung zu treffen.

Analysis Execution: In dieser Phase werden die vordefinierten Testabläufe auf die vorausgewählten Komponenten angewendet. Der ausführende Entwickler muss dabei eine Bewertung abgeben. Um Biasing zu vermeiden, sollten mindestens zwei verschiedene Entwickler die Komponenten in unterschiedlicher Reihenfolge testen.

Candidate Selection: Mittels der Ergebnisse der Analysis Execution und den gewichteten Anforderungen, muss eine Komponente für den Einsatz ausgewählt werden. Diese Aufgabe ist auch unter „Multiple-Criteria-Decision-Making“ (MCDM) bekannt und es sind entsprechende Lösungen verfügbar (Bspw. AHP, siehe [Saat90]).

3 COTS-Auswahl für Software Produktfamilien

In dem folgenden Abschnitt wird der Einsatz von Komponentenauswahl-Prozessen bei der Entwicklung von Software Produktfamilien beschrieben. Die Wechselwirkungen zwischen Komponentenauswahl-Prozessen und der Produktfamilien-Entwicklung wird unter Verwendung des vorgestellten Rahmenwerks erläutert. Die sich ergebenden Konsequenzen der COTS-Auswahl werden diskutiert, um die weiteren Arbeiten zu motivieren.

3.1 Das Rahmenwerk im Produktfamilienkontext

Bei der Planung einer Produktfamilie werden die Anforderungen an die gesamte Produktpalette, als auch die Anforderungen an die einzelnen Anwendungen zunächst gesammelt. Die Realisierung der Anforderungen wird unter Kosten- und Aufwandsabschätzungen auf das DE und AE aufgeteilt. Techniken der Zuordnungen von Funktionalitäten auf die Domäne oder die spezifischen Anwendungen sind im Produktfamilien Kontext bekannt (z.B. siehe [DeSc99]).

Der Einsatz der Komponentenauswahl in der Produktfamilien-Entwicklung kann stattfindet, sobald die groben Anforderungen an die Komponente feststehen[3]. Die Auswahl von Komponenten findet im Anschluss an die Analysephase der Softwareentwicklung statt. Für die Einordnung in die Produktfamilien-Entwicklung bedeutet dies eine Verzahnung des Auswahl-Prozesses mit der Domänenanalyse und dem Design der Domäne, beziehungsweise mit der Anforderungserfassung einer kundenspezifischen Anwendung und dem Design der Anwendung. Abbildung 2 verdeutlicht diese Verzahnung unter Verwendung des Rahmenwerks als „Platzhalter“ für einen konkreten Prozess. Die dargestellten Schnittstellen werden in Abschnitt 3.2 genauer beschrieben.

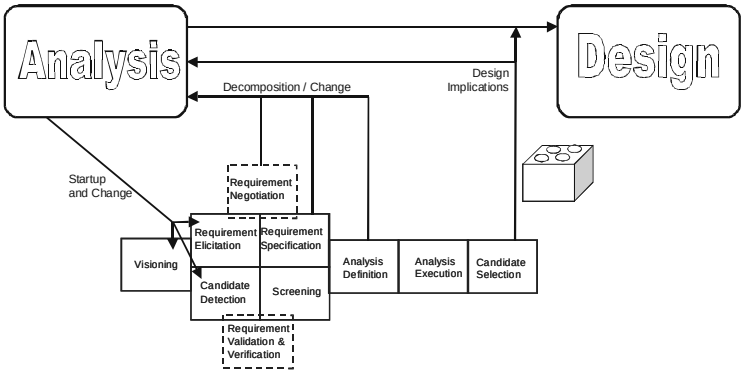


Abbildung 2: Das Rahmenwerk in der Produktfamilien Entwicklung

3.2 Auswirkungen und Konsequenzen der Komponentenauswahl

Die Komponentenauswahl wird durch die Zielvorgabe der Komponente und ersten groben Anforderungen initiiert. Die Domänenanalyse oder die Anforderungen an die kundenspezifische Anwendung liefert den Phasen Visioning, Requirement Elicitation und Candidate Detection des Rahmenwerks Eingaben (z.B. Ziele und Dokumente). Spätere Anpassungen in der Domänen- oder Anwendungs-Analyse führen zu Änderungen in genau diesen Phasen, d.h. Änderungen sollten eine erneute Betrachtung grober Anforderungen und eine Erweiterung der Marktstudie beinhalten (Abbildung 2, Pfeil „Startup and Change“).

[3] Die Existenz adäquater Komponenten wird vorausgesetzt. Falls mögliche Komponenten die Anforderungen nur unzureichend erfüllen, muss die Komponente angepasst werden.

Die Erfassung potentieller Komponenten-Kandidaten und den damit verbundenen Erfahrungen in der Screening-Phase können verborgene Anforderungen zu Tage fördern. Die Dekomposition der groben Anforderungen in eine genauere Anforderungs-Spezifikation kann zu neuen Anforderungen und Aufdeckungen von Widersprüchen innerhalb der Anforderungen führen. Eine so erkannte Änderung des Anforderungsmodells beeinflusst einerseits die lokalen Artefakte eines Komponentenauswahl-Prozesses, andererseits müssen diese Änderungen in das Model der Domäne und der Anwendung propagiert werden (Abbildung 2, Pfeil „Decomposition/Change“).

Die letztendliche Auswahl einer Komponente der Domäne beeinflusst direkt das Design der generischen Architektur, da diese Komponente in die Architektur integriert werden muss. Andererseits kann eine Komponente technische (Corba-Schnittstellen, etc.) Bedingungen mit sich führen, die das weitere Design beeinflussen. Diese technischen Bedingungen müssen auch in das Anforderungsmodell eingebracht werden, da diese Bedingungen zusätzliche Kosten implizieren können, die sich auf (z.B. finanzielle) Anforderungen auswirken (Pfeil „Design Implications“ in Abbildung 2).

Analog kann für die Auswahl einer anwendungsspezifischen Komponente argumentiert werden, allerdings können Änderungen an den Anwendungsanforderungen Rückwirkungen auf die Analyse oder das Design der Domäne auswirken:

- Ist eine wichtige Komponente nur für eine spezielle Architektur (z.B. COM-basiert) verfügbar und muss daher die generische Architektur geändert werden?
- Müssen durch die Auswahl dieser anwendungsspezifischen Komponente weitere Dienste in der Domäne erbracht werden?
- Ist es wirtschaftlicher diese Anwendung unabhängig von dem Domänen-Design zu realisieren, um eine Anpassung der Domäne zu vermeiden?

Diese Zusammenhänge zwischen den Komponenten der generischen Architektur und den anwendungs-spezifischen Komponenten sind mit dem Pfeil „Strong Interdependencies“ in Abbildung 3 dargestellt. Falls die Auswahl der Komponenten das System negativ beeinflusst (z.B. niedrige Performance), kann dies bedeuten, dass eine oder mehrere Komponenten falsch gewählt wurden (Kompatibilitäts-Anforderungen wurden nicht beachtet oder nicht erkannt). In diesem Fall müssen eine oder mehrere der beeinflussenden Komponenten neu ausgewählt werden. Unterliegt selbst die beste der gefundenen Komponenten-Konfigurationen solchen Wechselwirkungen, stellt sich die Frage, ob diese Lösung akzeptabel ist oder bestimmte Komponenten des Systems neu entwickelt, bzw. angepasst werden müssen.

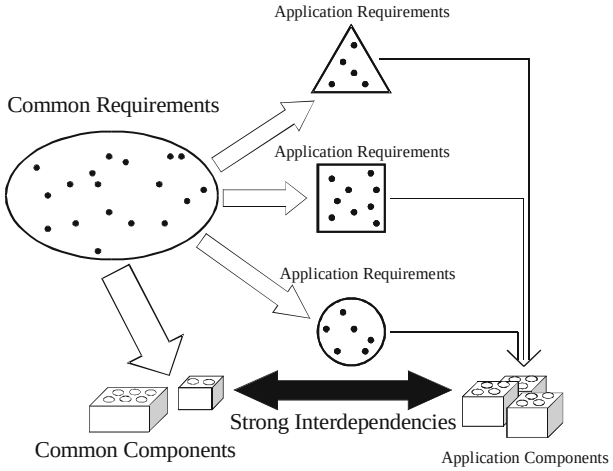


Abbildung 3: Komponentenermittlung bei Software-Produktfamilien

3.3 Änderungsunterstützung in Komponentenauswahl-Prozessen für Produktfamilien

Durch die in Abschnitt 3.2 gezeigten Auswirkungen der Komponentenauswahl auf die Produktfamilienentwicklung wird deutlich, dass die Auswahl von Komponenten selbst Änderungen unterliegt. Dies ist gerade in den frühen Stadien einer Produktfamilie der Fall, wenn die Identifizierung neuer kundenspezifischer Anforderungen Adaption des Domänen- und der Anwendungsdesigns nötig machen. Zusammenfassend muss ein Komponentenauswahlprozess folgende Änderungen unterstützen:

- Die Anforderungen an die Komponenten müssen in einer leicht abänderbaren Form vorliegen, da diese gerade in den frühen Planungsphasen einer Produktfamilie häufig Änderungen unterliegen können. Dies hat vor allen Dingen zwei Gründe: Zum einen ist die Abbildung von Anforderungen auf Komponenten-Funktionalitäten selten eine hundertprozentige Abdeckung. Daher müssen andere Komponenten die nicht erbrachten Funktionalitäten übernehmen oder Funktionalitäten abgeben, falls eine andere Komponente mehr Anforderungen abdeckt, als ursprünglich erwartet. Zum anderen beeinflusst die Auswahl einer Komponente das Design, was (architekturelle) Anforderungen an die weiteren Komponenten impliziert.

- Die Komponenten-Testfälle müssen zu den Komponenten-Anforderungen in Beziehung gesetzt werden, da die Änderungen einer Anforderungen die abgeleiteten Testfälle beeinflusst. Die Adaption eines Testfalls kann u.U. bis zu eine Aufhebung des Testfalls führen, was eine Reduktion des Testaufwands bedeutet.

Daher darf das Ergebnis der Komponentenauswahl nicht nur aus einer resultierenden Komponente bestehen, sondern es müssen vielmehr die wichtigen Artefakte und ihre Interrelationen identifiziert worden sein. Diese Artefakte müssen so aufbereitet sein, dass ihre Inhalte nachvollziehbar und zur weiteren Verwendung einsetzbar sind. Die Schnittstellen und Artefakte des Komponentenauswahl-Prozesses mit dem Produktfamilien-Entwicklungsprozess müssen genauer untersucht werden, um eine effektives, toolunterstütztes Anforderungsmanagement für die Komponentenauswahl entwickeln zu können.

4 Zusammenfassung und Ausblick

Die Verwendung von vorhandenen Komponenten ist ein wichtiger Bestandteil von Software-Entwicklungsprozessen. Da die Integration vorhandener Komponenten in den letzten Jahren zunehmend erleichtert wurde, gewinnt die Auswahl von Prozessen zunehmend an Bedeutung. Die Auswahl einer Komponente sollte basierend auf einen definierten Prozess erfolgen, um die Nachvollziehbarkeit der Ergebnisse zu gewährleisten und eine Verbesserung des Vorgehens zu ermöglichen.

Da die Definition eines Prozesses stets Domänenwissen benötigt, wurde ein Rahmenwerk für die Auswahl von Komponenten beschrieben. Durch ein solches Rahmenwerk wird eine Abstraktionsebene eingefügt, die den Vergleich existierender Ansätze zur Auswahl von Komponenten ermöglicht. Durch den Vergleich existierender Ansätze kann ein „best practice“ ermittelt werden, was die Definition neuer Komponentenauswahl-Prozesse erleichtert.

Die Auswahl von Komponenten in der Produktfamilienentwicklung beeinflussen maßgeblich die Analyse und das Design im Domain- und Application-Engineering. Wechselwirkungen von Domänen- und Anwendungskomponenten müssen berücksichtigt werden und führen zu einer wiederholten Anwendung der Komponentenauswahl. Ein Komponentenauswahl-Prozess muss daher geeignete Strukturen zur Wiederaufnahme des Auswahlprozesses bereitstellen und die Evolution der verwendeten Artefakte unterstützen.

Die Unterstützung der Komponentenauswahl im Produktfamilien Kontext ist Gegenstand aktueller Forschungsarbeiten. Zu diesem Zweck wird ein konkreter Prozess zur Auswahl von Komponenten definiert. Die Durchführung eines solchen Prozesses muss durch die Entwicklungsumgebung unterstützt werden. Daher haben wir uns die Entwicklung eines werkzeuggestützten Komponentenauswahl-Prozesses zum Ziel gesetzt.

Bibliography

[BrWa96] Brown, Alan W.; Wallnau, Kurt C.; “Engineering of Component-Based Systems”, Proceedings of the 2nd IEEE International Conference on Engineering of Complex Systems, 1996, 414 - 422.

[CaLo00] Carney, David; Long, Fred; “What Do You Mean by COTS”, IEEE Software March/April 2000, 83 – 86.

[ClNo99] Clements, Paul; Northrop, Linda; “A Framework for Software Product Line Practice – Version 2.0”, Technical Report, Software Engineering Institute, 1999. (via http://www.sei.cmu.edu/plp/framework.html available)

[CzEi00] Czarnecki, K.; Eisenecker, U.W.; “Generative Programming”, Addison-Wesley 2000.

[DeSc99] DeBaud, Jean-Marc; Schmid, Klaus; “A Systematic Approach to Derive the Scope of Software Product Lines”, Proceedings of ICSE 99, 34 – 43.

[DeVi97] Dean, John C.; Vigder, Mark R.; “System Implementation Using Commercial Off-The-Shelf (COTS) Software”, Proceedings of the 1997 Software Technology Conference (STC 1997).

[GAOc95] Garlan, David; Allen, Robert; Ockerbloom, John; “Architectural Mismatch: Why Reuse Is So Hard”, IEEE Computer November 1995, p. 17 – 26.

[Kont96] Kontio, Jyrki; “A Case Study in Applying a Systematic Method for COTS Selection”, Proc. ICSE 1996, 201 – 209.

[MaNc98] Maiden, Neil A.; Ncube, Cornelius; “Acquiring COTS Software Selection Requirements”, IEEE Software March/April 1998, 46 - 56.

[Pohl96] Pohl, Klaus; “Process-Centered Requirements Engineering”, Advanced Software Development Series, J. Wiley & Sons Ltd., Taunton, England, 1996.

[PWD*99] Pohl, Klaus; Weidenhaupt, Klaus; Dömges, Ralf; Haumer, Peter; Jarke, Matthias; Klamma, Ralf; “PRIME – Towards Process-Integrated Modeling Environments”, ACM Transactions on Software Engineering and Methodology, Vol. 8, No. 4, October 1999, 343 – 410.

[Saat90] Saaty, Thomas L.; “The Analytic Hierarchy Process”, McGraw-Hill, New York, 1990.

[SuKn96] Sullivan, Kevin J; Knight, John C.; “Experience Assessing an Architectural Approach to Large-Scale Systematic Reuse”, Proceedings 18th ICSE 1996, 220 - 229.

[Voas98] Voas, Jeffrey M. „COTS Software: The Economical Choice?“, IEEE Computer March 1998 15(2), 16 - 19.

[WPJH98] Weidenhaupt, Klaus; Pohl, Klaus; Jarke, Matthias; Haumer, Peter: “Scenario Usage in System Development: A Report on Current Practice“, In: IEEE Software, March 1998, 34 – 45.

Reengineering und Testen für Produktfamilien

Herleitung der Merkmal-Komponenten-Korrespondenz mittels Begriffsanalyse

Thomas Eisenbarth, Rainer Koschke, Daniel Simon

Bauhaus Stuttgart
Universität Stuttgart, Breitwiesenstr. 20-22, 70565 Stuttgart, Deutschland
{eisenbts, koschke, simondl}@informatik.uni-stuttgart.de

1. Einführung

Gemäß Martinez [7] waren die erfolgreichen Beispiele von Produktfamilien, die innerhalb von Motorola erzeugt wurden, ursprünglich überwiegend einzelne eigenständige Produkte. Erst im Laufe der Zeit erwuchs aus den ersten Produkten eine Architektur für eine Produktfamilie. Im Allgemeinen muss also die explizite Einführung des Produktfamilienansatzes den bereits existierenden Code einbeziehen. Reverse Engineering kann dabei helfen, die Komponenten zu identifizieren und zu analysieren sowie die tatsächliche Architektur aus dem System abzuleiten. Zu Anpassungszwecken wird der Code anschließend gegebenfalls geändert (Reengineering) oder gekapselt (Wrapping). Die oben genannten Varianten des Reverse Engineerings sind in Prozessmodellen des Produktlinienansatzes eher in späteren Phasen angesiedelt. Allerdings fordern Bayer et al. statt dessen zurecht eine sehr frühe Integration des Reverse Engineerings im Produktfamilienansatz [1]. Reverse Engineering wird auch benötigt, um erste, grobe Informationen über die existierenden Systemkomponenten an die Produktfamilienanalytiker zu liefern, um Machbarkeit und Kosten von Alternativen zur Entwicklung einer bestimmten Produktfamilie besser abschätzen zu können. Diese Information wird frühzeitig benötigt.

Ein wesentlicher Bestandteil einer existierende Merkmale integrierenden Produktlinienanalyse, der durch Reverse Engineering ermittelt werden kann, ist die sogenannte **Merkmal-Komponenten-Korrespondenz (MKK)**, die aussagt, welche Komponenten zur Implementierung eines bestimmten Merkmals benötigt werden. Da diese Information benötigt wird, um frühzeitig Alternativen abwägen zu können, kommt ein vollständiges und damit zeitaufwändiges Reverse Engineering des Systems nicht in Betracht. Insbesondere wird eine Entscheidung für eine bestimmte Alternative in vielen Fällen eine Konsolidierung auf bestimmte ökonomisch wichtige Kernelemente bedeuten und somit den Ausschluss einiger, weniger zentralen Komponenten zur Folge haben, in deren Reverse Engineering dann umsonst investiert worden wäre.

Statt dessen muss also mit möglichst einfachen Mitteln sehr schnell Aufschluss über die MKK gegeben werden. Dazu nennt der Produktfamilienanalytiker dem Reverse Ingenieur die relevanten Merkmale, für die die MKK ermittelt werden soll. Sobald auf Grundlage der vom Reverse Engineer ermittelten Merkmal-Komponenten-Korrespondenz und weiterer Information die Entscheidung für die nähere Betrachtung ausgewählter Komponenten gefallen ist, können dann aufwändigere Untersuchungen zur Qualität gezielt und kosteneffektiv durchgeführt werden.

In diesem Artikel stellen wir eine vollautomatische und rasch durchführbare Technik zur Ermittlung der MKK vor, die sich auf dynamische Analyse (Ausführungs-Traces) und Begriffsanalyse stützt. Begriffsanalyse ist eine Technik zur Untersuchung binärer Relationen.

Es ist jedoch anzumerken, dass viele nicht-funktionale Anforderungen, z.B. zeitliche Anforderungen, nicht unbedingt einzelnen Komponenten zugeordnet werden können, so dass primär funktionale Merkmale mit der beschriebenen Technik erfasst werden. Allerdings ist es in manchen Fällen auch möglich, nicht-funktionale Anforderungen, wie z.B. Sicherheit, im Code herauszufaktorisieren und durch einzelne dedizierte Komponenten zu realisieren. So könnte es z.B. sein, dass alle Netzwerkzugriffe in genau einer Komponente zusammengefasst wurden, um darin sichere Verbindungen gewährleisten zu können.

1.1. Terminologie

Im Folgenden bezeichnen wir realisierte (sowohl funktionale als auch nicht-funktionale) Anforderungen als **Merkmale** (engl. *features*). **Komponenten** sind Berechnungseinheiten im Sinne der Software-Architektur. Einfache Komponenten sind zum Beispiel einzelne Funktionen oder existierende Module. Komplexere Komponenten sind Subsysteme oder ganze Commercial-Off-The-Shelf-Produkte (COTS) (siehe Abschnitt 3.1).

1.2. Einordnung in den Produktlinienprozess

Ein einfacher Prozess zum merkmal-basierten Reengineering für Produktlinien, der Kenntnis über Merkmale und Komponenten voraussetzt, kann wie folgt beschrieben werden:

1. Die ökonomisch wichtigen Merkmale werden ermittelt.
2. Merkmal-Lokation: Die Merkmal-Komponenten-Korrespondenz wird ermittelt.
3. Eingrenzung auf bestimmte Merkmale und Komponenten anhand der MKK und ökonomischen Betrachtungen.
4. Komponenten-Bewertung: Die ausgewählten Komponenten werden näher analysiert (beispielsweise nach Wartbarkeit, Extrahierbarkeit oder Integrierbarkeit).
5. Alternativen werden abgewogen: Komponenten-Extraktion, Neuentwicklung/-einkauf oder Wrapping.
6. Ein Migrationsplan wird erstellt.

Die in diesem Artikel vorgestellte Technik unterstützt die Merkmal-Lokation.

1.3. Übersicht über die Merkmal-Lokation

Die hier vorgestellte Technik bedient sich der von ei-

nem Profiler erzeugten Ausführungs-Traces für verschiedene Verwendungsszenarien. Ein Szenario repräsentiert dabei die Verwendung eines Merkmals und liefert die Funktionen, die für dieses Merkmal aufgerufen wurden. Die Ausführungs-Traces für alle Szenarien werden anschließend der Begriffsanalyse unterzogen. Die Begriffsanalyse gibt Aufschluss über Beziehungen zwischen den Merkmalen und den dazu ausgeführten Komponenten sowie über Abhängigkeiten zwischen Merkmalen und auch zwischen Komponenten selbst.

Abschnitt 2 erläutert die Begriffsanalyse im Allgemeinen. Abschnitt 3 legt dar, wie Begriffsanalyse zur Herleitung der MKK verwendet werden kann, und Abschnitt 4 beschreibt eine Fallstudie, bei der die Technik zum Einsatz kam.

2. Begriffsanalyse

Der mathematisch-theoretische Grundstein der Begriffsanalyse wurde von Birkhoff bereits 1940 gelegt. Seit der Einführung der Begriffsanalyse im Software Engineering – hauptsächlich vorangetrieben von Snelting [4] – wurde die Technik für verschiedene Aufgaben verwendet, wie z.B. zur Restrukturierung von Klassenhierarchien, Herleitung von Konfigurationsstrukturen aus Präprozessoranweisungen und Erkennung logischer Module. Die Begriffsanalyse basiert auf einer Relation R zwischen einer Menge von Objekten O und einer Menge von Attributen A, d.h. $R \subseteq O \times A$. Diese Begriffsbildung besteht bereits seit 1940 und darf nicht mit objektorientierten Konzepten vermengt werden. Das Tripel $C = (O, A, R)$ wird **formaler Kontext** genannt. Für eine Menge von Objekten $O \subseteq O$ ist die Menge **gemeinsamer Attribute** σ definiert als:

$$\sigma(O) = \{a \in A \mid \forall (o \in O)(o, a) \in R\}$$

Entsprechend wird für eine Menge von Attributen $A \subseteq A$ die Menge ihrer **gemeinsamen Objekte** τ definiert als:

$$\tau(A) = \{o \in O \mid \forall (a \in A)(o, a) \in R\}$$

In Abschnitt 3.1 werden wir den formalen Kontext für die Begriffsanalyse zur Herleitung der MKK wie folgt festlegen:

- als Objekte werden Komponenten verwendet,
- als Attribute werden Merkmale verwendet,
- das Paar (*Komponente K*, *Merkmal M*) ist in der Relation, wenn *K* zur Ausführung von *M* verwendet wird.

Zunächst werden wir jedoch als ein abstraktes Beispiel die binäre Relation zwischen beliebigen Objekten und Attributen verwenden, die in Tabelle 1 angegeben ist. Ein Objekt o_i hat das Attribut a_j, wenn ein × in Zeile i und Spalte j der Tabelle 1 steht (das angegebene Beispiel geht auf Lindig und Snelting zurück [4]). Für diese Tabelle, auch **Relationentabelle** genannt, gelten zum Beispiel die beiden folgenden Gleichungen:

$$\sigma(\{o_1\}) = \{a_1, a_2\} \quad \text{und} \quad \tau(\{a_7, a_8\}) = \{o_3, o_4\}$$

Ein Paar (O, A) wird **Begriff** genannt, falls

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
o_1	×	×						
o_2			×	×	×			
o_3			×	×		×	×	×
o_4			×	×	×	×	×	×

Tabelle 1. Beispielrelation.

$A = \sigma(O) \wedge O = \tau(A)$ gilt, d.h. alle Objekte weisen alle Attribute auf. Für einen Begriff $c = (O, A)$ wird O als der **Extent**, angegeben als *extent*(c), und A als der **Intent** bezeichnet, angegeben als *intent*(c). Anschaulich korrespondiert ein Begriff mit einem maximal großen mit Kreuzen gefüllten Rechteck (modulo Spalten- und Zeilenpermutationen) in der Tabelle für die Relation. Beispielsweise enthält Tabelle 2 die folgenden Begriffe für die Relation in Tabelle 1.

C_1	$(\{o_1, o_2, o_3, o_4\}, \emptyset)$
C_2	$(\{o_2, o_3, o_4\}, \{a_3, a_4\})$
C_3	$(\{o_1\}, \{a_1, a_2\})$
C_4	$(\{o_2, o_4\}, \{a_3, a_4, a_5\})$
C_5	$(\{o_3, o_4\}, \{a_3, a_4, a_6, a_7, a_8\})$
C_6	$(\{o_4\}, \{a_3, a_4, a_5, a_6, a_7, a_8\})$
C_7	$(\emptyset, \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\})$

Tabelle 2. Begriffe für Tabelle 1.

Die Menge aller Begriffe einer gegebenen Relation bildet eine partielle Halbordnung mittels:

$(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2$ oder äquivalent mit
$(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow A_1 \supseteq A_2$.

Falls $c_1 \leq c_2$ gilt, dann wird c_1 ein **Unterbegriff** von c_2 und c_2 ein **Oberbegriff** von c_1 genannt. Beispielsweise gilt $(\{o_2, o_4\}, \{a_3, a_4, a_5\}) \leq (\{o_2, o_3, o_4\}, \{a_3, a_4\})$ in Tabelle 2.

Die Menge aller Begriffe L mit der angegebenen partiellen Halbordnung ergeben einen vollständigen Verband, **Begriffsverband** genannt:

$$L(C) = \{(O, A) \in 2^O \times 2^A \mid A = \sigma(O) \wedge O = \tau(A)\}$$

In diesem Verband wird das **Infimum** zweier Begriffe berechnet, indem man deren Extents schneidet:

$$(O_1, A_1) \wedge (O_2, A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2))$$

Das Infimum beschreibt die Menge gemeinsamer Attribute zweier Mengen von Objekten. In ähnlicher Weise wird das **Supremum** berechnet, indem man die Intents schneidet:

$$(O_1, A_1) \vee (O_2, A_2) = (\tau(A_1 \cap A_2), A_1 \cap A_2)$$

Das Supremum beschreibt die Menge gemeinsamer Objekte, die auf zwei Mengen von Attributen passen

Graphisch kann der Begriffsverband für die Beispielrelation in Tabelle 1 als Graph dargestellt werden, dessen Knoten Begriffe sind und dessen Kanten die Unter-/Ober-

begriffsbeziehung < reflektieren, wie in Abbildung 1 gezeigt. Der allgemeinste Begriff wird als **Top-Element** bezeichnet und durch ⊤ repräsentiert. Der speziellste Begriff wird als **Bottom-Element** bezeichnet und durch ⊥ dargestellt.

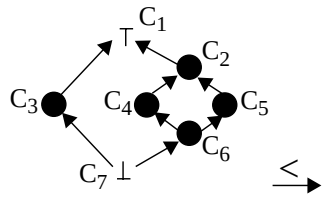


Abbildung 1. Begriffsverband für Tabelle 1.

Die Kombination der graphischen Darstellung in Abbildung 1 und der Begriffsinhalte in Tabelle 2 ergeben den Begriffsverband. Die gesamte Information lässt sich in einer leicht lesbaren Form darstellen, indem ein Knoten im Begriffsverband nur mit dem Attribut $a \in A$ markiert wird, für das der repräsentierte Begriff der allgemeinste Begriff ist, der a in seinem Intent hat. Analog wird ein Knoten markiert mit einem Objekt $o \in O$, wenn er den speziellsten Begriff repräsentiert, der o in seinem Extent hat. Das eindeutige mit a markierte Verbandselement μ ist somit:

$$\mu(a) = \bigvee \{c \in L(C) | a \in intent(c)\} \quad \textbf{(1)}$$

Das mit o markierte Element γ ist:

$$\gamma(o) = \bigwedge \{c \in L(C) | o \in extent(c)\} \quad \textbf{(2)}$$

Der äquivalente Graph für Abbildung 1 mit der angeführten Markierungsstrategie wird in Abbildung 2 gezeigt. Die Begriffsinhalte eines Knotens N in dieser Darstellung lassen sich wie folgt herleiten:

- die Objekte ergeben sich durch alle Objekte bei und unterhalb von N,
- die Attribute ergeben sich durch alle Attribute bei und oberhalb von N.

Beispielsweise repräsentiert der mit o_2 und a_5 markierte Knoten in Abbildung 2 dem Begriff $(\{o_2, o_4\}, \{a_3, a_4, a_5\})$.

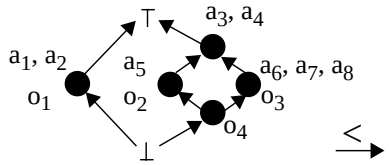


Abbildung 2. Vereinfachte Version zu Abbildung 1.

3. Merkmal-Komponenten-Korrespondenz

Zur Herleitung der Merkmal-Komponenten-Korrespondenz mittels Begriffsanalyse bedarf es der Definition des Kontextes (Objekte, Attribute, Relation) und der anschließenden Interpretation des ermittelten Begriffsverbands.

3.1. Kontext für Merkmale und Komponenten

Als Objekte für die Begriffsanalyse verwenden wir die Menge der Komponenten, als Attribute die Menge der Merkmale. Man beachte, dass bei einer umgekehrten Betrachtung nur der Begriffsverband invertiert und ansonsten die gleiche Information ermittelt wird.

Die Menge der relevanten Merkmale wird von Produktlinien-Experten zur Verfügung gestellt. Als Komponenten kommen folgende Alternativen in Betracht, abhängig davon, wie viel Wissen über die Architektur des Systems bereits vorhanden ist:

1. kohäsive Module und Subsysteme, wie sie durch den Software-Architekten festgelegt und dokumentiert oder durch einen Re-Ingenieur ermittelt wurden;
2. Module, wie sie im System vorhanden und durch die Programmiersprache definiert sind, von denen man aber nicht weiss, ob die darin deklarierten Elemente wirklich zusammengehörig sind; alternativ Dateien, falls die Programmiersprache kein Modulkonzept hat;
3. Unterprogramme des Systems, d.h. Funktionen und Prozeduren.

Im Idealfall wird man auf Alternative (1) zurückgreifen, sofern eine geeignete aktuelle Dokumentation existiert. Das System jedoch vorher vollständig zu analysieren, um zu den kohäsiven Modulen und Subsystemen zu kommen, die für die Begriffsanalyse als Objekte dienen können, ist vermutlich kein kosteneffektives Vorgehen. Erst später, wenn die Analyse ergeben hat, dass die Komponenten wirklich gebraucht werden, wird sich im Allgemeinen ein Reengineering lohnen (um kohäsive Module einfach erkennen zu können, wurden von uns eine Reihe semi-automatischer Analysen entwickelt [6]).

Alternative (2) eignet sich, wenn man den Programmierern des Systems weitgehend vertrauen kann. In allen anderen Fällen wird man auf Alternative (3) zurückgreifen müssen. Die Begriffsanalyse kann jedoch in diesem Fall über die Merkmal-Unterprogramm-Korrespondenz hinaus auch Hinweise auf die Zusammengehörigkeit von Unterprogrammen – somit auf komplexere Komponenten – geben.

Die für den Kontext zur Begriffsanalyse notwendige Relation ist definiert als:

> $(K, M) \in R$ g.d.w. Komponente K für Merkmal M ausgeführt wird; mit anderen Worten, wenn das Merkmal aktiviert wird, wird die Komponente ausgeführt.

Zur Ermittlung dieser Relation wird eine Menge von Verwendungsszenarien erstellt und angewandt, so dass ein Szenario jeweils ein bestimmtes Merkmal und so wenig wie möglich andere Merkmale ausnutzt. Der Ausführungs-Trace für den Programmlauf für ein einzelnes Verwendungsszenario liefert dann die ausgeführten Komponenten. Somit ergibt jeder individuelle Programmlauf genau eine Spalte in der Relationentabelle.

Falls unter einer Komponente nicht nur ein bestimmtes Unterprogramm, sondern eine Menge von Unterprogrammen und/oder Variablen zu verstehen ist, dann gilt eine Komponente als ausgeführt, wenn mindestens eines ihrer Unterprogramme ausgeführt wurde und/oder eine seiner Variablen referenziert wurde. Den Aufruf einer Funktion

kann man mittels eines Profilers ermitteln, den Zugriff auf eine Variable protokollieren jedoch einfache Profiler üblicherweise nicht. Statt einen besseren Profiler oder gar Code-Instrumentierung zu verwenden, kann man sich auch mit einer einfachen statischen Abhängigkeitsanalyse behelfen: Man betrachte für die ausgeführten Funktionen alle von ihnen direkt verwendeten Variablen als referenziert (die transitiv referenzierten ergeben sich, da alle ausgeführten Funktionen einbezogen werden). In diesem Falle entgehen einem jedoch die über Aliase manipulierten Variablen; außerdem werden auch solche Variablen fälschlicherweise einbezogen, deren Zugriff auf einem Pfad liegt, der gar nicht ausgeführt wird. Für eine erste einfache Analyse kann man Variablenzugriffe auch ignorieren.

3.2. Interpretation des Begriffsverbandes

Die Begriffsanalyse angewandt auf den im letzten Abschnitt eingeführten Kontext liefert einen Verband, aus dem interessante Beziehungen abgeleitet werden können. Diese Beziehungen lassen sich vollautomatisch aus dem Verband herleiten und dem Analytiker präsentieren, so dass diesem der ihm möglicherweise zu theoretische Hintergrund verborgen werden kann.

Wie in Abschnitt 2 bereits anwendungsneutral beschrieben lassen sich aus dem in vereinfachter Form dargestellten Begriffsverband die folgenden Beziehungen direkt ablesen:

- Eine Komponente o ist notwendig für alle Merkmale, die im Verband bei und oberhalb von $\mu(o)$ (wie durch Gleichung (1) definiert) auftreten.
- Ein Merkmal a benötigt alle Komponenten, die im Verband bei und unterhalb von $\gamma(a)$ (wie durch Gleichung (2) definiert) auftreten.
- Die Merkmale, zu denen zwei Komponenten o_1 und o_2 gemeinsam beitragen, werden durch $\mu(o_1) \wedge \mu(o_2)$ ermittelt; anschaulich ermittle man im Begriffsverband den nächsten gemeinsamen Knoten in Richtung des Top-Elements; alle Merkmale an und oberhalb dieses Knotens werden durch die zwei Komponenten gemeinsam implementiert.
- Die Komponenten, die für zwei Merkmale a_1 und a_2 gemeinsam gebraucht werden, sind durch $\gamma(a_1) \vee \gamma(a_2)$ beschrieben; anschaulich ermittle man im Begriffsverband den nächsten gemeinsamen Knoten in Richtung des Bottom-Elements; alle Komponenten an und unterhalb dieses Knotens werden für die zwei Merkmale gemeinsam vorausgesetzt.
- Komponenten, die für alle Merkmale benötigt werden, findet man im Bottom-Element.
- Merkmal, für die man alle Komponenten braucht, findet man im Top-Element.
- Enthält das Top-Element keine Merkmale, sind alle Komponenten im Top-Element unnütz.
- Enthält das Bottom-Element keine Komponenten, werden alle Merkmale im Bottom-Element vom System nicht implementiert.

Über diese Beziehungen zwischen Komponenten und Merkmale hinaus lassen sich weitere nützliche Aspekte zwischen Merkmalen einerseits und zwischen Komponenten andererseits ableiten:

- Gilt $\mu(o_1) < \mu(o_2)$ für zwei Komponenten o_1 und o_2, dann setzt o_2 die Komponente o_1 voraus.
- Gilt $\gamma(a_1) < \gamma(a_2)$ für zwei Merkmale a_1 und a_2, dann setzt Merkmal a_1 das Merkmal a_2 voraus.

Es sei angemerkt, dass letztere Beziehung zwischen Merkmalen nur sicher für das momentan untersuchte System gilt, dies aber nicht notwendigerweise generell der Fall sein muss, da die Beziehung aus einer spezifischen Implementierung abgeleitet wurde.

Die oben genannten Informationen können vom Re-Ingenieur an den Produktfamilien-Experten zurückgeliefert werden. Sobald eine Entscheidung für bestimmte Merkmale fällt, bilden alle dazu notwendigen Komponenten (wie sie einfach aus dem Begriffsverband abgeleitet werden können) den Ausgangspunkt für weitere statische Analysen zur Untersuchung der Qualität (wie z.B. Extrahierbarkeit, Wartbarkeit usw.) und des Aufwands möglicher weiterer Schritte (Wrapping, Neuentwicklung, Reengineering).

3.3. Implementierung

Die Implementierung des beschriebenen Ansatzes für Alternative (3) aus Abschnitt 3.1 ist überraschend einfach (sofern man bereits über ein Werkzeug zur Begriffsanalyse verfügt). Unsere prototypische Implementierung in einer Unix-Umgebung besteht aus folgenden Bestandteilen:

- Gnu-C-Compiler *gcc* zur Übersetzung des Systems (mit einer Option zur Erzeugung von Profiling-Information),
- Gnu-Objekt-Code-Viewer *nm* sowie ein kurzes Perl-Skript, um die im Objektcode definierten Funktionen zu ermitteln,
- Gnu-Profiler *gprof* sowie ein kurzes Perl-Skript, um die aufgerufenen Funktionen eines Ausführungs-Traces zu bestimmen,
- Begriffsanalyse-Werkzeug *concepts* [5],
- Grapheditor *Graphlet* [2], um den Begriffsverband zu visualisieren,
- sowie zwei weitere kurze Skripte, um die jeweiligen Formate von *concepts* und *Graphlet* zu konvertieren.

4. Fallstudie

Als Anwendungsbeispiel haben wir das System *xfig* (Version 3.2.1) untersucht. Der Übersichtlichkeit halber präsentieren wir hier nur einen Ausschnitt der gewonnenen Erkenntnisse.

Als Merkmale haben wir für xfig z.B. die Benutzung verschiedener Figuren (Kreis, Rechteck, Spline usw.), die

Editiermodi (Bewegen, Kopieren, Skalieren usw.) und die Attribute von Figuren (Farben, Füllmuster usw.) identifiziert. Durch verschiedene Verwendungsszenarien wurde die MKK mit der oben beschriebenen Methode hergeleitet. Eines der Experimente bestand aus folgenden Verwendungsszenarien:

A. Zeichnen eines Kreises durch Angabe des Radius.
B. Zeichnen eines interpolierten Splines.
C. Zeichnen eines Textobjektes.
D. Zeichnen eines Rechtecks mit abgerundeten Ecken.
E. Zeichnen eines Rechtecks, wobei vorher eine Farbe ausgewählt wurde.
F. Auswahl des Rechteck-Zeichen-Modus' sowie Farbselektion, jedoch ohne ein Objekt wirklich zu zeichnen.

Der Begriffsverband für dieses Experiment ist in Abbildung 3 zu sehen. Die Szenarien (A) bis (C) demonstrieren den einfachen Fall, dass die Merkmale genau auf einen Begriff abgebildet werden können. In Abbildung 3 repräsentieren die Knoten 19, 20 und 15 die Szenarien (A), (B) bzw. (C).

Die Abläufe (D) bis (F) überlagern die Merkmale *Rechteck*, *abgerundetes Rechteck* und *Farbauswahl* miteinander. Es ergibt sich so in diesem Bereich ein etwas komplexeres Bild. Die Merkmale lassen sich hier immer noch den Begriffen zuordnen, allerdings nicht mehr direkt am oberen Ende des Verbands. Knoten 7 repräsentiert das Zeichnen eines Rechtecks (D) (sowohl normal als auch abgerundet), Knoten 3 repräsentiert die Auswahl einer Farbe (E). Knoten 3 wurde sowohl unterhalb von Szenario (E) (Farbe auswählen und Rechteck zeichnen) als auch (F) (Farbe auswählen und Rechteck *nicht* zeichnen) angeordnet, wodurch das Merkmal *Farbauswahl* lokalisiert werden konnte.

Dieses Beispiel wurde anhand der Benutzeroberfläche von *xfig* erstellt. Merkmale, die sich nicht direkt auf dieser Oberfläche ausprägen (z.B. die Verwendung einer automatischen Speicherbereinigung), lassen sich auf diese einfache Weise nicht erfassen. Wir haben weitere Experimente durchgeführt, bei denen die Merkmale Figuren mit den Merkmalen Editiermodi und den Merkmalen Figurattribute kombiniert wurden. Diese ergaben Verbände, die sich trotz ihrer Komplexität leicht interpretieren ließen.

5. Schlussfolgerungen

Die Herleitung der MKK ist eine Aufgabe des Reverse Engineerings innerhalb des Produktlinienansatzes für existierende Systeme, die bereits in den frühen Phasen erledigt werden muss. Chen und Rajlich schlagen eine halb-automatische statische Merkmal-Lokation vor, bei der der Analytiker computer-gestützt den statisch hergeleiteten Abhängigkeitsgraphen absucht [3]. Da hier der Analytiker im Wesentlichen die Sucharbeit übernimmt, eignet sich diese Technik nicht für eine rasche Ermittlung der MKK.

Wilde und Scully verwenden dynamische Analyse zur Lokalisierung von Merkmalen wie folgt [8]:

1. Eine Eingabemenge *I* (*invoking input set*) wird identifiziert, die zur Verwendung des Merkmals führt.
2. Eine Eingabemenge *E* (*excluding input set*) wird identifiziert, bei der das Merkmal nicht ausgenutzt wird.
3. Das Programm wird für beide Eingabemengen separat ausgeführt.
4. Durch Vergleich der beiden Ausführungs-Traces ergeben sich die Komponenten, die das Merkmal implementieren.

Unsere Technik geht über die Technik von Wilde und Scully hinaus, indem wesentliche Beziehungen zwischen Komponenten und Merkmalen mittels Begriffsanalyse automatisch hergeleitet werden. Die Herleitung ergibt zusätzliche Abhängigkeiten, die Analytiker des Produktfamilienansatzes bei der Festlegung der Produktfamilie berücksichtigen müssen.

Die Technik ist nur für solche Merkmale wirklich geeignet, die sich auf einzelne Komponenten abbilden lassen; insbesondere für nicht-funktionale Merkmale ist dies oft nicht der Fall. Außerdem ist zur Gewinnung jedes Ausführungs-Traces in unserer einfachen Implementierung gegenwärtig ein eigener Programmlauf notwendig. Einfacher ließen sich Ausführungs-Traces gewinnen, wenn zu beliebigen Zeitpunkten die Aufzeichnung begonnen und beendet werden könnte. Diese Möglichkeit würde auch präzisere Traces liefern, da z.B. Aufrufe, die nur für den Programmstart benötigt werden, ignoriert werden könnten. Überdies hängt der Erfolg der Technik von der geschickten Auswahl von Verwendungsszenarien ab.

Referenzen

[1] Bayer, J., Girard, J.-F., Würthner, M, Apel, M., and DeBaud, J.-M., 'Transitioning Legacy Assets - a Product Line Approach', *Proceedings of the SIGSOFT Foundations of Software Engineering*, Toulouse, pp. 446-463, Association of Computing Machinery, 1999.

[2] Brandenburg, F.J., 'Graphlet', Universität Passau, http://www.infosun.fmi.uni-passau.de/Graphlet/.

[3] Chen, K. und Rajlich, V., 'Case Study of Feature Location Using Dependence Graph', Proc. of the 8th Int. Workshop on Program Comprehension, pp. 241-249, June 10-11, Limerick, Irland, IEEE Computer Society Press, 2000.

[4] Lindig, C. and Snelting, G., 'Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis', Proc. of the Int. Conference on Software Engineering, pp. 349-359, Boston, 1997.

[5] Lindig, C., Concepts, ftp://ftp.ips.cs.tu-bs.de:/pub/local/softech/misc

[6] Koschke, R., 'Atomic Architectural Component Recovery for Program Understanding and Evolution', Dissertation, Institut für Informatik, Universität Stuttgart, http://www.informatik.uni-stuttgart.de/ifi/ps/rainer/thesis, 2000.

[7] Staudenmayer, N.S. and Perry, D.E., '*Session 5: Key Techniques and Process Aspects for Product Line Development*', 10th International Software Process Workshop, June, Ventron FR, 1996.

[8] Wilde, N. and Scully, M.C., 'Software Reconnaissance: Mapping Program Features to Code', Software Maintenance: Research and Practice, vol. 7, pp. 49-62, 1995.

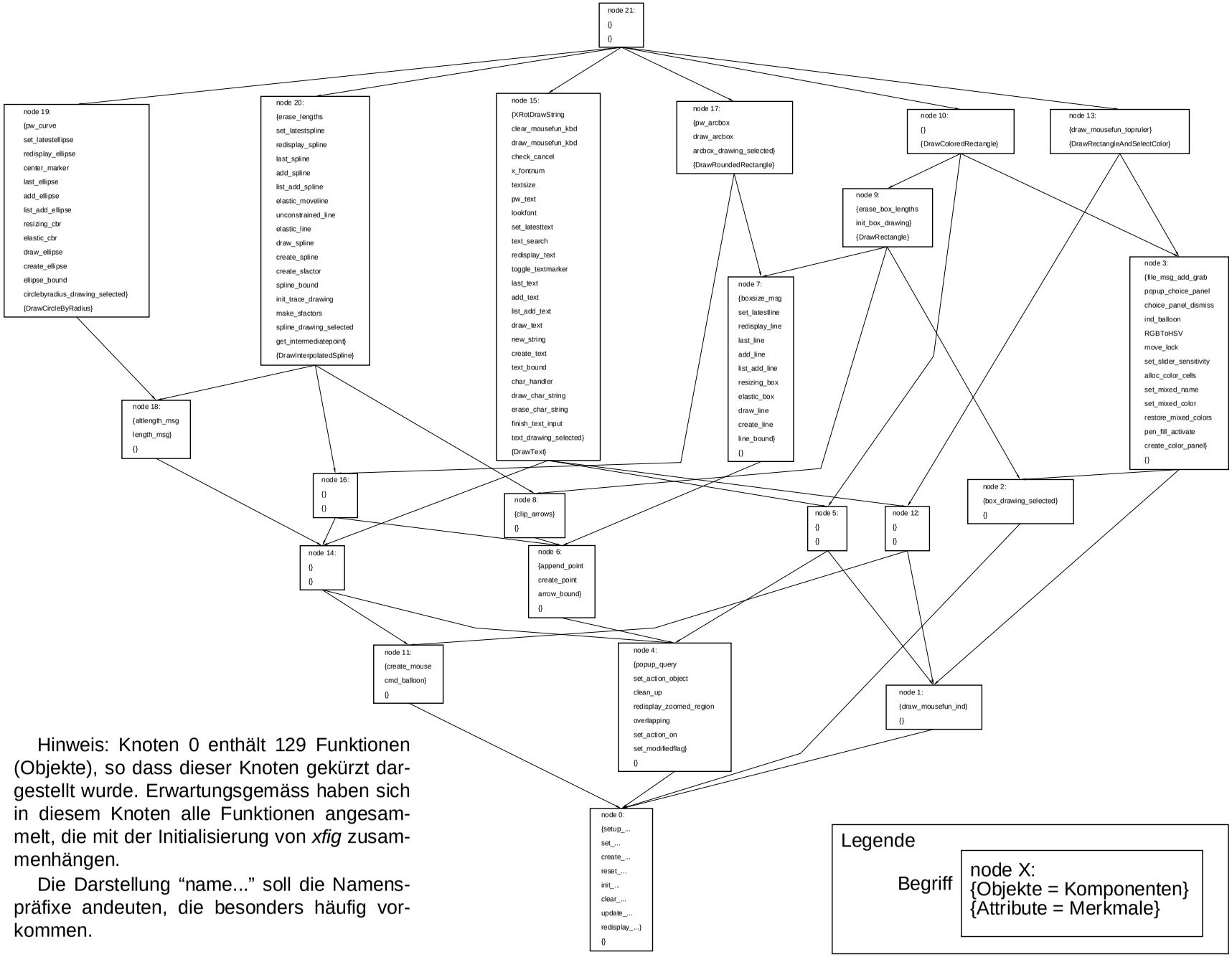


Abbildung 3. Ausschnitt des Begriffsverbands für *xfig*

Hinweis: Knoten 0 enthält 129 Funktionen (Objekte), so dass dieser Knoten gekürzt dargestellt wurde. Erwartungsgemäss haben sich in diesem Knoten alle Funktionen angesammelt, die mit der Initialisierung von *xfig* zusammenhängen.

Die Darstellung “name...” soll die Namenspräfixe andeuten, die besonders häufig vorkommen.

Reengineering von Metalevel-Abstraktionen mit Data-Mining-Methoden

Hans Wegener

Credit Suisse, Postfach 100, 8070 Zürich, Schweiz
Telefon: +41 (1) 334 66 51, Telefax: +41 (1) 334 50 60, Mail: hans.wegener@credit-suisse.ch

Zusammenfassung: Ein wichtiges Problem bei der Konstruktion von Metalevelarchitekturen ist die Identifikation eines stabilen Metamodells. Typischerweise erreicht man dies, indem die dem Problembereich zu Grunde liegende Ordnung im Modell abgebildet wird. Kenntnis über eine solche Ordnung gewinnt man aus Erfahrung oder durch eine Analyse der Domäne. Wir verwendeten Data Mining als Methode zum Reengineering des Zugriffskontrollsystems unseres Data Warehouse im Hinblick auf eine metadatengetriebene, komponentenorientierte Architektur. Unsere Erfahrung zeigt, dass Data Mining eine effiziente Methode zur Identifikation separierbarer Aspekte in Metalevelarchitekturen ist.

1. Einleitung

Metalevelarchitekturen bestehen aus zwei Schichten mit unterschiedlichen softwaretechnischen Eigenschaften, die im Interesse einer erhöhten Evolvierbarkeit gebildet werden. Die Basisebene evolviert verhältnismäßig häufig und schnell, während die Metaebene vergleichsweise sehr langsam evolviert. Alle Modellierung des Problembereichs geschieht auf der Metaebene, die eine sehr stabile Umgebung darstellt. Die Basisebene stellt den Lösungsbereich zur Verfügung [2].

Man kann Metalevelarchitekturen mit einer Wette vergleichen. Unter der Voraussetzung, dass

- viele Abhängigkeiten von Abstraktionen des Problembereichs hin zu Abstraktionen des Lösungsbereichs bestehen und
- die Struktur der Abstraktionen innerhalb der Lösungsbereich schnell evolviert,

dann ist es günstiger

- eine zusätzliche, datengetriebene Ebene einzuführen, die von den für einen Evolutionsschritt notwendigen Mechanismen abstrahiert und die Evolution selbst durch Datenmanipulation zu steuern

als

- die Evolutionsschritte manuell durch Aktualisierung der Abhängigkeiten zwischen Metaebene und Basisebene durchzuführen.

Man setzt also darauf, dass sich die wesentlichen Abhängigkeitsverhältnisse über die Zeit nur unwesentlich ändern. Unter diesen Umständen ist es wirtschaftlicher, eine komplexe Architektur einzuführen.

Zur Erreichung dieses Ziels stellt die Metaebene eine oder mehrere Abstraktionen bereit, die von Bedeutung für Entwickler oder Endanwender sind. Diese Aspekte werden in einer Weise modelliert, dass sie *auf dieser Ebene* weitgehend entkoppelt sind.

In der Realität sind die Aspekte aber stark mit Abstraktionen der Basisebene verkoppelt. Dieser Konflikt wird bereinigt, indem man einen Mechanismus bereitstellt, der die Semantik von Abstraktionen der Metaebene auf durch Abstraktionen der Basisebene realisiert. Dabei achtet man darauf, dass die Aspekte als entkoppelt behandelt werden können.

Dieses theoretische Bild weist eine angenehme Einfachheit auf und wirk ästhetisch sehr anziehend. Die Praxis aber hat gezeigt, dass dieses Ziel nur äußerst schwer zu erreichen ist. Es ist sehr schwer, gleichzeitig den Problembereich und die Evolutionsmuster im Lösungsbereich gut zu verstehen. Insbesondere folgende Voraussetzung ist häufig nicht gegeben: *Der Problembereich ist üblicherweise nicht hinreichend gut verstanden; folglich evolviert das assozierte Metamodell in nicht vernachlässigbarem Ausmaß.* Der Abbildungsmechanismus zwischen Meta- und Basisebene ist von noch höherer Komplexität ist als das Basismodell selbst. Deshalb entstehen signifikant hohe Kosten. (Es müssen nicht nur das Meta-

modell, sondern auch der Abbildungsmechanismus angepasst werden.) Dadurch trifft die oben erwähnte Wirtschaftlichkeitserwägung nicht mehr zu. *Ein wesentlicher Erfolgsfaktor für Metalevelarchitekturen ist deshalb die Möglichkeit zur Identifikation eines sehr stabilen Metamodells.*

Man kann auf Grund von Erfahrung eine Kenntnis des Problembereichs haben. Man kann das Wissen aber auch durch eine Analyse aufbauen. Dies ist von Interesse für Softwareprodukte, die sich bereits im Einsatz befinden und im Hinblick auf eine Metalevelarchitektur einem Reengineering unterzogen werden müssen. Dort muss man die bereits etablierte Semantik erhalten.

Die Credit Suisse befindet sich in einem Prozess, während dessen die existierenden Data Marts auf ein Data Warehouse mit komponentenorientierter Architektur und explizitem Metamodell migriert werden. Die Zugriffskontrolle ist ein Teil des Metamodells, das auf dem Common Warehouse Metamodel [3] basiert. Wir wollen die existierenden Zugriffsschutzsysteme in die neue Architektur einbinden und deren Semantik dabei erhalten.

In vielen Fällen war dies einfach möglich, jedoch widersetzte sich ein System unseren Bemühungen. Wir mussten einen Weg finden, der es uns nicht nur erlaubte, die existierenden Basislevelabstraktionen auf die neue Architektur zu übertragen, sondern auch auf Metalevelabstraktionen basierte, die sehr stabil sind.

Wir benutzten Data Mining um eine mögliche Separation of Concerns zu identifizieren, und machten sehr positive Erfahrungen mit dieser Methode. Dieser Artikel berichtet über unsere Erfahrungen und diskutiert den Wert des Data Mining im Software Engineering. Basierend auf unseren Ergebnissen treffen wir folgende Aussage:

Bei moderat-komplexem, nicht verstandenem Problembereich und Verfügbarkeit von Daten über die Abstraktionen des Lösungsbereichs ist Data Mining eine effiziente Methode zur Identifikation separater Abstraktionen des Problembereichs.

Dieser Artikel ist wie folgt aufgebaut: Im Abschnitt 2 beschreiben wir unsere Situation und die speziellen Umstände, die uns zu unserem Ansatz führten; die Abschnitte 3 und 4 beschreiben den Ansatz und unsere Ergebnisse; Abschnitt 5 fasst die Resultate zusammen und zieht Schlussfolgerungen.

2. Ausgangslage

Die Credit Suisse betreibt zur Zeit fünf unabhängig voneinander entwickelte Data Marts (Marketing, Zahlungsverkehr, Kredit- und Risikoüberwachung, Profitabilitätsanalyse, Datenpools). Im Interesse der Kostenreduktion wurde eine komponentenorientierte Zielarchitektur mit expliziten Metamodell definiert. Die Details der Architektur sind in [4] beschrieben.

Im Bereich Zugriffsschutz bestanden Probleme bei der Sicherstellung der Konsistenz, von einfacher Administrierbarkeit und schneller Evolvierbarkeit bei den ausgesprochenen Autorisierungen. Angesichts mehr als 10'000 Benutzern, einem Dutzend involvierter Komponenten und Security-Manager und der erwarteten Veränderungsgeschwindigkeit der Warehouse-Systeme erschien einzig eine metadatenbasierte Lösung sinnvoll.

Wir wählten einen rollenbasierten Ansatz zur Zugriffskontrolle und unterscheiden zwischen Benutzer- und Rechteadministration sowie Rollenmodellierung:

- Eine Rolle ist eine fachliche Beschreibung einer wohldefinierten (abtrennbaren) Benutzergruppe.
- Im Rahmen der Rollenmodellierung werden einer Rolle die Rechte zugewiesen, die zu einer angemessenen Erledigung der Rollenaufgaben nötig sind.
- Benutzeradministration beschränkt sich auf die Zuteilung und den Widerruf von Rollen an Benutzer.
- Rechteadministration beinhaltet die Kontrolle und Regulierung der mit den Rollen assoziierten Rechte im Interesse eines möglichst geringen Restrisikos.

Übersetzt in die oben erwähnten Begrifflichkeiten sind Rollen (z.B. Kreditportfolio-Manager) die Abstraktionen der Metaebene/Problembereich, die zur Umsetzung nötigen Mechanismen (z.B. relationale Sichten und Datenbankrollen) sind Abstraktionen der Basisebene/Lösungsbereich. Die Rechteadministration (z.B. Erweiterung einer Sicht um ein Attribut) vollzieht Evolutionsschritte auf der Basisebene nach, ohne Änderungen auf der Metaebene zu erfordern.

Die bereit in Betrieb befindlichen Systeme mussten auf diese Architektur angepasst werden. Wir mussten insbesondere folgende Aufgaben bewältigen:

- Identifikation eines Rollenmodells für das Gesamt-Warehouse,

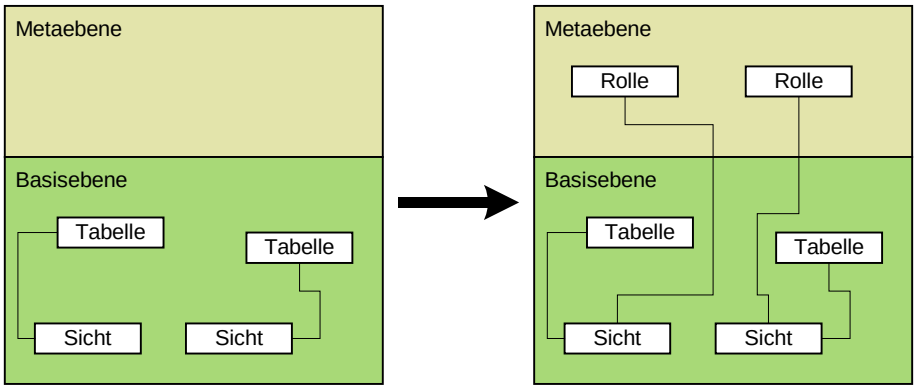


Abbildung 1: Die grundsätzliche Problematik beim Reengineering von Abstraktionen der Metaebene besteht darin, sie allein auf Grund der in der Basisebene vorhandenen Abstraktionen zu erstellen.

- Umsetzung mit Hilfe von Basislevelabstraktionen und
- Entwicklung von Software für die Abbildung von Metalevel- auf Basislevelabstraktionen.

Die Identifikation der Rollen bereitete Probleme. Vier der fünf Systeme besitzen klar umrissene Benutzergruppen, die sich in ganz bestimmten Organisationseinheiten befinden. Ein System jedoch hatte Benutzer quer über die gesamte Bank verteilt. Dieser Datenpool stellt Rohdaten in unaggregierter Form zur Verfügung, z.B. Personal- und Referenzdaten oder Kontoinformationen. Die meisten Benutzer waren individuell autorisiert worden, weshalb sich keine Struktur ausmachen ließ. Die einzige uns zur Verfügung stehende Information waren die existierenden Sichten und wem sie zugewiesen waren.

Aufgrund der Tatsache, dass die Benutzer so weit verstreut waren verbot sich eine (am Problembereich orientierte) Top-Down-Analyse. Dies hätte bedeutet, dass wir Rollen für große Teile der Bank hätten definieren müssen – eine zeitlich inakzeptable Vorstellung. Daher entschieden wir uns für eine (am Lösungsbereich orientierte) Bottom-Up-Analyse.

Weiterhin mussten wir die Granularität der identifizierten Rollen optimal bestimmen. Theoretisch hätte es ausgereicht, jedem Benutzer eine eigene Rolle zuzuweisen. Das hätte aber ein extrem unübersichtliches Rollenmodell zur Folge gehabt, mithin eine Einschränkung der Verständlichkeit. Eine zu grobe Rolleneinteilung wiederum hätte zu möglichen Sicherheitslöchern geführt und eine Erhöhung des Risikos bedeutet.

Auf Grund unserer architekturellen Entscheidungen mussten wir das Metamodell in einem Reengineeringprozess erstellen. Für letzteres mussten wir Metalevelkomponenten bereitstellen. Gleichzeitig sollte das Resultat möglichst stabil und verständlich sein sowie die bestehenden Autorisierungen originalgetreu abbilden. *Unsere Aufgabe war eine Separation der Abstraktionen des Problembereichs ohne hinreichende Kenntnis desselben, aber mit substanzieller Kenntnis des Lösungsbereichs.*

3. Lösung

Konkret analysierten wir die angelegten Datenbanksichten und die darauf autorisierten Benutzer. Daraus ermittelten wir Annahmen über mögliche Rollen, die dann in Zusammenarbeit mit Vertretern der geschäftlichen Seite validiert wurden.

Wir benutzten Data Mining als Methode für das Reengineering der Metalevelabstraktionen aus existierenden Autorisierungen, Sichten und Attributen. Wir gingen mit drei Grundannahmen in den Miningprozess:

1. Es gibt Korrelationen zwischen den Organisationseinheiten der Benutzer und den Sichten, auf die sie Zugriff haben. Hier sind Aufgaben bezogen auf die hierarchische Arbeitsaufteilung zu erledigen (vertikale Abhängigkeiten).
2. Es gibt Benutzer, für die keine solche Korrelation besteht. Hier sind Aufgaben gleicher Natur quer über die Organisation verteilt (horizontale Abhängigkeiten).
3. Es gibt eine gewisse Zahl untypischer Fälle (keine statistisch relevanten Abhängigkeiten).

Wir hatten zwei Informationsquellen, nämlich die vergebenen Autorisierungen und Daten über die Organisationszugehörigkeit der Benutzer.

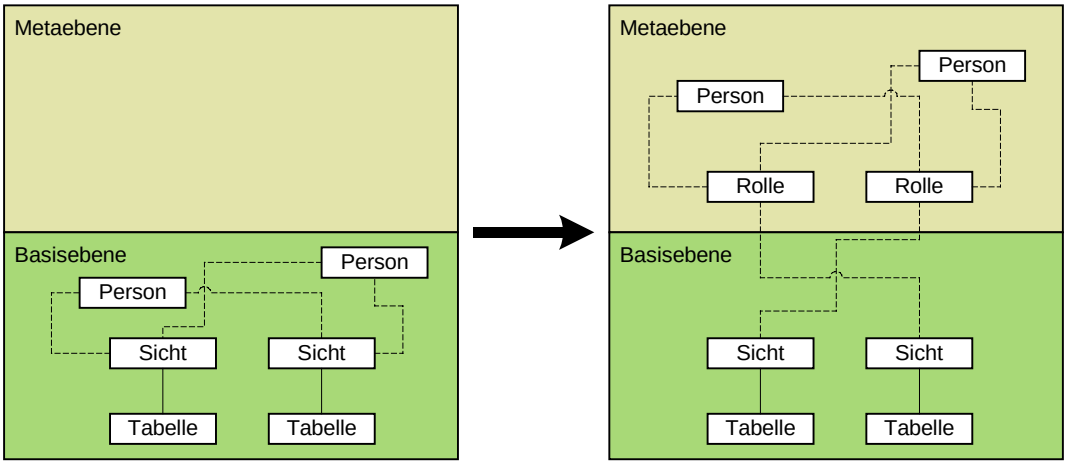


Abbildung 2: Das Reengineering des Zugriffskontrollmodells beinhaltete zwei Aspekte. Aus den vorhandenen Autorisierungsdaten (gestrichelte Linien links) mussten eine neue Abstraktion Rolle gebildet und Person als Abstraktion von der Basisebene in die Metaebene verschoben werden. Autorisierungsdaten wurden Metadaten und die Abbildung der Rollen auf die Sichten führte neue Metadaten ein (gestrichelte Linien rechts). Der Miningprozeß lieferte uns Anhaltspunkte für die Abbildung zwischen Rollen, Personen und Sichten.

Um den Miningprozess überhaupt zu ermöglichen (kombinatorische Komplexität), reduzierten wir die Redundanz in den Daten durch die Bildung von Äquivalenzklassen. Zwei Sichten wurden genau dann als äquivalent angesehen wenn jeweils die gleiche Benutzergruppe Zugriff darauf hatte. Für diesen Zweck entwickelten wir UNIX-Skripte. Die Äquivalenzklassen wurden dann dem eigentlichen Mining unterzogen. Das Resultat waren Muster, die mögliche vertikale Abhängigkeiten wiederspiegeln. Horizontale Abhängigkeiten wurden größtenteils durch Teamwissen über den Problembereich ermittelt.

Die identifizierten Kandidaten wurden anschließend einer weiteren Analyse auf Basis unseres Wissens über die organisatorische Struktur der Bank unterzogen. Wir entwarfen Rollenbeschriebe, die wir im Anschluss daran mit Repräsentanten der Geschäftsseite diskutierten. Im Rahmen dieser Validierungsphase versuchten wir unsere Annahmen bestätigen zu lassen. Das Ergebnis dieser Phase war eine fachlich orientierte Beschreibung der Rollen (Problembereich) zusammen mit einer technischen Spezifikation der Implementierung (Lösungsbereich).

In der letzten Phase sind die Rollen nun Basis für die Implementierung eines Werkzeugs zur Benutzeradministration. Die Rollen dienen zur Realisierung von Metalevelkomponenten. Die identifizierten Abstraktionen werden auch mit dem Metamodell des Gesamt-Warehouse abgeglichen. Das Werkzeug generiert Datenbankprivilegien auf Basis der technischen Spezifikation in den Rollenbeschrieben.

4. Ergebnisse

Tabelle 1 zeigt die während der ersten Phase identifizierten Muster. Zu dieser Zeit zählten wir 637 Benutzer, 627 Sichten und 647 Attribute. Während dieser Phase benutzten wir den Induktionsalgorithmus für Assoziationsregeln, *Apriori* [1]. Die Grenze der statistischen Signifikanz legten wir bei

$$\mathrm{ld}(637) \approx 9$$

fest. Einige Muster mit 8 Treffern nahmen wir aufgrund von Wissen über den Problembereich ebenfalls mit auf.

32 Muster wurden identifiziert, aus denen wir 9 Rollenkandidaten ableiteten. Diese wurden mit Repräsentanten der Geschäftsseite validiert. 354 Benutzer konnten durch die Rollen abgedeckt werden. Aus betrieblichen Gründen adressierten wir zunächst nur Benutzer aus unserer eigenen Geschäftseinheit. Dort konnten wir etwa 90% Abdeckung erzielen. Lediglich 4 der 32 Muster verblieben ungenutzt, was mehr auf Zeitmangel als auf schlechte Verwertbarkeit zurück zu führen ist.

Wir schätzen die Stabilität der identifizierten Abstraktionen (Rollen) als sehr hoch ein. Es kam in keinem der Interviews vor, dass unsere Entwürfe von den Repräsentanten abgelehnt wurden. Es kam nur

Nr	Organisa-tionseinheit	Impliziert Sichtenklasse	Warschein-lichkeit	Sichtenklasse	Impliziert Organisa-tionseinheit	Warschein-lichkeit	Verwendet
1	OE111	SK0001	51:8.4%, 1.0				Ja
2	OE1111	SK0001	9:1.5%, 1.0				Ja
3	OE1112	SK0001	20:3.3%, 1.0				Ja
4	OE1113	SK0001	9:1.5%, 1.0				Ja
5	OE1121	SK0002	34:5.6%, 1.0	SK0002	OE1121	36:5.9%, 0.944	Ja
6	OE112	SK0003	20:3.3%, 1.0				Ja
7	OE1121	SK0003	9:1.5%, 1.0				Ja
8	OE121	SK0004	77:12.7%, 1.0				Ja
9	OE1211	SK0004	8:1.3%, 1.0				Ja
10	OE1212	SK0004	39:6.4%, 1.0				Ja
11	OE1213	SK0004	27:4.4%, 1.0				Ja
12	OE1311	SK0005	10:1.6%, 1.0	SK0005	OE1311	10:1.6%, 1.0	Ja
13	OE141	SK0006	23:3.8%, 1.0				Ja
14	OE1411	SK0006	8:1.3%, 1.0				Ja
15	OE1412	SK0006	11:1.8%, 1.0				Ja
16	OE151	SK0007	10:1.6%, 1.0				Ja
17	OE1511	SK0007	10:1.6%, 1.0	SK0007	OE1511	10:1.6%, 1.0	Ja
18	OE161	SK0008	16:2.6%, 1.0				Ja
19	OE1611	SK0008	8:1.3%, 1.0				Ja
20	OE171	SK0009	15:2.5%, 1.0				Ja
21	OE1711	SK0009	15:2.5%, 1.0	SK0009	OE1711	15:2.5%, 1.0	Ja
22	OE181	SK0010	9:1.5%, 1.0				Ja
23	OE1811	SK0010	9:1.5%, 1.0	SK0010	OE1811	12:2.0%, 0.75	Ja
24	OE21	SK0014	9:1.5%, 1.0				Ja
25	OE31	SK0011	10:1.6%, 1.0				Nein
26	OE311	SK0011	10:1.6%, 1.0				Nein
27	OE3211	SK0012	11:1.8%, 1.0	SK0012	OE3211	11:1.8%, 1.0	Ja
28	OE331	SK0013	41:6.7%, 1.0				Nein
29	OE3311	SK0013	18:3.0%, 1.0				Nein
30				SK0015	OE3411	10:1.6%, 1.0	Ja
31				SK0016	OE3421	8:1.3%, 0.875	Ja
32				SK0017	OE3421	8:1.3%, 0.875	Ja

Tabelle 1: Mining-Kandidaten für Metalevel-Abstraktionen. Die linke Seite der Tabelle zeigt die identifizierten Muster bezüglich Implikation von der Organisationseinheit zu Viewklassen und die assoziierte Wahrscheinlichkeit (Benutzerzahl, prozentualer Anteil relativ zur Benutzerpopulation, Wahrscheinlichkeit). Die rechte Seite zeigt Muster für die umgekehrte Richtung von Viewklassen zu Organisationseinheiten. Die äusserste rechte Spalte gibt an, ob das Muster bei der Bildung einer Metalevelabstraktion genutzt werden konnte. Die Namen der Organisationseinheiten sind hierarchisch organisiert, d.h. OE1111 ist ein Teil der OE111, die wiederum ein Teil von OE11 ist, etc. Die Daten wurden anonymisiert.

zu geringfügigen Änderungen. Mit einer Ausnahme wurden die Interviews nach nur einer Iteration abgeschlossen. Wir denken, dass dies ein guter Indikator für verlässliche Abstraktionen ist. Aus den gleichen

Gründen gehen wir davon aus, dass die Granularität der Abstraktionen gut gewählt ist.

5. Schlussfolgerungen

Wir benutzten Data Mining als Methode zur Identifikation von Metalevelabstraktionen. Es ist ein effizienter Ansatz für Probleme moderater Komplexität. Wir sind überzeugt, dass sich dieser Ansatz auch auf Probleme höherer Komplexität anwenden lässt. Dies werden wir in Zukunft tun.

Unserer Erfahrungen waren äußerst positiv; dies erscheint verständlich, da Data Mining insbesondere zur Bildung von Modellen eingesetzt wird. Zur Zeit implementieren wir die technische Infrastruktur zur Benutzeradministration. Wir denken, dass die identifizierten Abstraktionen stabil und problemangemessen sind, und dass sie die natürliche Ordnung unseres Problembereiches adäquat abbilden. Das ist eine gute Basis für die Erreichung unserer ursprünglichen Architekturziele.

Der von uns gewählte Ansatz basiert auf Daten. Diese Voraussetzung kann nicht immer erfüllt werden, speziell in softwareintensiven Umgebungen. Ein durchaus denkbares Szenario ist allerdings die Verwendung von Daten über Abhängigkeitsverhältnisse in Softwaresystemen im Rahmen einer auf Data Mining basierenden Analyse. Wir denken, dass dieser Ansatz ein gutes Potenzial dafür besitzt.

Danksagung

Ein großes Dankeschön für seinen Anteil an dieser Arbeit geht an meinen Data-Mining-Kollegen Ahmed Rida.

Literatur

1. R. Srikant Agrawal: Fast Algorithms for Mining Association Rules, Proc. of the 20th Int'l Conference on Very Large Databases, Santiago, Chile, September 1994
2. Krzysztof Czarnecki, Ulrich Eisenecker: Generative Programming. Reading 2000 (Addison-Wesley)
3. Object Management Group, Inc.: Common Warehouse Metamodel (CWM) Specification, Februar 2000
4. Hans Wegener: Erste Erfahrungen mit Komponenten, Metadaten und Wiederverwendung im Data Warehousing der Credit Suisse. In Rony G. Flatscher and Klaus Turowski (Hrsg.): 2. Workshop komponentenorientierte betriebliche Anwendungssysteme (WKBA 2), Wien, Österreich, Februar 2000

Definition von Testfällen für kundenspezifische produktfamilienbasierte Anwendungen

Klaus Pohl, Mark Strembeck
Universität GH Essen
Software Systems Engineering
Altendorfer Str. 97-101
45117 Essen
{pohl | strembeck}@informatik.uni-essen.de

Zusammenfassung

Der Fokus einer aktuellen Kooperation des Unternehmens SHS mit der Software Systems Engineering Forschungsgruppe liegt auf der Reduktion des Testaufwands für kundenspezifische Anwendungen, die aus den generischen Artefakten einer Produktfamilie abgeleitet werden. Hierdurch wurde die Erweiterung des V-Modells durch eine szenariobasierte Vorgehensweise initiiert. Diese Vorgehensweise ermöglicht die explizite Berücksichtigung der Definition und Verfeinerung von Tests in jeder Entwicklungsphase. Durch die parallele Aufzeichnung von entsprechenden Traceability-Informationen kann zudem die Abschätzung des Änderungsaufwands sowie die Propagierung von Änderungen in das Testmodell unterstützt werden.

1. Einleitung und Motivation

Bevor mit der Argumentation begonnen wird, soll an dieser Stelle noch eine Definition für die im folgenden verwendeten Begriffe *Software-Produkt* und *Software-System* gegeben werden:

- Ein *Software-Produkt* wird hierbei als eine Konfiguration von Komponenten bzw. Architekturartefakten betrachtet.
- Ein *Software-System* stellt wiederum eine Konfiguration mehrerer Software-Produkte dar.

Zusätzlich muss darauf hingewiesen werden, das jedes der einzelnen Software-Produkte, die zu einem Gesamtsystem integriert werden i.d.R. eine Instanz einer eigenen Produktfamilie ist und prinzipiell auch als "Standalone-Produkt" vertrieben werden kann. Im Fall von SHS werden beispielsweise (u.a.) ein "Radiologie Informationssystem", ein sog. "Healthcare Information System" sowie ein sog. "Picture Archiving and Communication System" zu einem Gesamtsystem integriert. Jedes dieser Produkte kann jedoch auch separat vertrieben werden und mit entsprechenden Systemen anderer Hersteller integriert werden. Ein Software-System im obigen Sinne stellt demnach eine Familie von Produkt-Familien (bzw. von deren Instanzen) dar.

1.1. Problemstellung

In einer aktuellen Kooperation des Unternehmens Siemens Health Services (SHS) mit der Software Systems Engineering Forschungsgruppe der Universität Essen geht es darum, szenariobasierte Testfallableitung im Rahmen der Produktfamilienentwicklung zu untersuchen.

Im Rahmen der angesprochenen Kooperation wurde festgestellt, dass die ohnehin bereits hohen Anforderungen an die Testbarkeit der entwickelten Software sowie an die Möglichkeiten zur Testfallableitung (vgl. z.B. [Harr00], [Hetz88], [Myer79] und [Whit00]) im Produktfamilienkontext nochmals zunehmen. Dies folgt insbesondere aus dem Umstand, dass sowohl die generischen Teile einer Produktfamilie als auch jede der abgeleiteten kundenspezifischen Applikationen umfassend getestet werden müssen. Mit anderen Worten: eine Produktfamilie (bzw. die domänenspezifischen Artefakte) ist für sich bereits ein sehr komplexes System, welches darüberhinaus verwendet wird, um eine prinzipiell beliebige Anzahl kundenspezifischer Applikationen abzuleiten. Für jede abgeleitete Anwendung müssen daher sowohl die generischen als auch die kundenspezifischen Anforderungen erfüllt und dementsprechend auch getestet werden. Das Testen der essentiellen Kundenanforderungen auf korrekte Erfüllung durch die resultierende Software und die Entdeckung evtl. vorhandener Lücken ist kritisch für den Erfolg jedes (Sofware-)Produkts und Systems.

Die Tatsache, daß sowohl die generischen Teile der Produktfamilie als auch die abgeleiteten spezifischen Applikationen im Laufe ihres Lebens Änderungen unterworfen sind, führt zu einer weiteren Erhöhung der Komplexität. Entsprechend muss die Möglichkeit bestehen im Falle der Änderung eines Artefakts schnell die zugehörigen Test-

fälle zu identifizieren und evtl. an die Änderung anzupassen. Hierdurch kann die Entwicklungszeit von Software-Produkten verkürzt werden.

Ebenso wie in der traditionellen Software-Entwicklung verursacht das Testen der Software in der produktfamilienbasierten Entwicklung einen Großteil der gesamten Software-Entwicklungskosten (vgl. [Harr00]). Dies folgt aus der Tatsache, dass der Entwicklungsaufwand bei der Ableitung von Applikationen aus einer Produktfamilie durch einen hohen Grad an Wiederverwendung reduziert werden kann, während die aufwendigen Integrations- und Systemtests jedoch für jede abgeleitete Applikation erneut durchgeführt werden müssen. Da es nicht möglich ist komplexe Software vollständig zu testen und weil darüberhinaus bereits die geringfügige Änderung einer Anforderung oder das Hinzufügen einer neuen kundenspezifischen Anforderung einen großen Testaufwand verursacht, ist es erforderlich aus allen möglichen Tests - nach Signifikanz - die wichtigsten auszuwählen (vgl. z.B. [Hetz88], [Harr00], [Myer79] und [Whit00]). In anderen Worten: es wird eine Möglichkeit zur sinnvollen Reduktion der durchzuführenden Testfälle sowie zur schnellen Anpassung der Testfälle an evtl. Änderungen benötigt.

Aufgrund der oben erwähnten, erhöhten Test-Komplexität im Rahmen der produktfamilienbasierten Entwicklung hätte ein entsprechendes Vorgehen das Potential, die positiven Effekte der Produktfamilien-Entwicklung, wie z.B. erhöhte Qualität der resultierenden Produkte oder verkürzte "Time-to-Market", weiter zu erhöhen.

1.2. Lösungsansatz

Szenarien im allgemeinen und Use Cases im besonderen sind ein bewährtes Mittel, um (funktionale) Kundenanforderungen zu identifizieren und zu modellieren (vgl. z.B. [Jaco92], [JaBC98] und [WPJH98]). Des weiteren ist ein Use-Case-Modell ein hervorragender Ausgangspunkt für die Auswahl der Funktionalität, die eine spezifische Applikation, welche von einer Produktfamilie abgeleitet wird, zur Verfügung stellen soll. Daher ist es sehr wahrscheinlich, dass (funktionale) Änderungen innerhalb einer Produktfamilie bzw. einer kundenspezifischen Anwendung zuerst auf der Use-Case-Ebene identifiziert werden. Aus diesen Gründen wurden Use-Cases von uns als Ausgangspunkt gewählt, um eine an Kundenanforderungen orientierte Reduktion der Testfälle zu ermöglichen.

SHS befindet sich zur Zeit in einer Übergangsphase von der traditionellen hin zu einer produktfamilienbasierten Software-Entwicklung. Der von SHS verwendete Software-Entwicklungsprozess ist an das V-Modell [VMod97] angelehnt. Entsprechend besteht die Aufgabe darin eine geeignete Anpassung des V-Modells vorzunehmen, um die szenariobasierte Testfallableitung im Rahmen einer produktfamilienbasierten Entwicklung zu ermöglichen.

Die Methodenzuordnung für das V-Modell [VMMZ97] verweist bereits auf Methoden für die Ableitung und die Durchführung von Tests. Im Verlauf des Projektes mit SHS wurde aber festgestellt, dass die im V-Modell vorgesehenen Methoden bei weitem nicht ausreichend sind, um die erhöhten Anforderungen im Kontext der produktfamilienbasierten Software-Entwicklung zu erfüllen. Insbesondere existiert keine durchgängige Methodik die eine sinnvolle Reduktion des Testaufwandes und die effiziente Anpassung der Tests bei Änderungen ermöglicht.

Eine an der Universität Essen derzeit in Entwicklung befindliche Erweiterung des V-Modells hat zum Ziel während jeder Entwicklungsphase sog. Testszenarien abzuleiten, die ihrerseits zur Definition von Systemtests und Integrationstests herangezogen werden können. Das besondere Augenmerk liegt auf den frühen Phasen der Software-Entwicklung, wie der Anforderungsdefinition und dem Architekturentwurf.

2. Ein Vorgehen zur szenariobasierten Definition von Testfällen

Im folgenden geben wir einen Überblick über die Erweiterung des V-Modells zur Ermöglichung einer kontinuierlichen, szenariobasierten Definition von Testfällen.

2.1. Das V-Modell

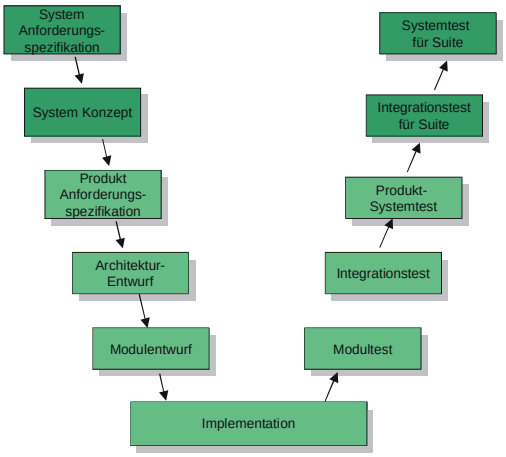


Abbildung 1: Das V-Modell

Das V-Modell (vgl. Abbildung 1) ist ein bekannter Standard für die Entwicklung von IT-Systemen und seit Juni 1996 in Deutschland vorgeschrieben für die Erstellung von IT-Systemen des Bundes (vgl. [VMod97]). Das V-Modell umfasst ein Prozessmodell, eine Methodenzuord-

nung sowie funktionale Werkzeuganforderungen. Es legt fest, welche Schritte im Einzelnen durchzuführen sind, welche Methoden innerhalb der verschiedenen Entwicklungsaktivitäten zur Anwendung gelangen und welche funktionalen Merkmale die verwendeten Werkzeuge besitzen müssen. Ein besonderes Charakteristikum des V-Modells ist die Möglichkeit zur Anpassung an verschiedenste Projekte und Organisationen ("Tailoring").

Bei SHS werden Software-Produkte und Software-Systeme, wie in Abschnitt 1 definiert, unterschieden. Nachfolgend soll nun jeweils kurz auf diejenigen Phasen eingegangen werden, die sich besonders auf die Erstellung von Software-Systemen beziehen (vgl. Abbildung 1):

- Die Phase *System-Anforderungsspezifikation* dient der Beschreibung der Anforderungen an ein aus mehreren Software-Produkten zusammengesetztes Software-System.
- Die Phase *Systemtest für Suite* korrespondiert mit der Phase System-Anforderungsspezifikation und ist für die Durchführung der Testszenarien vorgesehen, die das Gesamtsystem ("die Suite") auf seine korrekte Funktion überprüfen.
- In der Phase *System-Konzept* werden die Beziehungen zwischen den verschiedenen Software-Produkten beschrieben (z.B. Interaktionen und die physikalische Verteilung). In anderen Worten: es wird eine abstrakte System-Architektur beschrieben, in der die einzelnen Software-Produkte als Architekturbausteine fungieren.
- Die Phase *Integrationstest für Suite* korrespondiert mit der Phase System-Konzept und beinhaltet die Durchführung der Testszenarien, die die Integration von zwei oder mehreren Software-Produkten adressieren.

2.2. Szenariobasierte Ableitung von Tests

Mit Hilfe des 1,5 V-Modells (vgl. Abbildung 2) können in jeder Entwicklungsphase Testszenarien abgeleitet werden, die in der jeweils korrespondierenden Testphase ausgeführt werden müssen.

Die Bezeichnung "1,5 V-Modell" rührt von den zusätzlichen, in Abbildung 2 erkennbaren Aktivitäten her, die parallel zu den Entwurfs- und Erstellungsphasen verlaufen und speziell auf die kontinuierliche Ableitung von Testszenarien und die Aufzeichnung von sog. Traceability Informationen (vgl. z.B. [GoFi94]), zur effektiven Unterstützung von Change Management Prozessen, ausgerichtet sind. Im Rahmen der produktfamilienbasierten Entwicklung liegt hierbei ein besonderer Schwerpunkt auf der Erstellung generischer Test- und Traceability-Artefakte, die bei der Erstellung jeder Produktfamilien-Instanz wiederverwendet werden können.

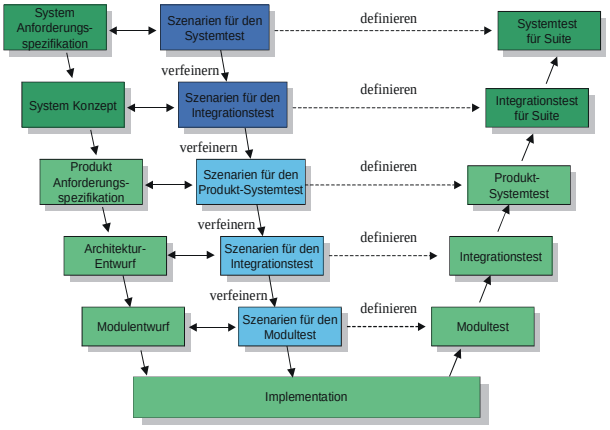


Abbildung 2: Das 1,5 V-Modell

Wie bereits kurz erwähnt zielt das in Abbildung 2 dargestellte 1,5 V-Modell u.a. darauf ab in jeder Phase der Software-Entwicklung Testszenarien abzuleiten, die in der korrespondierenden Testphase ausgeführt werden. Auf diese Weise kann sichergestellt werden, das die entsprechenden Tests in zeitlicher Nähe zu der Spezifikation entworfen werden, deren Erfüllung sie im Anschluss überprüfen sollen. Hierdurch kann die Gefahr verringert werden, dass wichtige Testfälle evtl. vergessen oder nur unzureichend spezifiziert werden. Weiterhin wird durch dieses Vorgehen die kontinuierliche Aufzeichnung der Beziehungen (Links) zwischen Entwurfsartefakten und den zugehörigen Testfällen gewährleistet. Diese Beziehungen ermöglichen im Fall einer Änderung die unmittelbare Bestimmung der evtl. betroffenen Testszenarien und bilden somit die Grundlage für ein effizientes Change Management mit Bezug auf Testfälle.

Dieses Vorgehen bietet somit auch eine direkte Unterstützung der sog. "Feedback-Schleife" zwischen Tests und Änderungen, wobei davon ausgegangen wird das Tests zu neuen Änderungswünschen und durchgeführte Änderungen zu einer erneuten Testinstanziierung führen.

Zwischen den Entwurfsartefakten wie Anforderungen oder Architekturkomponenten und den entsprechenden Testszenarien besteht eine n:m Beziehung. Das bedeutet, dass die korrekte Erfüllung einer Anforderung durch ein oder mehrere Testszenarien überprüft wird, während das gleiche Szenario u.U. eingesetzt werden kann, um die richtige Umsetzung mehrerer Anforderungen bzw. von Teilen verschiedener Anforderungen nachzuweisen. Die in Abbildung 2 sichtbaren (vertikalen) Pfeile zwischen den verschiedenen Ebenen der Testszenarien deuten entsprechend an, dass Szenarien einer höheren Ebene durch

weitere Testszenarien auf einer tieferen Ebene verfeinert werden können. Eine Verfeinerung bedeutet immer eine Detaillierung der zugehörigen Testszenarien.

Entsprechend existieren Teile eines Testszenarios, die komplett auf einer Testebene ausgeführt werden, während andere Teile evtl. über mehrere Ebenen verteilt getestet werden müssen. Mit anderen Worten: Ein Szenario auf der Ebene des Produkt-Systemtests wird mit hoher Wahrscheinlichkeit durch ein oder mehrere Test-Szenarien auf der Ebene des Integrationstests verfeinert. Durch diese Verfeinerungsbeziehungen wird ein "Leveling" der Testszenarien erreicht. Dies erleichtert wiederum die Abschätzung des Testaufwandes sowie die Propagierung von Änderungen in das Testmodell.

Des weiteren besteht die einfache Möglichkeit z.B. nur die Testfälle auf der Ebene des Produkt-Systemtests auszuführen ("Test in die Breite") oder einen bestimmten Produkt-Systemtest auszuwählen und inklusive aller assoziierten Detailszenarien zu testen ("Test in die Tiefe").

2.3. Fokus der Kooperation

Fokus der Kooperation mit SHS ist die Ableitung von Testfällen in den frühen Produkt-Entwicklungsphasen, d.h. während der "Produkt-Anforderungsspezifikation" bzw. des "Use-Case-Entwurfs" und des "Architekturenwurfs". Hierzu werden zunächst die in Form von Szenarien modellierten Anforderungen analysiert, um anschließend geeignete Testfälle daraus ableiten zu können. Durch die fortlaufende Verfeinerung der Szenarien soll erreicht werden, dass die korrekte Erfüllung der Kundenanforderungen durch das resultierende Software-Produkt effizient getestet werden kann.

Da nicht nur die Anpassung eines bereits vorhandenen Artefakts sondern z.B. auch das Hinzufügen eines komplett neuen Artefakts als Änderung betrachtet wird (vgl. z.B. [Wein97]), kann die Ableitung einer speziellen Anwendung aus den generischen Artefakten einer Software-Produktfamilie ebenfalls als Änderungsprozess betrachtet werden. Aus diesem Grund muss zudem berücksichtigt werden wie sich Änderungen auf Use-Case- und Architekturebene auf die zugehörigen Testfälle auswirken. Um diese Auswirkungen korrekt abschätzen und in die betroffenen Artefakte integrieren zu können, ist es besonders wichtig, dass geeignete sog. Tracebility-Links zur Verfügung stehen. Um diese Links sowie die damit zusammenhängenden Daten in ähnlichen Änderungsfällen wiederverwenden zu können, müssen entsprechende Link-Typen identifiziert und geeignet beschrieben werden. Die durch diese Links definierten Beziehungen sowie die zugehörigen Attributwerte werden dauerhaft gespeichert, um Nachvollziehbarkeit zu gewährleisten und somit das Change-Management zu unterstützen. Nachfolgend werden einige einfache Typen von Beziehungen beispielhaft in kurzer Form charakterisiert:

- *erfüllt* zwischen Teilarchitektur und Anforderung
- *realisiert* zwischen Funktions-Spezifikation und Anforderung
- *abgedeckt* zwischen Test und Anforderung
- *verfeinert* zwischen Test und Test

Um die Aufzeichnung sinnvoller Trace-Informationen zu ermöglichen, werden geeignete Strukturen benötigt zwischen denen Beziehungen (Links) hergestellt werden können. Auf diese Weise wird die schnelle Identifikation der evtl. von einer Änderung betroffenen Artefakte (bzw. der betroffenen Strukturen innerhalb dieser Artefakte) ermöglicht. Die Änderung einer Vorbedingung in der Beschreibung eines bestimmten Use-Case könnte z.B. Auswirkungen auf die Schnittstellen der zugehörigen Architekturkomponenten haben. Ein entsprechender Link zwischen Vorbedingungen und Schnittstellen ermöglicht somit die schnelle Propagierung einer solchen Änderung und erleichtert die Abschätzung des zu erwartenden Änderungsaufwandes. Links zu den entsprechenden Testszenarien gewährleisten zudem die umgehende Abschätzung des durch eine bestimmte Änderung verursachten Testaufwandes und ermöglichen das gezielte Testen der geänderten Strukturen. Außerdem wird so die Propagierung von Änderungen in die Testszenarien ermöglicht, was wiederum zu einer Beschleunigung der evtl. nötigen Anpassung von Testszenarien an geänderte Entwurfsartefakte führt. Für die Identifikation geeigneter Link-Typen wurde das nachfolgend verkürzt dargestellte Vorgehen gewählt:

- Definition jeweils eines Metamodells für die Beschreibung von Use-Cases, Architekturartefakten und Testszenarien.
- Identifikation der Beziehungen zwischen den verschiedenen Bestandteilen zwischen Use-Case- und Architekturmetamodell auf der einen und dem Test-Metamodell auf der anderen Seite.

Auf diese Weise konnten Link-Typen zur Beschreibung von Beziehungen zwischen funktionalen Anforderungen (in Form von Use-Cases) und Testszenarien sowie Architekturartefakten und Testszenarien abgeleitet werden.

2.4. Vorteile

Ein szenariobasiertes Vorgehen für die Ableitung von Testfällen weißt folgende Vorteile auf:

- Sinnvolle Reduktion der Testfälle durch Orientierung an Kundenanforderungen.
- Explizite Berücksichtigung von Tests in jeder Entwicklungsphase.
- Fortlaufende Verfeinerung der Testszenarien und somit verbesserte Möglichkeit zur Erreichung einer guten Testabdeckung (in Bezug auf Kundenanforderungen).
- Unterstützung von Change Management Aktivitäten für Testfälle durch die parallel aufgezeichneten Traceability-Informationen.

3. Zusammenfassung und Ausblick

Im Rahmen der Kooperation unserer Forschungsgruppe mit dem Unternehmen SHS wurde die Entwicklung des 1,5 V-Modells als Erweiterung des V-Modell Standards [VMod97] initiiert. Ziel der Kooperation ist insbesondere die sinnvolle Reduktion des Testaufwands. Dies gilt sowohl für die Durchführung als auch für die Anpassung von Tests an Änderungen.

Der von uns gewählte Use-Case- bzw. szenariobasierte Ansatz ermöglicht die explizite Orientierung an Kundenwünschen und führt zu einer kundenspezifischen Selektion von Testfällen. Der besondere Fokus des 1,5 V-Modells liegt auf der expliziten und kontinuierlichen Berücksichtigung von Tests in jeder Phase der Entwicklung sowie der Aufzeichnung von entsprechenden Traceability-Informationen. Im Kontext der Produktfamilien-Entwicklung soll hierbei insbesondere auf die Erstellung generischer Traceability- und Test-Artefakte geachtet werden, die bei der Ableitung einer neuen Instanz der Produktfamilie wiederverwendet werden können. Der Ansatz zur Adaption des V-Modells wurde gewählt, da sich SHS aktuell in einer Übergangsphase von der traditionellen zur produktfamilienbasierten Software-Entwicklung befindet und das vielfach bewährte V-Modell auch weiterhin verwenden möchte.

Das initiale 1,5 V-Modell befindet sich zur Zeit in der Erprobung und wird im Detail kontinuierlich angepasst bzw. fortentwickelt. Die bisherigen Erfahrungen zeigen, dass das 1,5 V-Modell ein sinnvolles Vorgehen zur Reduktion des Testaufwandes im Rahmen der produktfamilienbasierten Software-Entwicklung liefert.

Zukünftige Arbeiten werden sich vor allem auf die weitere Betonung der Wiederverwendbarkeitsaspekte der verschiedenen Traceability- und Test-Artefakte sowie auf die Entwicklung einer szenariobasierten Methodik zur zielgerichteten Anpassung und Durchführung von Tests bei der Entwicklung von produktfamilienbasierten, kundenspezifischen Anwendungen konzentrieren.

4. Danksagung

Wir möchten uns für die intensive und kooperative Zusammenarbeit mit SHS und insbesondere bei Norbert Herzog, Josef Weingärtner und Christian Zapf bedanken. Diese Arbeit wurde im Rahmen des ESAPS Projekts teilweise durch das BMBF finanziert (Eureka ∑! 2023 Programme, ITEA project 99005).

5. Literatur

[GoFi94] Gotel, O.C.Z.; Finkelstein, A.C.W.: “An Analysis of the Requirements Traceability Problem”, In: Proc. of the 1st Int. Conference on Requirements Engineering, IEEE CS Press, 1994

[Harr00] Harrold, M.J.: "Testing: A Roadmap", In: A. Finkelstein (editor), The Future of Software Engineering, ACM Press, 2000

[Hetz88] Hetzel, B.: "The complete Guide to Software Testing - Second Edition", John Wiley & Sons, 1988

[JaBC98] Jarke, M.; Bui, X.T.; Carroll, J.M.: “Scenario Management: An Interdisciplinary Approach“, In: Requirements Engineering Journal, Springer-Verlag, 1998

[Jaco92] Jacobson, I.: “Object-Oriented Software Engineering : A Use Case Driven Approach”, Addison-Wesley, 1992

[Myer79] Myers, G.J.: "The Art of Software Testing", John Wiley & Sons, 1979

[VMod97] Das V-Modell - Entwicklungsstandard für IT-Systeme des Bundes, 1997, http://www.v-modell.iabg.de/, Zugriff Sept. 2000

[VMMZ97] V-Modell - Methodenzuordnung, allgemeiner Umdruck Nr. 251, 1997, http://www.v-modell.iabg.de/vm97.htm

[Wein97] Weinberg, G.M.: " Quality Software Management Volume 4: Anticipating Change", Dorset House Publishing, 1997

[Whit00] Whittaker, J.A.: "What is Software Testing ? And Why It Is So Hard ?", IEEE Software January/February 2000

[WPJH98] Weidenhaupt, K; Pohl, K.; Jarke, M.; Haumer, P.: “Scenario Usage in System Development: A Report on Current Practice“, In: IEEE Software, March 1998

Document Information

Title:	Proceedings of 1. Deutscher Software-Produktlinien Workshop (DSPL-1), Kaiserslautern, November 2000
Date:	November 10, 2000
Report:	IESE-076.00/E
Status:	Final
Distribution:	Public