Proceedings for First Workshop on Aspect-oriented Product Line Engineering (AOPLE-1)

Lancaster University Computing Department Technical Report

COMP-004-2007

Held at GPCE 2006
Portland, Oregon, USA

Sunday October 22nd 2006

AOPLE Web site at
http://www.softeng.ox.ac.uk/aople/index.html

Preface

Workshop Papers

Preface

Aims and Goals of the Workshop

Product Line Engineering (PLE) is an increasingly important paradigm in software development whereby commonalities and variations among similar systems are systematically identified and exploited. PLE covers a large spectrum of activities, from domain analysis to product validation and testing. Variability is manifested throughout this spectrum in artefacts such as models, requirements, code and components and it is often of crosscutting nature. These characteristics make Aspect-Oriented Software Development (AOSD) techniques appealing as suitable candidates to modularize variability. Work on Generative Programming (GP) and Component Engineering (CE) has shown the crucial role they play in PLE and the potential benefits of its synergy with AOSD.

The workshop aimed at expanding and capitalizing on the increasing interest of researchers from these communities. The main goal of the workshop is therefore to share and discuss ideas, identify research opportunities and foster collaboration to tackle the challenges these opportunities may bring about.

Attendees

Neil Loughran - loughran@comp.lancs.ac.uk

Roberto Lopez-Herrejon - Roberto.Lopez@comlab.ox.ac.uk

Daniel Lohmann - dl@cs.fau.de

Karen Cortes Verdin - karen@cimat.mx

Janet Liu - janetlj@iastate.edu

Sven Apel - apel@iti.cs.uni-magdeburg.de

Florian Heidenreich - florian.heidenreich@inf.tu-dresden.de

Sergio Soares - sergio@dsc.upe.br

Paulo Borba - phmb@cin.ufpe.br

Leslie Seymour - lseymour@MagellanGPS.COM

Michael Haupt - michael.haupt@hpi.uni-potsdam.de

Mario Sudholt – sudholt@emn.fr

Workshop Program

The morning session was set aside for presentations of the accepted papers as follows:

9:00 Coffee and Welcome

9:20 Paper 1: Concern Hierarchies

9:40 Paper 2: Assessment of Product Line Architecture and Aspect-Oriented Software Architecture Methods

10:00 Paper 3: Using Graph-Rewriting for Model Weaving in the context of Aspect-Oriented Product Line Engineering

10:20 Paper 4: From Conditional Compilation to Aspects: A Case Study in Software Product Lines Migration

10:40 *Coffee Break*

11:00 Paper 5: On the Structure of Crosscutting Concerns: Using Aspects or Collaborations?

11:20 Paper 6: Towards Crosscutting Metrics for Aspect-Based Features

11:40 Paper 7: The Role of Aspects in Modeling Product Line Variabilities

12:00 Wrap up morning session

12.15 Lunch (until 2.00pm)

The afternoon session was set aside for discussion and debate regarding the role of AOSD in software product line engineering. A separate workshop report was created which discusses the many topics addressed. It can be downloaded at:

http://www.softeng.ox.ac.uk/aople/aople1/report.pdf

Assessment of Product Line Architecture and Aspect-Oriented Software Architecture Methods

Karen Cortes Verdin, Cuauhtemoc Lemus Olalde
Computer Science Department
Center for Research in Mathematics (CIMAT, A.C.)
Calle Jalisco S/N
Col. Mineral de Valenciana Guanajuato, Gto. 36240, Mexico
karen@cimat.mx, clemola@cimat.mx

Abstract

The Product Line Architecture is the most important asset of a Product Line. The Product Line Architecture defines not only the product line quality attributes but also encompasses the capability of reuse, product derivation, and product line evolution. Aspect-Oriented approaches seek proper separation of concerns in order to obtain evolvable, maintainable, comprehensible, customizable and reusable software. Current Product Line Architecture and Aspect-Oriented Software Architecture methods are assessed considering those characteristics that a product line architecture must fulfill as well as the aspect-oriented software architecture's characteristics. The results from this effort will provide the basis for the development of an Aspect-Oriented Product Line Architecture approach. The objective is the proper identification, separation, and modeling of a PLA's crosscutting concerns.

1. Introduction

A Software Product Line (SPL) is a "set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [1]. Software Product Line Engineering attempts to capitalize reuse by setting a framework for planning, development and management of core assets. The Product Line Architecture (PLA) is the most valuable core asset since it enables cost-effective development of Product Line (PL) products, models commonality (a key feature for PLA and therefore for a successful PL) and supports SPL evolution. A PLA encompasses interesting challenges in addition to those of a software architecture. When dealing with a PLA, in addition to the quality attributes of the products, the PLA has to exhibit its own quality attributes. Furthermore, the PLA should support commonality and variability, and it also has to be generic enough to support the realization of the planned products within the PL.

Aspect-Oriented (AO) approaches seek to apply the separation of concerns principle in order to obtain software which is simpler to evolve, maintain, comprehend, customize, and reuse. Such separation of concerns can be applied from the early phases of software development. Concerns in these phases crosscut requirements' and/or architecture's artifacts. These concerns are known as Early Aspects (EA) [2].

Current PLA and Aspect Oriented Software Architecture (AOSA) methods are assessed herein. The objective is to appraise if current PLA and AOSA methods consider PL-specific quality attributes, the characteristics that a PLA must fulfill (generality, and support for commonality and variability), as well as if they follow an AO or EA approach. What is sought by incorporating an AO approach into the design of a PLA, is the proper identification of crosscutting concerns, and their inclusion into a PLA. If these concerns are not properly considered, they can lead to tangled code during implementation, with the obvious consequences of loss of maintainability, integrability, manageability, and evolvability for the PL. This assessment is part of the work done during the first. year of doctoral research. The corresponding outcomes will justify the development of an Aspect-Oriented Product Line Architecting (AOPLA) methodology.

The assessment considers as criteria, in the first place, the PL-specific quality attributes as specified in the CAFÉ Quality Model [3]. The assessment addresses whether the architecting method considers the identification and design of such quality attributes. The PL-specific quality attributes are [3]:

1. Variability. An asset contains common and varying parts covering in this way aspects of different product line members.
2. Derivability. A generic core asset has attributes that provide for product-specific instance derivation..
3. Reusability. The asset's capability of being reused in different product line members.
4. Rateability. Capability to estimate a core asset's worth.
5. Integrability. The extent to which a a system-specific asset can be into the PL infrastructure.
6. Correctness. Extent to which an asset realizes satisfies its specification and meets the PL's mission objectives.
7. Evolvability. The degree to which an asset can deal with growing complexity and demand as well as continuous change.
8. Manageability. A core asset is manageable if one can plan, decide and observe with respect to the different states the asset can get into.
9. Maintainability. Capability of an asset to be modified

Generality is considered too, as well as the support for commonality and variability.

From the AO point of view, the assessment contemplates whether the method specifically considers: 1) a process for concern development, 2) aspect modularization, 3) whether the aspects are depicted in the architecture, and 4) if the method provides some mechanism for specifying concern composition.

2. Assessment

The PLA methods selected for this assessment are the most well know and applied methods for Product Line Architecture development. The selection of AOSA methods is mainly based on the survey presented in [14]. This survey already makes a selection of the most significant AOSA approaches.

PLA methods are presented in section 2.1 followed by AO methods in section 2.2. Table 2.1, in section 2.3 summarizes the actual assessment. It should be noted that, in the case of PLA methods, as they have not been developed to consider crosscutting concerns in a specific way, they do not fulfill the AO criteria already mentioned. The AOSA methods, in turn, do not address PL-specific quality attributes or generality. They also do not support commonality and variability.

2.1. PLA methods

2.1.1. ADD. Attribute Driven Design Method (ADD) [4] was developed within the SEI's Product Line initiative along with Robert Bosch GmbH. ADD is a recursive decomposition method. The method makes an explicit consideration of quality attributes since it asks, from the beginning, for an explicit statement of quality attribute requirements. ADD guides the architect through a series of design decisions that help to meet those requirements. Architectural drivers constitute the guide for making such design decisions. Architectural drivers are "the combination of quality, functional, and business requirements that shape the architecture" [4]. Commonality is supported by identification of commonalities across architecture's component instances. However, in order for the PL-specific quality attributes and generality to be explicitly considered, care should be taken that they are defined within the architectural drivers.

2.1.2. PuLSE-DSSA. PuLSE-DSSA [5] is a process that integrates software architecture creation and evaluation. It develops a reference architecture by applying generic scenarios in an iterative process. The application of generic scenarios proceeds from the most to the least architecturally significant. Evaluation is integrated in each iteration of the PLA creation process. Inputs to PuLSE-DSSA are the scope definition from PuLSE-ECO and the domain model. The output is a PLA which consists of a PLA description, an Architecture Decision Model (ADM), and optionally, a prototype. In order for PL-specific quality attributes and generality to be explicitly considered they must be described as critical use cases in scenarios. PuLSE-DSSA supports commonality and variability.

2.1.3. FAST. Family-Oriented Abstraction, Specification, and Translation (FAST) is actually a pattern for software production processes that "strives to resolve the tension between rapid production and careful engineering" [6]. Any process that conforms to such a pattern is called a FAST process. FAST relies on the concept of concurrent engineering: both product and process are designed together. The FAST process specifically encompasses commonality analysis. Since FAST is a pattern it can be customized according to the business' needs. Therefore, PL-specific quality attributes, generality and AO characteristics can be incorporated into FAST,

2.1.4. FORM. Feature-Oriented Reuse Method (FORM) [7] is an extension of Feature-Oriented Domain Analysis method (FODA) [8] supporting the construction of a reference architecture. FORM is based on a commonality analysis expressed in a domain model in terms of features. Features are used due to the fact that they are abstractions that both customers and developers understand. From the domain model, product architectures and components can be derived. FORM considers different types of features:

- Services or functions provided by the system, about which users are more concerned.
- Domain technologies, which are the concern of system analysts and designers.
- Implementation techniques, which are of developer's interest.

Only during FORM's domain engineering phase, by the identification and modeling of features, can PL-specific quality attributes and generality be considered. Commonality analysis, on the other hand, is explicitly addressed during domain or feature modeling in the domain engineering phase.

2.1.5. QADA. Quality-driven Architecture Design and quality Analysis (QADA) [9] provides a systematic way to transform functional and quality requirements into software architecture. QADA is a quality driven method. It uses architectural styles and patterns for the achievement of quality requirements. QADA is also a scenario-based method. It guides the development of PLA documentation and specifically addresses variation modeling.

One of the activities in QADA is requirements engineering, which, in the case of this method, consists of an interface between the requirements engineering phase and architectural design. Only if the PL-specific quality attributes have been specified during the requirements engineering phase will they be considered during QADA.

With respect to generality, the method is not clear on how to consider it during the architecting process. Nevertheless, it does assess the architecture's generality when performing the scenario-based analysis of the concrete architecture design (the PLA design is divided into a conceptual architecture design and a concrete architecture design).

2.1.6. QASAR. (Quality Attribute oriented Software ARchitecture) This PLA approach explicitly includes an assessment of quality attributes and the corresponding design decisions to achieve such quality attributes. The architecture consists of four artifacts [10]: the system context, the archetypes, the architecural structure and the design decisions. Commonality is represented by means of architectural archetypes. The first step is to define a requirement specification for the PLA that combines the functional requirements for each feature in one set of functional requirements. A similar activity is done for the quality requirements. Next, a functionality based architecture is developed. Then, a qualtiy attributes assessment is performed in this first resulting architecture. The assessment considers three main techniques [10]: scenarios, simulation, or mathematical modeling. According to the results of the assessment, the architecture is transformed. If any transformations are identified, these may be: "imposing an architectural style, imposing and architectural pattern, applying a design pattern and converting quality requirements to functionality" [10].

In order for QASAR to address the PL-specific quality attributes and generality, they must be specified during scoping and requirements engineering. During the assessment step the derivation of specific products is explicitly considered.

2.1.7. KobrA. Komponentbasierte Anwendungsentwicklung (KobrA, german for component-based application development), claims to be "the first comprehensive methodology to support model driven architectures" [11]. It can be used in combination with other PL methods such as FODA and FAST.

KobrA applies the separation of concerns principles along the process by considering three orthogonal dimensions of development: level of abstraction, level of genericity, and decomposition. During the framework engineering phase (a framework is a reusable set of artifacts whose core is embedded within all products), the PLA is developed. During this phase, by applying the genericity principle, commonalities and variabilities within the PL are captured.

KobrA makes use of KobrA components or Komponents. Komponents are not physical components in the sense of actual component technologies. They rather correspond to logical building blocks of a software system. In this sense the term Komponent refers to any kind of component including components. KobrA supports the principle of encapsulation by modeling a Komponent in terms of a “specification” and a “realization.” A Komponent specification defines what the Komponent does, while a realization describes how the Komponent does it. At the framework level it is possible to have specifications without accompanying realizations. The non-functional or quality requirements specification is an auxiliary artifact of a Komponent specification. Such quality requirements must be oriented towards the specific Komponents and should be contained in the corresponding Komponent requirements document. Therefore, if PL-specific quality and generality attributes are to be considered, they should be specified in the corresponding Komponent’s requirements document.

2.2. AO methods

2.2.1. PCS (Perspectival Concern-Space). PCS “represents a technique of depicting concerns of multiple kinds (or dimensions) in an architectural view consisting of one or more models and diagrams” [12]. A perspective is a “way of looking at a multidimensional space of software concerns from a specific viewpoint” [7]. Every perspective has an orientation, and the orientation of a perspective is determined by a set of related concerns, a purpose, a context and a viewpoint. PCS addresses multidimensional separation of concerns [13] in combination with UML and IEEE Std-1471.

PCS concern development process is concern reification. This process reifies stakeholders’ concerns into viewpoint language elements. Reification is accomplished via projection. A projection is “an architectural abstraction that defines the relationship between a viewpoint and a view – or between a view and a set of models” [12]. A projection defines how to reify one or more concerns into descriptive units. Descriptive units compose the software architecture. Descriptive units can be simple and composite, and they correspond, in the case of simple units, to a link, an attribute, a parameter. In the case of composite units, they can be classes, subsystems, packages and any kind of UML diagrams.

2.2.2. DAOP-ADL. DAOP-ADL is an XML-based architecture description language used to describe the architecture of an application in terms of a set of components, a set of aspects and the interconnections among them.

DAOP-ADL does not consider a concern development process. Since DAOP-ADL is an ADL it provides the constructs for representing aspects and components in the architecture. It does not provide any guideline for aspect modularization. DAOP-ADL handles two different kinds of composition constraints [14]:

1. The *componentCompositionRules* describe the rules that drive the composition of components
2. The aspectEvaluationRules, which are equivalent to aspect pointcuts in AOP languages. These rules describe the weaving rules between components and aspects.

2.2.3. AOGA. Aspect-Oriented Generative Approaches (AOGA) [14] is an architecture-centric approach that was originally focused on multi-agent systems. AOGA encompasses domain-specific languages, modeling notations and code generation tools. It is divided in three stages: domain analysis and specification, architecture design, and implementation. Aspects can be captured and specified in previous development stages. During domain analysis and specification is a stepwise process that allows modeling, specifying, and modularizing crosscutting as well as non crosscutting concerns. Since this is an AO architecting method aspects are represented within the software architecture. Concern composition specification is accomplished via its own domain-specific-language named Agent-DSL.

2.2.4. TranSAT. TranSAT is a framework that focuses on facilitating software evolution through the realization of Aspect Oriented Software Development (AOSD) principles [15]. It consists of an incremental process during which the architecture is defined by weaving a new architecture plan within the architecture. In TranSAT, concerns are merged until the system is complete. A new concern is integrated into the architecture by the use of a software architecture pattern. A pattern contains all the necessary information to enable the inclusion of a new concern. It organizes the information in three parts: an architecture plan, a join point mask, and a set of transformation rules. The actual binding or integration is performed by a weaver, which uses the join point mask and the transformation rules.

2.2.5. PRISMA. PRISMA is an extension to OASIS (Open and Active Specification for Information Systems) that encompasses Component-Based Software Development and Aspect-Oriented Software development. PRISMA provides a component definition language to define architectural types at a high abstraction level [16]. It also provides a configuration language which "designs the software architecture of systems by creating and interconnecting instances of the defined types including all the imported COTS" [16]. OASIS is a formal language for defining conceptual models of object-oriented systems. PRISMA extends OASIS to allow the specification of software architectures. PRISMA includes a higher level of abstraction, that is conceptual design. A PRISMA type can include several aspects: functional, coordination, distribution, quality, context-awareness, and evolution. PRISMA is a model to define architectures. The architectural model consists of different types: interface, aspect, component, connector, and system. PRISMA types are defined by composition of aspects

PRISMA employs reflection as the means for the modification of its metamodel. In this way PRISMA can be tailored to fulfill specific needs.

2.3. Assessment

The actual assessment of PLA and AOSA methods is presented in Table 2.1 on next page. These results she that neither method fulfills all the assessment criteria. PLA methods were designed to consider conventional architectural concerns, not to follow an AO approach. Nevertheless, some of the methods are suitable to consider crosscutting concerns. AOSA methods, on the other hand although not designed to consider the development of a PLA are suitable for considering PLA characteristics, specially the PL-Quality attributes and generality.

3. Summary

PLA and AOSA methods have been assessed. The objective of this effort was to determine the status of current PLA and AOSA methods in addressing PL and AO characteristics.

The assessment considered the main characteristics that a PLA architecture must fulfill: PL-specific quality attributes, generality and support for commonality and variability. In relation to AO, the appraisal considered whether the methods encompassed; 1) a process for concern development, 2) aspect modularization, 3) description of aspects within the architecture, and 4) if the method provides some mechanism for specifying concern composition.

Neither architecting methods satisfies all the assessment criteria. These results justify therefore, the development of an Aspect-Oriented Product Line Architecting (AOPLA) methodology. This methodology should explicitly encompass the PL-specific quality attributes, generality and commonality and variability support. Furthermore, by incorporating an AO approach it is with the intent to properly identify crosscutting concerns, and their inclusion into a PLA.

Table 2.1. Assessment of PLA and AOSA methods.

Approach/ Characteristic	**PL-specific quality attributes**	**Support for commonality and variability**	**Generality**	**Concern development process**	**Aspect modularization**	**Aspects representation in architecture**	**Concern composition specification**
ADD	PL-specific quality attributes must be in the architectural drivers in order to be considered during the process	At each iteration, commonalities across component instances must be identified	It should be considered in the architectural drivers	No	No	No	No
PuLSE-DSSA	PL-specific quality attributes must be specified as critical use cases in the scenarios	It is explicitly considered during the process	It should be considered in a scenario	No	No	No	No
FAST	FAST process can be customized	Commonality and variability is explicitly considered during the process	It is explicitly performed during Domain Modeling	FAST process can be customized	FAST process can be customized	FAST process can be customized	FAST process can be customized
QADA	The quality attributes must be specified during the requirements engineering phase	Supports variation modeling	Assessed during scenario-based analysis of the concrete architecture	No	No	No	No
KobrA	Must be specified in the requirements specification document of the corresponding Komponents	Commonalities and variabilities are captured during framework engineering by applying the genericity principle	By separating concerns, the method explicitly defines a dimension for genericity	No	No	No	No
PCS	No	No	No	By means of projection, PCS reifies stakeholders' concerns into viewpoint language	By means of projection, concerns are transformed into descriptive units within the architecture	Basic descriptive units correspond to UML elements. Composite descriptive units correspond to	By means of interaction concerns. This is done using a perspectival concern-space

				elements		classes, subsystems, packages and any kind of UML diagram	named Aspect-oriented Construction PCS
DAOP-ADL	No	No	No	No	No	As an ADL it describes aspects within the architecture	It specifies *aspectEvalautionRules*, which constraint composition among components and aspects
AOGA	No	No	No	Aspects can be captured and specified in preliminary development stages. In domain analysis and specification, the specification of crosscutting features is done as domain aspects	The architect defines architectural aspects in the AO architecture during architecture design in a stepwise fashion	During AO architecture design	By means of the Agent-DSL
TranSAT	No	No	No	By means of an architecture pattern, TranSAT allows the inclusion of new concerns into the architecture	The new architecture plan within the architecture pattern identifies a self-sufficient component assembly, which in turn, implements a given concern	By means of the new architecture plan, the component assemblies implementing a concern are identified, this becomes part of the architecture	The architecture pattern's join point mask and set of transformation rules specify the architecture's concern integration
PRISMA	Can be considered within the aspects in the PRISMA types	No	An aspect categorization is pre-defined in PRISMA, however additional aspects can be included	By using the types of PRISMA architectural model	By means of the PRISMA architectural model	By means of PRISMA types defined in the architectural model	PRISMA types are obtained by composition of aspects; functional, quality, distribution, coordination, context-awareness, and evolution. The composition is pre-defined by the aspects

5. References

[1] P. Clements, and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, USA, 2001.

[2] Early Aspects Web Site, http://www.early-aspects.net/

[3] M. Anastasopoulos, J. Bayer , Product Family Specific Quality Attributes (IESE Report No. 042.02/E, Version 1.0). Kaiserslautern, Germany: Fraunhofer Institut Experimentelles Software Engineering, 2002.

[4] ADD Method Web Site, http://www.sei.cmu.edu/productlines/add_method.html

[5] M. Anastasopoulos, J. Bayer , O. Flege, C. Gacek, "A Process For Product Line Architecture and Evaluation. PuLSE-DSSA – Version 2.0," Fraunhofer Institut Experimentelles Software Engineering, Kaiserslautern, Germany, IESE Report No. 038.00/E, Version 1.0, 2000.

[6] D. M. Weiss, and C. T. Robert Lai, Software Product-Line Engineering. A Family-Based Software Development Process, Addison-Wesley, 1999.

[7] FORM Method Web Site, http://selab.postech.ac.kr/publication/1998_FORM_AFeature-Oriented Reuse Method with Domain-Specific Reference Architectures.pdf

[8] http://www.sei.cmu.edu/domain-engineering/FODA.html

[9] M. Matinlassi, E. Niemelä , L. Dobrica, "Quality-driven architecture design and quality analisis method. A revolutionary initiation approach to a product line architecture," Technical Research Centre of Finland (VTT) , Oulu, Finland , VTT Report No. P456, 2002.

[10] J. Bosch: Design and Use of Software Architectures. Adopting and evolving a Product-line approach, Addison-Wesley, Great Britain, 2000

[11] C. Atkinson, I. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel, *Component-based Software Product-Line Engineering with UML*, Addison-Wesley, Great Britain, 2002.

[12] M. M. Kandé, "A concern-oriented approach to software architecture," Ph.D. Dissertation, Faculté Informatique et Communications, École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Suisse, 2003.

[13] H. Ossher, and P. Tarr, "Using Multi.dimensional Separation of Concerns to (Re)Shape Evolving Software,“ *Communications of the ACM, v*ol. 44, no. 10, October 2001, pp. 43-50.

[14] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Pinto Alarcón, J. Bakker, B. Tekinerdogan., S. Clarke, and A. Jackson, "Survey of Aspect-Oriented Analysis and Design Approaches," University of Lancaster, Lancaster, UK, AOSD-Europe-ULANC-9, Editor(s): R. Chitchyan, A. Rashid, 2005.

[15] R.E. Filman, T. Elrad, S. Clarke, M. Aksit, *Aspect-Oriented Software Development,* Addison-Wesley, USA, 2004.

[16] J. Pérez, I. Ramos, J. Jaen, P. Letelier, and E. Navarro, "PRISMA: towards quality, aspect oriented and dynamic software architectures", *Proc. 3rd International Conference on Quality Software (QSIC'03)*, IEEE, Dallas, Texas, USA, 2003, pp. 59-66.

Concern Hierarchies

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

ABSTRACT

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

1. INTRODUCTION

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

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

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

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

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

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

2. FUNCTIONAL HIERARCHIES

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

2.1 The FAMOS Structure

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

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

[1] Family of Operating Systems

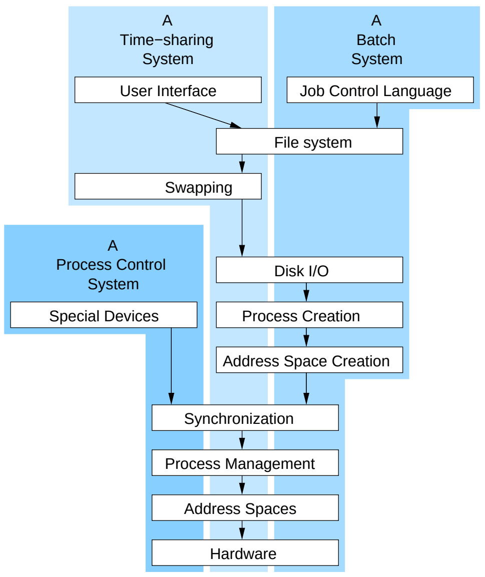


Figure 1: Functional hierarchy of the FAMOS operating system family

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

2.2 The PURE Structure

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

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

1. **Application-Driven Configuration:** Operating systems for deeply embedded systems normally have to support only one specific application. A PURE operating system was used by applications like an ordinary static C++ class library. Hence, we could exploit the C++ compiler and linker for the static system configuration. For example, if the application only instantiated the class *Native*, the code of the classes *Bundle*

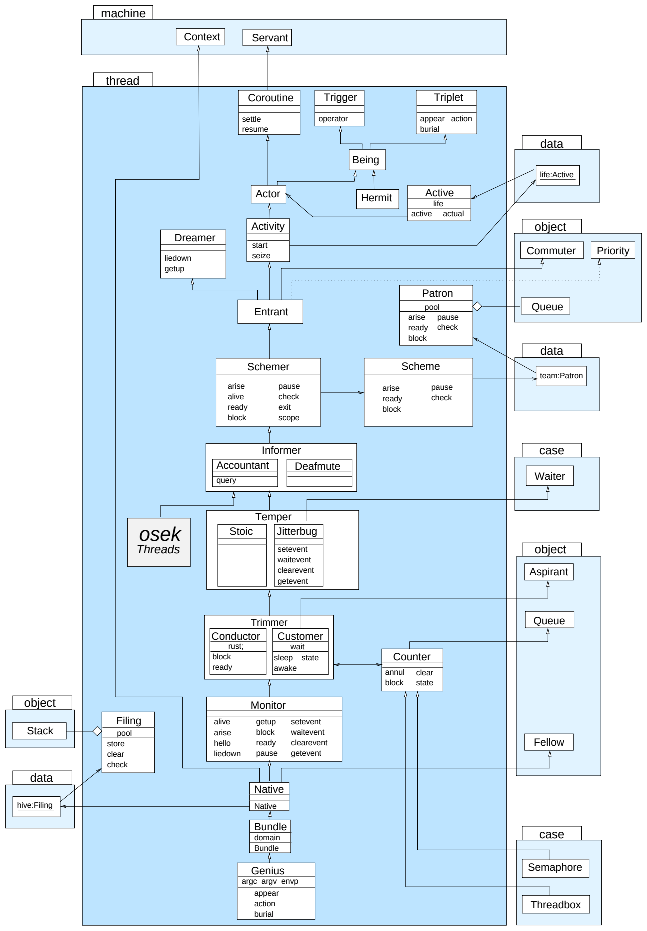


Figure 2: Class diagram of the PURE thread subsystem

and *Genius* would not be referenced and, thus, not linked into the final system.

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

layers were implemented in PURE.[2]

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

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

2.3 Lessons Learned

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

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

1. **no nested hierarchies:** highly configurable systems cannot be represented by a single deep functional hierarchy. The functions of FAMOS were rather course-grained in comparison to the functions of PURE. Therefore, nested class hierarchies would have been necessary to cope with the complexity.
2. **no crosscutting concerns:** on the modeling level functional hierarchies offer no adequate element to describe crosscutting concerns. That makes it very difficult to derive a class diagram with aspects in a systematic way.

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

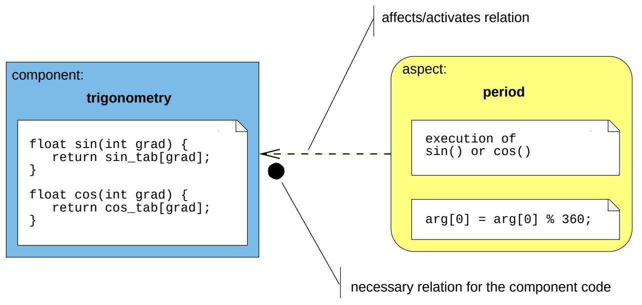
3. THE CONCERN HIERARCHY MODEL

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

3.1 Sub-Concern Modeling

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

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


affects/activates relation
component:
trigonometry
aspect:
period
necessary relation for the component code

```
float sin(int grad) {
   return sin_tab[grad];
}

float cos(int grad) {
   return cos_tab[grad];
}
```

```
execution of
sin() or cos()
```

```
arg[0] = arg[0] % 360;
```

Figure 3: Component code depends on an aspect

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

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

3.2 Crosscutting Concern Modeling

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

3.2.1 Relations Between Crosscutting Concerns and Ordinary Concerns

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

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

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

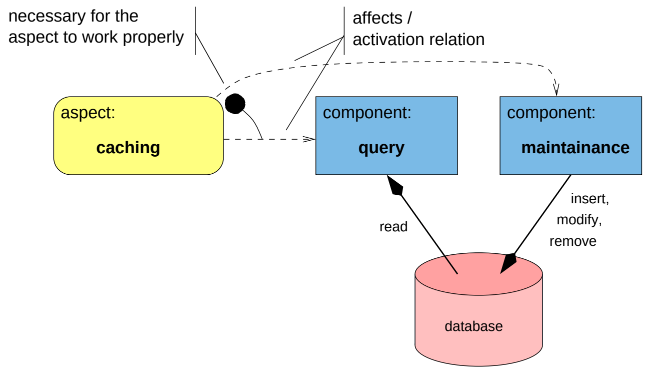


Figure 4: Aspect code depends on activation

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

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

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

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

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

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

3.2.2 Relations Among Crosscutting Concerns

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

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

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

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

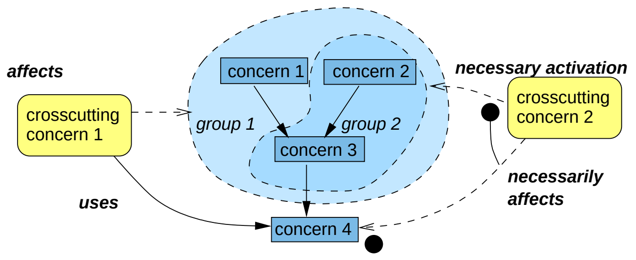


Figure 5: Concern hierarchy with concern groups

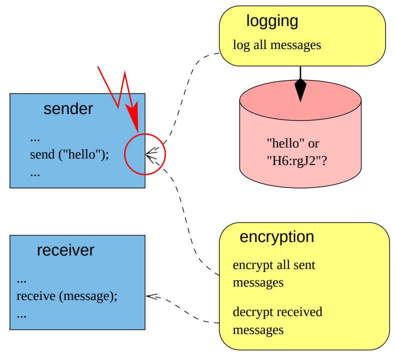


Figure 6: Interactions of crosscutting concerns

4. TOWARDS A DOMAIN DESIGN

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

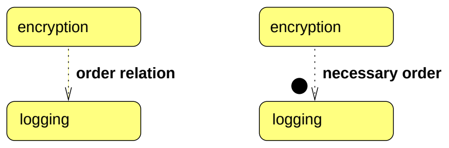


Figure 7: Normal and necessary oder relations

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

4.1 Ordinary Concern Modeling

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

4.2 Crosscutting Concern Modeling

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

4.3 Sub-Concern Modeling

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

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

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

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

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

4.4 Derivation of a Dependency Model and Tailoring

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

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

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

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

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

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

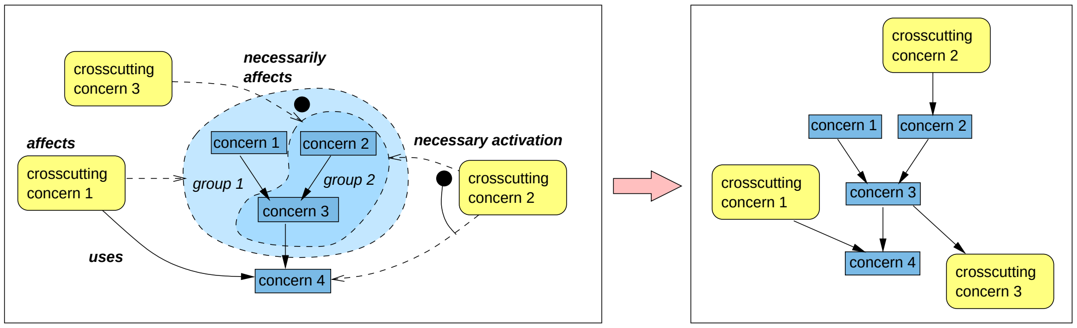


Figure 8: Derivation of the dependency model from a concern hierarchy

5. RELATED WORK

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

6. CONCLUSIONS AND FUTURE WORK

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

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

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

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

7. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

On the Structure of Crosscutting Concerns: Using Aspects or Collaborations?

Sven Apel
Department of Computer Science
University of Magdeburg, Germany
apel@iti.cs.uni-magdeburg.de

Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu

Marko Rosenmüller
Department of Computer Science
University of Magdeburg, Germany
rosenmueller@iti.cs.uni-magdeburg.de

Abstract

While it is well known that crosscutting concerns occur in many software projects, little is known about the inherent properties of these concerns nor how *aspects* (should) deal with them. We present a framework for classifying the structural properties of crosscutting concerns into (1) those that benefit from AOP and (2) those that should be implemented by OOP mechanisms. Further, we propose a set of code metrics to perform this classification. Applying them to a case study is a first to step toward revealing the current practice of AOP.

1. Introduction

While many studies have examined the capabilities of *aspect-oriented programming (AOP)* to improve the modularity, customization, and evolution of software [8, 9, 13, 14, 21, 38], little is known on *how* AOP has been used. We are interested in knowing which language mechanisms are used in current aspect-oriented programs, to what extent, and for what kinds of problems. Knowing this helps (1) build AOP tools that reflect the programmer's needs; (2) provide programming guidelines for exploiting AOP mechanisms better, i.e., what kind of crosscutting concern is implemented best using which programming mechanism; and (3) discover misuse of AOP mechanisms, which may lead to significant problems and penalties [11, 13, 14, 16, 19, 24, 28].

To address these issues we propose a framework for classifying crosscutting concerns (a.k.a. *crosscuts*). Our framework enables us to assign individual crosscuts to two distinct categories: (1) crosscuts that really demand AOP mechanisms and (2) crosscuts that can be implemented appropriately using well-known OOP mechanisms. This distinction follows a long line of prior work on *collaboration-based designs* [31,32,35], *feature-oriented programming* [4], and *design patterns* [12]. All of them advocate object-oriented mechanisms for a certain class of design and implementation problems, so called *collaborations*, which fall into one category.

We propose four metrics to analyze aspect-oriented programs to make the above distinctions, i.e., do the aspects of a program implement crosscutting concerns that really demand AOP language mechanisms? We are building a tool that will collect data from a representative spectrum of software projects that employ AOP. We discuss the data for one *AspectJ*[1] project exemplarily.

[1] http://www.eclipse.org/aspectj/

2. Crosscut Classification Framework

2.1 Homogeneous and Heterogeneous Crosscuts

A *homogeneous crosscut* extends a program at multiple join points by adding one *extension*, which is a coherent piece of code [10]. For example, an advice may advise a whole set of method executions or an inter-type declaration may introduce a field to a set of target classes (left column of Table 1).

A *heterogeneous crosscut* extends multiple join points by adding multiple extensions, where each individual extension is implemented by a distinct piece of code that affects exactly one join point [10]. For example, an aspect might bundle a set of advice that extends a set of methods, whereby each advice extends exactly one method; or an aspect bundles a set of inter-type declarations – each intended for a distinct class (right column of Table 1).

2.2 Static and Dynamic Crosscuts

A *static crosscut* extends the structure of a program statically [29], i.e., it adds new classes and interfaces as well as injects new fields, methods, interfaces, and super-classes etc.[2] Inter-type declarations are examples of static crosscuts (first row of Table 1).

A *dynamic crosscut* affects the runtime control flow of a program [29]. The semantics of a dynamic crosscut can be understood and defined in terms of an event-based model [36]: it runs additional code when predefined events occur during program execution. Such events are also called *dynamic join points* [27, 36]. A piece of advice implements a dynamic crosscut (second row of Table 1).

Basic and advanced dynamic crosscuts. Dynamic crosscuts are especially interesting when they exceed the level of known events such as method calls or executions. Work on AOP suggests that expressing a program extension in terms of sophisticated events increases the abstraction level and captures the programmer's intension more directly. There are proposals for new language constructs for defining and catching new kinds of events during the program execution [26,30]. In order to distinguish these new kinds of events and the novel language mechanisms that support them from known events in OOP, we distinguish between *basic dynamic crosscuts* and *advanced dynamic crosscuts*, which are defined as follows:

1. A basic dynamic crosscut addresses only events that are related to method calls and executions; advanced dynamic crosscuts address all other events, e.g., throwing an exception or assigning a value to a field.
2. Basic dynamic crosscuts affect a program control flow unconditionally; advanced dynamic crosscuts may specify a condition

[2] Some AOP languages do not support adding classes by aspects, e.g., AspectJ. While it is correct that one can just add another class to an environment, this is at the tool level, and is not at a model level [23].

	homogeneous	**heterogeneous**
static	`declare parents : (Line || Point) implements Shape`	`void Point.setX(int x) { /* ... */ }`
basic dynamic	`before() : execution(* set*(..)) { /* ... */ }`	`before() : execution(void Point.setX(int)) { /* ... */ }`
advanced dynamic	`before() : execution(* set*(..)) && !cflow(execution(* rotate(..))) { /* ... */ }`	`before() : execution(void Point.setX(int)) && !cflow(execution(void Line.rotate(double))) { /* ... */ }`

Table 1. A classification framework for crosscutting concerns (AspectJ examples).

that is evaluated at runtime, e.g., a method execution is only advised if it occurs in the control flow of another method execution.

3. Basic dynamic crosscuts address events known from OOP; advanced dynamic crosscuts can specify composite events that trigger the execution of an extension, e.g., *trace matches* are executed when events fire in a specific pattern thereby involving the history of computation [1].

With AOP, an advanced dynamic crosscut is implemented by an *advanced advice* and a basic dynamic crosscut by a *basic advice*. The distinction between basic and advanced advice is useful to identify which pieces of advice make use of advanced AOP mechanisms and which pieces of advice mimic well-known OOP method extensions.

3. Two Categories of Crosscutting Concerns

We argue it is crucial to decide which crosscutting concerns should be implemented as aspects, and how, and which using traditional object-oriented techniques. For that purpose, we divide the space of possible crosscuts that is defined by our classification framework into two categories, (1) those crosscuts that abstract collaborations and (2) those that address the dynamic program semantics and/or that are homogeneous. The two categories map roughly to the two programming paradigms, OOP and AOP.

3.1 Collaborations

A *collaboration* of classes is a set of classes that communicate with one another to implement a semantically coherent piece of functionality. Classes of a program play different *roles* in different *collaborations* [35]. A set of collaborating classes being added to a program can be understood as a *feature* of that program [4]. That is, a collaboration extends a program by adding new classes and by applying new roles to existing classes, whereby each role is implemented as a refinement (e.g., using *virtual classes* [25] or *mixins* [6]). From that perspective, a role adds new elements to a class and extends existing elements, such as methods.

Figure 1 depicts a collaboration-based design of a graph implementation, where the classes *Graph*, *Node*, and *Edge* collaborate together.[3] The feature *WeightedGraph* adds the class *Weight* and extends the classes *Graph* and *Edge* simultaneously. For example, the class *Edge* plays two roles, one in the *BasicGraph* and one in the *WeightedGraph*.

A significant body of work has observed that collaborations of classes are predominantly of a heterogeneous structure [4, 5, 20, 29, 32–35]. That is, the roles and classes added to a programs differ in their functionality, as in our graph example. Hence, a collaboration is a heterogeneous crosscut and a heterogeneous crosscut can be understood as collaboration applied to a program. Therefore, it is straightforward to employ from techniques for encapsulating

[3] The diagram follows the UML notation with some extensions: white boxes represent classes *or* roles; gray boxes denote collaborations; filled arrows mean refinement, i.e., to apply a role to a class.

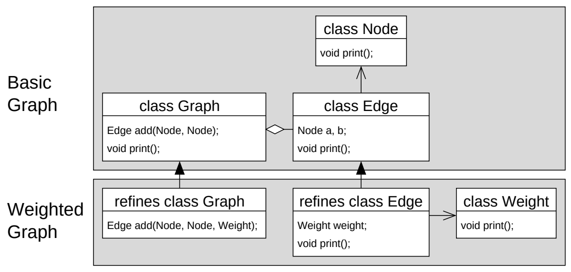


Figure 1. Collaboration-based design of a graph implementation.

and composing object-oriented collaborations when implementing heterogeneous crosscuts [6, 25, 29, 32, 35].

3.2 Homogeneous and Advanced Dynamic Crosscuts

Crosscuts that do not fall in the above category are either homogeneous crosscuts and/or advanced dynamic crosscuts.

Aspects perform well in extending a set of join points using one coherent advice or one localized inter-type declaration, thus, modularizing a homogeneous crosscut. Thereby, programmers avoid code replication. Figure 2 depicts an aspect that implements the feature *Color*, which is homogeneous. It defines an interface for colored entities (Line 2) and declares via inter-type declaration that *Node* and *Edge* implement that interface (Line 3). Furthermore, it introduces via inter-type declarations a field *color* (Line 4) and two accessor methods to *Node* and *Edge* (Lines 5-7,8-10).[4] Finally, it advises the execution of the method *print* of all colored entities to change the display the color (Lines 11-13).

```
 1  aspect Color {
 2    interface Colored { Color getColor(); }
 3    declare parents: (Node || Edge) implements Colored;
 4    Color (Node || Edge).color;
 5    void (Node || Edge).setColor(Color c) {
 6      color = c;
 7    }
 8    public Color (Node || Edge).getColor() {
 9      return color;
10    }
11    before(Colored c) : this(c) && execution(* print()){
12      Color.changeDisplayColor(c.getColor());
13    }
14  }
```

Figure 2. The feature *Color* implemented as aspect.

Advice is well-suited for implementing advanced dynamic crosscuts [29]. When advising the printing mechanism of our graph implementation we can take advantage of the sophisticated mechanisms of AOP. Background is that the *print* methods of the par-

[4] Our notation of inter-type declarations differs from AspectJ. Declaration *int (A || B).i* means that field *i* is introduced to both classes, *A* and *B*.

ticipants of the graph implementation call each other (especially, composite nodes that call *print* of their inner nodes). To make sure that we do not advise all calls to *print*, but only the top-level calls, i.e., calls that do not occur in the dynamic control flow of other executions of *print*, we can use the *cflowbelow* pointcut as conditional (Fig. 3). This is an example of an advanced advice.

```
aspect PrintHeader {
  before() : execution(void print()) &&
  cflowbelow(execution(void print())) { header(); }
  void header() { System.out.print("header:␣"); }
}
```

Figure 3. Advising *print* advanced advice.

Though language abstractions such as *cflow* and *cflowbelow* can be implemented (emulated) using traditional OOP, usually that results in code replication, tangling, and scattering.

3.3 Discussion

Table 2 depicts the guidelines for using AOP and OOP mechanisms based on their individual strengths. First, aspects should be used for modularizing homogeneous crosscuts to avoid code replication. Second, aspects avoid code scattering and tangling in case of using advanced advice for implementing advanced dynamic crosscuts. For heterogeneous crosscuts which extend only methods and classes, OOP techniques for collaboration-based designs suffice. It has been observed that although both approaches are able to implement the crosscuts of the other, they cannot do so elegantly [2, 3, 29].

	heterogeneous	**homogeneous**
static	set of roles that add elements	inter-type declaration
basic dynamic	set of roles that override methods	basic advice
advanced dynamic	set of advanced advice	advanced advice

Table 2. What implementation technique for what kind of crosscutting concern?

4. Metrics

We propose a set of metrics to provide insight into the current practice of AOP. They enable to decide in which category a given aspect falls. The metrics are quantified by the *number of occurrences (NOO)* of a certain software artifact and/or the *lines of code (LOC)* associated with it.

Classes, interfaces, and aspects (CIA). The CIA metric determines the NOO of classes, interfaces, and aspects, as well as the LOC associated with each. It tells us if aspects (as opposed to classes and interfaces) are a small or a large fraction of the used modularization mechanisms in a software project, and if these implement a significant or only a small part of the code base of that project.

Homogeneous crosscuts (HC). The HC metric measures the extent in which homogeneous and heterogeneous crosscuts are used. We calculate the fraction of advice and inter-type declarations that implement homogeneous crosscuts (NOO) and the fraction of the code base that is associated with them (LOC). The HC metric tells us if the aspects of a program exploit the pattern-matching mechanisms of AOP or merely emulate OOP mechanisms.

Advanced dynamic crosscuts (ADC). This metric determines the NOO of advanced advice and the overall LOC associated with them.[5] It tells us to what extent aspects make use of the advanced capabilities of AOP for implementing dynamic crosscuts.

Code replication reduction (CRR). The CRR metric determines the reduction in LOC when using homogeneous advice and inter-type declarations, as opposed to the LOC resulting from using traditional OOP mechanisms. The code reduction for one piece of homogeneous advice / inter-type declaration is roughly the number of affected join points, multiplied by the LOC associated with them.

5. Collecting Statistics of AspectJ Programs

CIA metric. Collecting data for the CIA metric we traverse all source files of a given project and calculate the number and LOC of aspects, classes, and interfaces – excluding blank lines and comments.

HC metric. Homogeneous crosscuts are indicated by inter-type declarations and advice that contain wildcards (* and +). If we discover logical operators in pointcuts that combine two pointcuts of the same type (e.g., *execution(...)* || *execution(...)*) then the associated advice are also counted as homogeneous. Inter-type declarations that contain logical operators are considered homogeneous as well as advice that do not qualify a target method or field completely, e.g., by omitting the type or the arguments.

ADC metric. We define all advice as advanced advice except those associated to *call* and *execution* and that are not combined with any other pointcuts, except with *target* and *args* (*execution* can also be combined with *this*). This is an overestimation that might consider some pieces of advice that are not advanced as advanced advice, but not vice versa. The remaining advice are considered basic advice.[6]

CRR metric. For determining the code reduction due to eliminating replicated code, we determine the number of join points per homogeneous advice and inter-type declaration. We multiply the number of join points minus one for each advice or inter-type declaration, with the LOC associated. Finally, we sum up the saved LOC of all advice and inter-type declarations to get the overall code reduction.

6. A Case Study

As case study we analyzed *FACET* (6364 LOC), an AspectJ-based CORBA event channel, implemented at the Washington University [18]. We used our tool *AJStats*[7] for collecting the NOO and LOC of all artifacts of FACET. We determined the properties of advice / inter-type declarations and the caused code reduction by hand.

Table 3 depicts our collected statistics. Column *NOO* lists the number of artifacts we found of a specific type (e.g., homogeneous advice) and its fraction with regard to the overall number of this type (e.g., all pieces of advice). Column *LOC* depicts the LOC associated with a certain kind of artifact and its fraction of the overall code base. In the following paragraphs we examine the data in depth.

[5] Recall that advanced advice can be either heterogeneous or homogeneous (cf. Fig. 1).

[6] Although the semantics of *call* is to advise the client side invocations of a method, it can be implemented as method extension – preconditioned that *all* calls to the target method are advised; the above definition ensures that.

[7] http://wwwiti.cs.uni-magdeburg.de/iti_db/ajstats/

metric		NOO (% of artifacts)	LOC (% of code base)
CIA	classes/int.	181 (62%)	5143 (81%)
	aspects	113 (38%)	1221 (19%)
HC	heterogen.	150 (93%)	572 (9%)
	homogen.	12 (7%)	24 (0.4%)
ADC	basic	38 (78%)	187 (3%)
	advanced	11 (22%)	110 (2%)
CRR	adv. + itds	—	534 (8%)

Table 3. FACET statistics.

CIA metric. FACET uses relatively many aspects, compared to other studies [2, 8, 9, 21, 38]. This observation is remarkable since it demonstrates that aspects are used in different software projects to a different extent. 38% of all modularization mechanisms were aspects, which occupied 19% of the overall code base.

HC metric. In FACET we found 4 of 49 pieces of advice and 8 of 113 inter-type declarations were homogeneous.[8] That is, 7% of all implemented crosscuts were homogeneous, which occupied 0.4% of the overall code base. In contrast, 93% of all crosscuts were heterogeneous, occupying 9% of the code base.

ADC metric. We found 11 of 49 advice were advanced advice. They are associated to *cflow* pointcuts or use the *returning* clause. That is, 22% of all advice were advanced advice, which occupied 2% of the overall code base. The remaining 38 advice were basic advice, which occupied 3% of the overall code base.

CRR metric. 4 pieces of advice and 8 inter-type declarations are homogeneous. We calculated the effective code reduction of 534 LOC, which is a 8% reduction compared to a version that uses OOP mechanisms for implementing homogeneous crosscuts.

7. Related Work

AOP case studies. Colyer and Clement refactored an application server using aspects [9]. Specifically, they factored 3 homogeneous and 1 heterogeneous crosscuts. While the number of aspects is marginal, the size of the case study is impressively high (millions of LOC). Although they draw positive conclusions, they admit (but do not explore) a strong relationship to collaborations.

Coady and Kiczales undertook a retroactive study of aspect evolution in the code of the FreeBSD operating system (200-400 KLOC) [8]. They factored 4 concerns and evolved them in three steps; inherent properties of concerns were not explained in detail.

Lohmann et al. examined the applicability of AOP to embedded infrastructure software [21]. For their study they factored 3 concerns of a commercial embedded operating system; 2 concerns were homogeneous and 1 heterogeneous.

Lopez-Herrejon et al. explored the ability of AOP to implement product lines [22]. They demonstrated how collaborations are translated automatically to aspects. They did note that less than 1% of their code base was attributable to heterogenous advice. They did not address in what situations which implementation technique is most appropriate nor how the generated aspects affect program comprehensibility.

Xin et al. evaluated *Jiazzi* and AspectJ for feature-wise decomposition [37]. They reimplemented FACET by replacing aspects with *Jiazzi units*, which encapsulate collaborations. They do not examine the structure of the resulting collaborations. Our analysis of FACET revealed that some crosscuts should be implemented using aspects.

[8] Note that the code associated to advice and inter-type declarations (596 LOC) is only a subset of the overall aspect code (1221 LOC), which includes also fields, methods, etc.

Metrics for AOP. Zhang and Jacobson used a set of object-oriented metrics to quantify the program complexity reduction when using AOP for implementing middleware [38].

Garcia et al. applied seven metrics to Hannemann's [17] implementation of design patterns [15]. They found that most aspect-oriented solutions improve separation of pattern related concerns, although only 4 aspect-oriented implementations have exhibited significant reuse.

Zhao and Xu propose several metrics for aspect cohesion based in aspect dependency graphs [39]. Ceccato and Tonella propose metrics for measuring the coupling degree between program elements [7].

None of the above metrics and case studies take the different structure of crosscutting concerns into account. We argue that the structure of a concern decides over how it is implemented best.

8. Conclusions

Comparatively many aspects were used in FACET and they sum up to a significant part of the code base (19%) – but only 3% of the overall code base exploits the advanced capabilities of AOP to implement homogeneous and dynamic crosscuts. 97% can be implemented straightforward using traditional OOP and collaborations. Nevertheless, the used AOP mechanisms reduce the code base by 8% compared to an OOP implementation. It follows that the plain number of aspects, advice, etc. is not meaningful to judge the successful application of AOP to a software project.

Our classification framework, categories, and metrics form a quantitative basis for analyzing aspect-oriented programs in this respect, and it can assist in exploiting the benefits of AOP. In further work we intend to analyze and compare further AOP projects to collect more data.

Acknowledgments

Sven Apel is sponsored by the German Research Foundation (DFG), project number SA 465/31-1 and SA 465/32-1. Marko Rosenmüller is sponsored by the German Research Foundation (DFG), project number SA 465/32-1. The presented study was conducted when Sven Apel was visiting the group of Don Batory at the University of Texas at Austin. Batory's research is sponsored by NSF's Science of Design Project #CCF-0438786.

References

[1] C. Allan et al. Adding Trace Matching with Free Variables to AspectJ. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[2] S. Apel and D. Batory. When to Use Features and Aspects? A Case Study. In *Proceedings of International Conference on Generative Programming and Component Engineering*, 2006.

[3] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of International Conference on Software Engineering*, 2006.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6), 2004.

[5] J. Bosch. Superimposition: A Component Adaptation Technique. *Information and Software Technology*, 41(5), 1999.

[6] G. Bracha and W. Cook. Mixin-Based Inheritance. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) and European Conference on Object-Oriented Programming*, 1990.

[7] M. Ceccato and P. Tonella. Measuring the Effects of Software Aspectization. In *Workshop on Aspect Reverse Engineering*, 2004.

[8] Y. Coady and G. Kiczales. Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2003.

[9] A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2004.

[10] A. Colyer, A. Rashid, and G. Blair. On the Separation of Concerns in Program Families. Technical report, Computing Department, Lancaster University, 2004.

[11] R. Douence, P. Fradet, and M. Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2004.

[12] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] A. Garcia et al. Separation of Concerns in Multi-agent Systems: An Empirical Study. In *Software Engineering for Multi-Agent Systems II, Research Issues and Practical Applications*, 2003.

[14] A. Garcia et al. Modularizing Design Patterns with Aspects: a Quantitative Study. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2005.

[15] A. Garcia et al. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2005.

[16] K. Gybels and J. Brichau. Arranging Language Features for More Robust Pattern-based Crosscuts. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2003.

[17] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[18] F. Hunleth and R. Cytron. Footprint and Feature Management Using Aspect-Oriented Programming Techniques. In *Proceedings of Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems*, 2002.

[19] K. Lieberherr. Controlling the Complexity of Software Designs. In *Proceedings of International Conference on Software Engineering*, 2004.

[20] K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5), 2003.

[21] D. Lohmann et al. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proceedings of the ACM SIGOPS EuroSys 2006 Conference*, 2006.

[22] R. Lopez-Herrejon and D. Batory. From Crosscutting Concerns to Product Lines: A Function Composition Approach. Technical Report TR-06-24, University of Texas at Austin, 2006.

[23] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proceedings of European Conference on Object-Oriented Programming*, 2005.

[24] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proceedings of International Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 2006.

[25] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proceedings of International Conference on Object-Oriented Programming Systems, Languages and Applications*, 1989.

[26] H. Masuhara and K. Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, 2003.

[27] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proceedings of European Conference on Object-Oriented Programming*, 2003.

[28] N. McEachen and R. T. Alexander. Distributing Classes with Woven Concerns: An Exploration of Potential Fault Scenarios. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2005.

[29] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2004.

[30] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In *Proceedings of European Conference on Object-Oriented Programming*, 2005.

[31] T. Reenskaug et al. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-Oriented Programming*, 5(6), 1992.

[32] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology*, 11(2), 2002.

[33] F. Steimann. On the Representation of Roles in Object-Oriented and Conceptual Modeling. *Data and Knowledge Engineering*, 35(1), 2000.

[34] F. Steimann. Domain Models are Aspect Free. In *Proceedings of International Conference on Model Driven Engineering Languages and Systems*, 2005.

[35] M. VanHilst and D. Notkin. Using Role Components in Implement Collaboration-based Designs. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1996.

[36] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. *ACM Transactions on Programming Languages and Systems*, 26(5), 2004.

[37] B. Xin et al. A Comparison of Jiazzi and AspectJ for Feature-Wise Decomposition. Technical Report UUCS-04-001, University of Utah, 2004.

[38] C. Zhang and H.-A. Jacobsen. Resolving Feature Convolution in Middleware Systems. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[39] J. Zhao and B. Xu. Measuring Aspect Cohesion. In *Proceeding of International Conference on Fundamental Approaches to Software Engineering*, 2004.

Towards Crosscutting Metrics for Aspect-Based Features

Roberto E. Lopez-Herrejon
Computing Laboratory
Oxford University
Oxford, England, OX1 3QD
`rlopez@comlab.ox.ac.uk`

Abstract

Features are increments in program functionality and are the building blocks of product lines. Typical implementation of features using Object Oriented techniques commonly crosscuts several classes and interfaces. There exist many techniques that implement crosscuts, of which Aspect Oriented ones distinguish themselves by their support of dynamic crosscuts expressed with advice. Despite being a core tenet of AOP, very little attention has been paid to measuring crosscuts and how they are implemented by different language constructs in particular advice. In this paper we present a semi-formal definition of a set of basic metrics to measure crosscutting in features that use aspects. Our metrics categorize features within a crosscutting spectrum that ranges from heterogeneous to homogeneous according to the relative number and types of crosscuts features implement. This categorization helps assessing the actual use of aspects for feature implementation and provides a quantitative framework to gauge at and analyze the impact of aspects for product line development. We apply our metrics to a non-trivial product line case study implemented using AspectJ and relate our results to the ongoing assessment of aspects vs. collaboration-based designs for feature implementation.

1 Introduction

Features are increments in program functionality and are the building blocks of product lines [17]. Typical implementation of features using Object Oriented techniques commonly crosscuts several classes and interfaces, fact that has attracted the attention of Aspect Oriented Programming researchers as a promising area for the application and development of AO techniques and tools. We call *aspect-based features* those features that are implemented with a set of aspects, classes, and interfaces [2][3][15][16].

Despite the increasing interest in AOP, very little attention has been paid to measuring the crosscutting nature of programs and how it is tackled by different crosscutting mechanisms of aspect languages [5].

In this paper we present a set of elementary metrics to measure crosscutting that highlights the use and contribution of pieces of advice to the overall feature and program crosscutting. We describe our metrics in a semi-formal notation using a functional programming style over a simplified abstract program structure. Our metrics categorize features within a crosscutting spectrum that ranges from heterogeneous to homogeneous according to the relative number and types of crosscuts features implement.

This categorization helps assessing the actual use of aspects for feature implementation and provides a quantitative framework to gauge at and analyze the impact of aspects for product line development. We apply our metrics to a non-trivial product line case study implemented using AspectJ and relate our results to the ongoing assessment of aspects vs. collaboration-based designs for feature implementation.

2 Crosscutting Feature Metrics

In this section we provide a semi-formal description of our crosscutting metrics. A goal of our metrics is to highlight the use and contribution of advice (the distinctive characteristic of aspect oriented languages) in the overall feature and program crosscutting.

We describe our metrics using a functional programming style (similar to Haskell [11]) over a simplified abstract program structure. This notation provides a more concise description than natural language and can serve as a guideline for the implementation of tools that automatically gather these and related metrics.

We start by describing the abstract structure of our programs, followed by the description of auxiliary functions used to define our metrics which we present at the end of this section.

2.1 Abstract Program Structure

We define a program `P` to be a set of features F_i, denoted with the following list:

$$P=[F_1,F_2,...,F_n]$$

Where `P` is of type `program` and `Fi` is of type `feature`. Figure 1 summarizes the abstract representation of our programs.

A feature `F` consists of a list of feature elements that can be classes, interfaces or aspects. A `class` is a list of `class_element` which can be of type `method`, `constructor`, etc. An `interface` is a list of `interface_element` which can be of type `field` or method declaration (`methoddecl`). An `aspect` is a list of method (`methodITD`), constructor (`constructorITD`), and field (`fieldITD`) *Inter-Type Declarations (ITDs)*, and pieces of advice. These ITDs are denoted as tuples of class and the corresponding element definition. For example, the tuple for `methodITD` is of type `(class, method)`. For pieces of `advice` we consider the pointcut expression `pce` and a `body`. We consider both named and anonymous pointcuts but we focus only on the pointcut expression formed with poincut designators and their combinations denoted with operators &&, | |, ( ), and !.

Finally we define an auxiliary type `shadow` with a tuple whose elements are a `program_element` (elements of classes, interfaces and aspects), a `class`, and a pointcut expression `pce`. A *shadow* is a place on the source code whose execution creates join points [25]. We represent a shadow with a tuple of three elements. The first element of `shadow` contains the program element that has the shadow (a `method` for example for a `execution` join point), the `class` that contains the program element, and pointcut expression `pce` that casts the shadow in that program element. This data structure is not created when programs are originally parsed, instead it is the result of a weaving mechanism.

In this paper we use only the subset of program structures of AspectJ shown in Figure 1. However this abstract program representation can be extended, the same is true for the set of auxiliary functions and metrics we describe in next subsections.

```
program :: [feature]
feature :: [feature_element]
feature_element :: class | interface | aspect

class :: [class_element]
class_element :: method | constructor | ...

interface :: [interface_element]
interface_element :: methoddecl | field

aspect :: [aspect_element]
aspect_element :: methodITD | constructorITD
                | fieldITD | advice

methodITD :: (class, method)
constructorITD :: (class, constructor)
fieldITD :: (class, field)
advice :: (pce,body)

pce :: pointcut_expression
shadow :: (program_element, class, pce)

program_element :: class_element |
        interface_element | aspect_element
```

Figure 1. Abstract Program Representation

2.2 Auxiliary Functions

The following functions provide the basic building blocks of the definitions of our metrics. Note that the names of some of these functions are the plural of the type of element they return as result.

count. This function returns the number of elements in a list. It has signature, where `a` is any type and `n` is a number:

```
count :: [a] -> n
```

union. N-ary and polymorphic disjoint set union. It receives any number of arguments, unions them and eliminates any repeated elements. We denote its signature with `n` entries of type `b` that when unioned return a list of `b` elements:

`union :: [b`$_1$`] -> ...-> [b`$_n$`] -> [b]`

sum. Receives as input a list of numbers and performs the summation on them. It has the following signature where `n` is a number:

```
sum :: [n] -> n
```

foreach. Receives as input a list and a function. which applies to all the elements in the list. It has signature (where `a` and `b` are any type):

```
foreach :: [a] -> a -> b -> [b]
```

classes. Receives a feature and returns the list of classes in that feature. It has signature:

```
classes :: feature -> [class]
```

interfaces. Receives a feature and returns the list of interfaces in that feature. It has signature:

```
interfaces :: feature -> [interface]
```

aspects. Receives a feature and returns the list of aspects in that feature. It has signature:

```
aspects :: feature -> [aspect]
```

advices. Receives as input a list of aspects and returns the list of pieces of advice contained in the aspects.

```
advices :: [aspect] -> [advice]
```

methodITDs. Receives as input a list of aspects and returns the list of method ITDs or introductions contained in the aspects.

```
methodITDs :: [aspect] -> [methodITD]
```

constructorITDs. Receives as input a list of aspects and returns the list of constructor ITDs or introductions contained in the aspects.

```
constructorITDs::[aspect] -> [constructorITD]
```

ccclasses. This function computes the *c*ross*c*utting *classes* from a list of method ITDs, constructor ITDs or field ITDs, and removes any repeated elements. It has signature (where symbol `|` stands for logical `or`):

```
ccclasses :: [ methodITD | constructorITD |
               fieldITD ] -> [class]
```

pointcuts. Receives as input a list of aspects and returns a list of pointcut expression (`pce`).

```
pointcuts :: [aspect] ->[pce]
```

shadows. This function receives as input a list of pointcuts, finds the join point shadows in a program and returns them in a list:

```
shadows :: [pce] -> [shadow]
```

sclasses. This function receives a list of shadows, extracts their classes (second elements in the shadow tuples), and removes any duplicates.

```
sclasses :: [shadow] -> [class]
```

extensions. This function receives a list of pointcuts and filters those whose shadows correspond exclusively `execution` and `call` join points. This kind of poinctuts are of particular interest in classifying features as we elaborate more on Section 3 and Section 5.

```
extensions :: [pce] -> [pce]
```

NOF. Number of features.

```
NOF (P) = count (P)
```

2.3 Feature Crosscutting Metrics

In AOP literature, an *homogenous concern* is one that applies a same piece of advice to several places; whereas an *heterogeneous concern* adds different pieces of advice to different places [4][16]. Our metrics broaden this concept to features and provide a quantitative criteria to classify features according to the number and type of crosscuts they implement.

Let `f` be a feature of a program `P`, we define the following metrics:

ECD. *Extension Crosscutting Degree*. Corresponds to the number of classes crosscut by pieces of advice whose pointcuts capture only `execution` and/or `call` join points on methods and constructors.

```
ECD(f,P) = count( sclasses(shadows(
     extensions(pointcuts(advices(aspects(f)))),
     P)))
```

FCD. *Feature Crosscutting Degree*. Corresponds to the number of classes that are crosscut by all pieces of advice in a feature and those crosscut by the ITDs.

```
FCD(f,P)= count(union(
  ccclasses( methodITDs(aspects(f))),
  ccclasses(constructorITDs(aspects(f))),
  ccclasses( fieldITDs(aspects(f))),
  sclasses(shadows(
           pointcuts(advices(aspects(f))),P))
))
```

HD. We define the *Heterogeneity Degree* of a feature as a pair of values, the Feature Crosscutting Degree `FCD` and the number of pieces of advice.

```
HD(f,P) = [FCD(f,p),count(advices(aspects(f)))]
```

HQ. We define *Heterogeneity Quotient* as the division of the number of pieces of advice by the feature crosscutting degree (or the first entry by the second entry of `HD`):

```
HQ(f,P)= HD(f,P)₁/HD(f,P)₀ if FCD(f,P)!=0
       = 1                   otherwise
```

PHQ. *Program Heterogeneity Quotient*. It corresponds to the summation of the heterogeneity quotients for all the features in a program, divided by the Number of Features NOF.

```
PHQ(P) = sum(foreach(P, λf.HQ(f,P)))/NOF(P)
```

3 Homogeneous vs. Heterogeneous Features

Let us now analyze how our metrics help categorizing features according to their crosscutting. We can depict the values for the Heterogeneity Degree (HD) as a two dimensional graph which we call *Heterogeneity Graph*. The vertical dimension is the number of pieces of advice and the horizontal dimension corresponds to feature crosscutting degree `FCD` as shown in Figure 2. Also shown in this graph, is the *Perfect Heterogeneity Line (PHL)* where the number of pieces of advice is the same as the number of classes crosscut by a feature.

By plotting the features that constitute a program into an Heterogeneity Graph, it is possible to gauge how crosscutting capabilities are used for its implementation. If most features cluster around the perfect heterogeneity line, that is an indication that the program makes little use of advice crosscutting capabilities of aspects.

If the plotting of a feature falls into the area above the `PHL`, then that feature employs a larger number of advice yet it is crosscutting the same classes many times. If it falls in the area below the line, it means that the feature is using advice that crosscut more than one class. Falling either above or

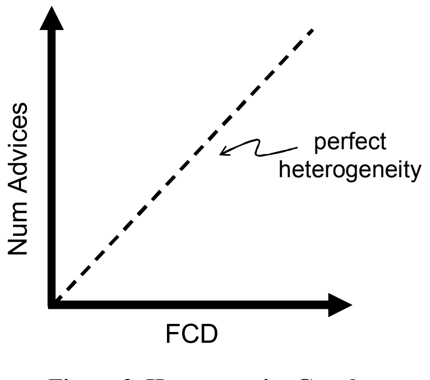


Figure 2. Heterogeneity Graph

below the PHL does not necessarily imply a subuse or misuse of aspect crosscutting.

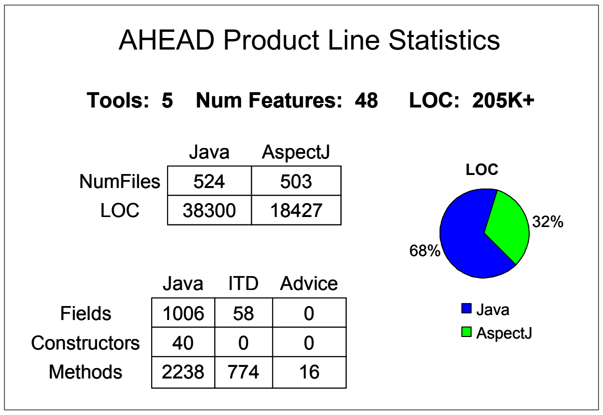
Quantitatively, if the Program Heterogeneity Quotient or PHQ tends to value 1 the program is making little advice crosscutting capabilities of aspects. Also, if PHQ tends to value 0, it can have two interpretations: a) that the program has very few pieces of advice that crosscut many classes (think the case of tracing), or b) many class crosscuttings are due to inter-type declarations (features cluster around the horizontal axis).

Another measurement that helps us assess the use of aspects in features is Extension Crosscutting Degree or ECD. If the ECD of a feature approximates its Feature Crosscutting Degree value (FCD) then such feature most likely could be implemented with OO extension mechanisms. We elaborate more on this issue in Section 5.

4 AHEAD Case Study

We applied our metrics to a non-trivial product line. The *AHEAD Tool Suite (ATS)* is a set of stand alone and language-extensible tools [1] which implement *Feature Oriented Programming (FOP)*, a technology that studies feature modularity in program synthesis for product lines [10]. We reimplemented five core tools of ATS using AspectJ to which we applied our metrics [24].

The core tools are formed with 48 features, implemented with 524 standard Java classes and interfaces amounting to 38300 LOC, and 503 aspects with 18427 LOC. This gave us a ratio of 68% to 32% of Java and aspect code respectively. The total LOC generated for the five tools analyzed are 205K+ LOC. To the best of our knowledge, we are not aware of any product line in AspectJ of scale comparable to this case study. Most of the aspect code ATS uses is for adding fields (58) and methods (774) using ITDs and only 16 pieces of advice where utilized. In terms of LOC, these pieces of advice account for less than 1% of the total LOC of the product line. These statistics are summarized in Figure 3.


AHEAD Product Line Statistics

Tools: 5 Num Features: 48 LOC: 205K+

	Java	AspectJ
NumFiles	524	503
LOC	38300	18427

	Java	ITD	Advice
Fields	1006	58	0
Constructors	40	0	0
Methods	2238	774	16

LOC
68%
32%
Java
AspectJ

Figure 3. ATS Product Line Statistics

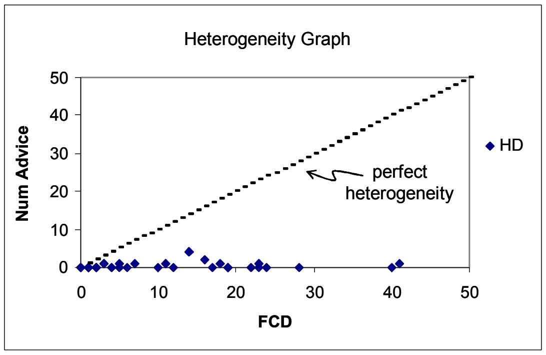


Figure 4. ATS Heterogeneity Graph

The Heterogeneity Degree values computed for the 48 features of ATS are depicted in Figure 4. As expected given the statistics of ATS, features cluster around the horizontal dimension. This indicates a number of inter-type declarations significantly larger than the number of pieces of advice. Furthermore the histogram of Heterogeneity Quotient (HQ) values, shown in Figure 5, clearly illustrates the fact that most of ATS features do not utilize advice, HQ value 0. Symmetrically, four features provide only standard classes significantly, HQ value 1. The value of the Program Heterogeneity Quotient (PHQ) for ATS is 0.12 which in this case corroborates that most of ATS crosscutting is due to ITDs.

5 Collaborations and Heterogeneous Features

A *collaboration* is a set of objects and a protocol that determines how the objects interact. The part of an object that enforces the protocol in a collaboration is called a *role* [28][30]. Collaboration-based designs have a long history of research [20][27][29][30]. One of their goals is to provide a more flexible modularity unit to improve reuse in multiple configurations or compositions for the development of different programs. Thus collaborations are mechanisms to implement features for product lines [9].

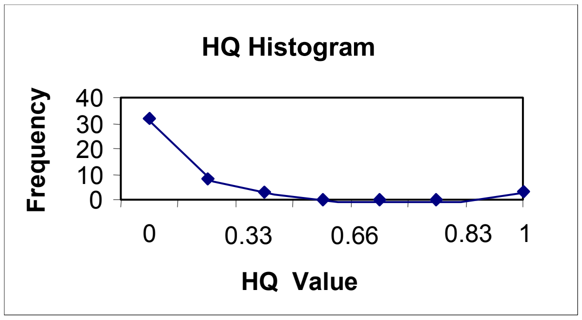


Figure 5. Heterogeneity Quotient Histogram for ATS

Collaborations can be implemented using several Object Oriented techniques. The kinds of program increments these techniques support are ultimately bound by the Object Oriented ideas they rely upon (i.e. inheritance, polymorphism, encapsulation, etc.). A technique that implements collaborations is FOP and its implementation in AHEAD [10]. FOP categorizes features into *base features* which contain standard classes and interfaces, and *function features* which contain *class extensions* and *interface extensions*. This type of extensions or *refinements* add new functionality to existing classes and interfaces respectively. Let us illustrate these concepts with the following examples. Consider class `C` defined as follows (where `s1`, and `s2` stand for any statements):

```
class C{
   int f;
   void m() { s1; s2 }
}
```

Consider now a refinement to this class, denoted in AHEAD with keyword `refines` (where `s3` and `s4` stand for any statement):

```
refines class C {
   double g;
   int n() {...}
   void m() { s3; Super.m(); s4; }
}
```

Let us analyze the structure of this extension. The first line adds to class `C` new field `g` while the second adds new method `n`. The type of program increment on the third line is called a *method extension*. It extends the functionality of method `m` in the following way. Note statement `Super.m()`. This statement has a similar effect to `super` in standard Java method overriding, namely, executing the method that is being overridden in a super class. However in this context, it describes the execution of a method being extended that was defined in a previous feature in a composition chain. Thus the composition of class `C` and its class extension can be conceived as:

```
class C{
   int f;
   double g;
   int n() {...}
   void m() { s3; s1; s2; s4}
}
```

The result is that field `g` and method `n` are added to class `C`, and method `m` is extended. Notice that an execution of `m` would execute first statement `s3`, followed by `s1` and `s2` which correspond to executing the original method `m`, and then by `s4`. Extensions to constructors are handled similarly. The implementation of this composition model is described in [1][10].

A feature in FOP is a collection of classes, interfaces, and their extensions, thus is crosscutting in nature. Notice however that a class extension only adds and extends elements to a single class, fact which makes FOP features heterogeneous as they add different program fragments to different places.

As we have shown, describing addition of new methods, fields and constructors, method extensions, and constructor extensions is straightforward using collaboration-based techniques. Using our example of class `C` extension we now illustrate how these program changes are implemented using AspectJ. This is done as follows [22][24]:

```
aspect C_extension {
   double c.g;
   int c.n( ) {...}
   void around() : execution (int C.m())) {
      s3; proceed(); s4;
   }
}
```

Adding fields and methods is implemented as inter-type declarations, whereas method extensions and constructor extensions rely on advice that captures the `execution` join points of the methods or constructors they extend. Notice in this example that we use `proceed` to mimic the semantics of FOP's `Super`.

However, there are several factors that complicate the use of advice for the implementation of constructor and method extensions. In the case when a method or constructor has arguments, their values must be captured using an `args` pointcut and included on the parameters list of the advice so that it can be passed properly in a `proceed` call. Also, there are subtle issues in the semantics of `execution` and `call` join points that limit the reusability of extensions defined in this way [24]. Another important issue for composition of features implemented using aspects is advice precedence management [23]. Precedence is the mechanism that AspectJ provides to define composition order for method and constructor extensions [22]. Unfortunately, current rules of AspectJ advice precedence make defining composition order (in the general case) a non-trivial and error-prone task which is exacerbated as the number of features increases [6][23].

These are some of the arguments that have been used to suggest that the types of increments that resemble method and constructor extensions as well as adding new fields, methods, and constructors can be better handled by collaboration-based techniques [5]. Furthermore, it has been proposed to use aspects in combination with collaboration-based techniques to better exploit the strengths of both approaches [4].

However, other than for the most straightforward cases: pieces of advice associated to control-flow pointcuts, and those that capture only `execution` or `call` join points that do not use type patterns; it is unclear how and when to argue that collaboration-based techniques or advice are better suited for particular instances of crosscutting implementation. We believe the metrics presented in this paper can serve as a foundation for devising a set of quantitative guidelines and heuristics to determine which of both approaches are better tailored to implement the different kinds of crosscutting requirements features may have. For instance, a feature with pieces of advice whose Heterogeneity Quotient HQ tends to value 1 may be better implemented using collaboration-based techniques.

Throughout the years, in the products lines that we have developed using different technologies, most features appear to be inherently heterogeneous [5][24]. We frame this empirical finding in the form of a conjecture:

> **Heterogeneous Nature of Features Conjecture**. Most features in feature-based programs and product lines are of heterogeneous nature regardless of the technology used for their implementation.

Intuitively, the reason behind this conjecture is that large programs are not synthesized by adding the same piece of code in different places, but rather, adding different pieces of code in different places.

The application of our crosscutting metrics to several aspect-based programs and product lines, developed by us and others, would definitively provide quantitative evidence to support or refute our conjecture. Incidentally, it would strengthen or weaken the argument that Aspects Oriented languages can significantly benefit from collaboration-based designs techniques, as proposed by *Aspectual Mixin Layers (AML)* [4], and already supported by *CaesarJ* which implements a variation of mixin layers [12]. This line of work is part of our future research.

6 Related Work

Several metrics have been proposed for aspects. Zhao and Xu describe metrics for aspect cohesion based on aspect dependences graphs [32]. Zhao also utilizes a similar framework to define measurements for aspect coupling [31]. Their metrics are formally described, however they lack concrete architectural interpretation and, to the best of our knowledge, have not been applied to actual case studies.

Coupling metrics have been proposed by Ceccato and Tonella [13]. They extend and adapt to AOP some of Chidamber and Kemerer's metrics for Object Oriented systems [14]. This set of metrics is defined informally and it is applied to a tiny case study (250+ LOC), Hannemann's implementation of the Observer Pattern [19]. However, its is unclear how these metrics would extrapolate to larger case studies and their architectural significance.

Bartsch and Harrison evaluate five metrics in Ceccato and Tonella's work [8]. They argue that only one of the evaluated metrics can be considered well-defined (lacks any interpretation ambiguities), and none of them are completely valid from a measurement theory point of view. Along the same lines, Mehner proposes a series of steps to validate AOP metrics and their application [26].

An extensive study on modularizing design patterns have been performed by Garcia et al. [18]. They use Hannemann's implementation of GoF patterns to apply seven metrics that extend and adapt to AOP Chidamber and Kemerer's metrics [14]. Their metrics are informally defined and their results are given an interpretation in terms of improvement of separation of concerns and reuse. However, we believe that design patterns, though important and common programming practices, offer a limited perspective on the actual and potential use of aspects and their architectural relevance. We argue that this perspective can be broaden by applying their metrics to larger case studies.

Coupling metrics for AOP certainly depend on the crosscutting capabilities of aspects. Our metrics focus only on crosscutting relations produced by pointcut shadows and ITD's, and do not consider cases such as method calls or field references which the above coupling metrics account for.

A different approach to analyze modularity in aspect designs is being studied by Lopes and Bajracharya [21]. They adapt the theory of modular design proposed by Baldwin and Clark [7] to aspect orientation and report that on certain cases AOP techniques can add value to the design. However it is unclear how this analysis relates to crosscutting and feature heterogeneity.

7 Conclusions and Future Work

In this paper we present a semi-formal description of a set of crosscutting metrics for aspect-based features. Our metrics categorize features within an spectrum from heterogeneous to homogeneous depending on the relative type and number of crosscuts features implement. We applied our set of metrics to a non-trivial case study. This study supports our conjecture that regardless of implementation techniques most features are inherently heterogeneous.

To provide more evidence to corroborate or refute our conjecture, it is necessary to analyze and measure more actual case studies. We intend to collect as many publicly available programs of significant size (1K+ LOC) as possible for measurement and analysis. Such endeavour would provide more arguments in the aspects vs. collaboration-based techniques assessment.

We plan to extend our set of metrics to address issues such as cohesion and coupling for features. These extended metrics could help identify opportunities for feature refactoring.

Currently our measurements are gathered manually. We are exploring different possibilities for developing tool support. Our goal is to develop tool infrastructure that would allow the implementation of these and other metrics in a simple and extensible way.

Acknowledgments. We thank Sven Apel for this comments on earlier drafts of this paper.

8 References

[1] AHEAD Tool Suite (ATS). *http://www.cs.utexas.edu/users/schwartz*

[2] V. Alves, P. Matos, L. Cole, P. Borba, and G. Ramalho. Extracting and Evolving Game Product Lines. *SPLC* 2005.

[3] M. Anastasopoulus, and D. Muthig. An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. *ICSR* 2004.

[4] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. *ICSE 2006*.

[5] S. Apel and Don Batory. When to Use Features and Aspects? A Case Study. To appear, *GPCE 2006*.

[6] AspectJ, *http://eclipse.org/aspectj/*.

[7] C.Y. Baldwin, and K.B. Clark. Design Rules vol I. The Power of Modularity. MIT Press, 2000.

[8] M. Bartsch and R. Harrison. An Evaluation of Coupling Measures for AspectJ. LATE Workshop *AOSD 2006*.

[9] D. Batory, R. Cardone, and Y. Smaragdakis. Object-Oriented Frameworks and Product-Lines. *SPLC 2000*.

[10] D. Batory, J.N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, June 2004

[11] R. Bird. Introduction to Functional Programming using Haskell. Prentice Hall, 1998.

[12] CaesarJ. *http://www.caesarj.org/*

[13] M. Ceccato and P. Tonella. Measuring the Effects of Software Aspectization. *First Workshop on Aspect Reverse Engineering. Delft, The Netherlands*, 2004.

[14] S. Chidamber and C. Kemerer. A Metrics Suite for OOD Design. *IEEE Transactions on Software Engineering, 20(6), 1994*

[15] A. Coyler and A. Clement. Large-scale AOSD for Middleware. *AOSD (2004)*.

[16] A. Coyler, A. Rashid and G. Blair. On the Separation of Concerns in Program Families. TRCOMP-001-2004, Computing Department, Lancaster University, UK (2004)

[17] K. Czarnecki, and U.W. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[18] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. Transactions on TAOSD I. LNCS 3880, 2006.

[19] J. Hannemann. AspectJ implementation of GoF patterns. *http://www.cs.ubc.ca/~jan/AODPs*

[20] I. Holland. Specifying Reusable Components using Contracts. E*COOP 1992*.

[21] C. Lopes, and S.K. Bajracharya. An Analysis of Modularity in Aspect Oriented Design. *AOSD 2005*.

[22] R.E. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Techniques. *ECOOP* 2005.

[23] R.E. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. *PEPM* 2006.

[24] R.E. Lopez-Herrejon, and D. Batory. From Crosscutting Concerns to Product Lines: A Function Composition Approach. Tech. Report UT Austin CS TR-06-24. May 2006

[25] H. Masuhara and G. Kiczales. Modeling Crosscutting Aspect-Oriented Mechanisms. *ECOOP (2003)*

[26] K. Mehner. On Using Metrics in the Evaluation of Aspect-Oriented Programs and Designs. LATE Workshop associated to *AOSD 2006*.

[27] T. Reenskaug, E. Anderson, A. Berre, A. Hurlen, A. Landmanrk, O. Lehne, E. NOrdhagen, E. Ness-Ulseth, G. Ofdetal, A. Skaar, and P. Stenslet. OORASS : Seamsless Support for the Creation and Maintenance of Object-Oriented Systems. Journal of Object Oriented Programming, 5(6): October 1992

[28] Y. Smaragdakis and B. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM TOSEM April 2002.

[29] M. Van Hilst and D. Notkin. Using C++ Templates to Implement Role-Based Designs. JSSST International Symposium on Object Technologies for Advanced Software. Springer-Verlag, 1996.

[30] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. OOPSLA 1996.

[31] J. Zhao. Measuring Coupling in Aspect-Oriented Systems. Technical Report SE-142-6. Information Processing Society of Japan (IPSJ), June 2003.

[32] J. Zhao and B. Xu. Measuring Aspect Cohesion. FASE 2004.

The Role of Aspects in Modeling Product Line Variabilities

Jing (Janet) Liu
Department of Computer Science
Iowa State University
1-515-294-2735
janetlj@cs.iastate.edu

Robyn R. Lutz
Department of Computer Science
Iowa State University and
Jet Propulsion Laboratory/Caltech
1-515-294-3654
rlutz@cs.iastate.edu

Hridesh Rajan
Department of Computer Science
Iowa State University
1-515-294-6168
hridesh@cs.iastate.edu

ABSTRACT

As of today, it is unclear whether aspect-oriented modeling can benefit the model-driven development of software product lines. Although some preliminary studies exist at the requirements and implementation level that investigate the interaction of crosscutting behaviors and product-line variabilities, to the best of our knowledge these interactions at the modeling level are not yet investigated. The contribution of this work is a preliminary study of the object-oriented and aspect-oriented approaches for handling crosscutting variabilities. This study helps us identify desired characteristics of aspect-oriented modeling techniques for product lines. A pacemaker product line, extracted from the real industry case, serves as a running example to illustrate our findings.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design – *Representation.*

General Terms

Design, Standardization.

Keywords

Model-Driven Development, Software Product Lines, Variability, Aspect, Aspect-Oriented Modeling.

1. INTRODUCTION

Model-driven development (MDD) [29], [30], [37] has played a very important role in software product-line engineering [18], [20], [26]. The executable models help exemplify the requirements, detect design flaws, validate the effects of variability management and help future maintenance [31]. However, the variability realization techniques in this area are geared toward local variabilities [7], [18]. We define crosscutting variabilities as those whose realizations are "fragmented across a system" [36]. We define local variabilities as those that can be captured in a modularized, object-oriented software artifact (e.g., a use case, an architectural block, a class, etc.) in the dominant decomposition. The lack of designated mechanisms for handling crosscutting variabilities in the product-line modeling level has hindered the sufficient support of all types of variabilities in MDD and created a void in validating design decisions regarding them.

Aspect-Oriented Software Development (AOSD) [15], [21], [23] has emerged as a promising solution for handling crosscutting concerns [47] in all phases of the software development lifecycle. Several approaches [5], [24], [25] have already extended AOSD into product-line requirements and implementation. Thus, it is natural to seek to combine Aspect-Oriented approaches with MDD in software product line practice.

This paper conducts a preliminary study of the Object-Oriented (OO) and Aspect-Oriented (AO) approaches in handling crosscutting, behavioral variabilities. A pacemaker product line, extracted from a real industry case, is used to illustrate our findings. For example, we observe that in this product line the AO approach handles variabilities that have a common mechanism but differ in locations better than the OO approach, if an automatic weaving mechanism is provided. However, the AO approach does not necessarily support more variability than the OO approach.

The rest of this paper is organized as follows. Section 2 presents needed background information. Section 3 introduces the running example (a pacemaker product line) and reports experience in modeling the crosscutting behavior using OO and AO techniques. Section 4 discusses our observations to some open problems found during the case study. Section 5 provides related work. Finally, Section 6 concludes and describes future work.

2. BACKGROUND

Model Driven Development (MDD) [29], [30], [37] is a software development approach that uses diagrams to communicate and uses models to understand and validate the designs, as well as to help software implementation, deployment and maintenance. It often uses the Object-Oriented paradigm [28] to abstract the system functionality into models.

A software product line is a set of software systems developed by a single company that share a common set of core requirements yet differ amongst each other according to a set of allowable variations [12], [49]. The product-line engineering concept is advantageous in that it exploits the potential for reusability in the analysis and development of the core and variable requirements in each member of the product line. Variability is the part of the software artifact that makes a product line member differ from others [49]. Four main approaches used to model variability in product lines are [48]: parameterization, information hiding, inheritance, and variation points. These approaches can be readily integrated with component engineering [34] and MDD [18]. However, none of them is designed to address those variabilities that crosscut multiple software artifacts. (See Sect. 3 for an example.)

AOSD [15], [21], [23] is emerging as a way to complement the traditional Object-Oriented Software Development by modularizing crosscutting concerns in a new software artifact called an aspect [23]. The places where an aspect crosscuts a software system are called join points [23].

Existing work in this area has covered a broad spectrum of the software development processes for single systems, from requirements analysis [8], [11], [25], [32], [38], architectural design [22], [40], modeling [1], [6], [45], coding [5], [24], and testing [50], [51]. Because of its ability in handling crosscutting concerns, AOSD is a natural candidate to manage crosscutting variabilities in a product line setting.

3. CASE STUDY

In this section we first present the running example, then give some concrete crosscutting variabilities and describe the modeling process using the OO and the AO approaches. Some findings are provided at the end.

3.1 Pacemaker Product Line

We use a pacemaker product line to evaluate different techniques for modeling crosscutting variabilities. These are real-time, embedded, and safety critical systems, that have been successfully developed in industry using MDD and software product line practices [26]. Studying the modeling techniques used for its variabilities can not only help enhance the safety assurance level of such systems, but may also yield observations that raise our confidence in other similar systems.

A pacemaker is an embedded medical device designed to monitor and regulate the beating of the heart when it is not beating at a normal rate. It consists of a monitoring device embedded in the chest area as well as a set of pacing leads (wires) from the monitoring device into the chambers of the heart [14]. In our simplified example, the monitoring device has three basic parts: a sensing part (sensor) that senses heart beat, a stimulation part (pulse generator) that generates pulses to the heart, and a controlling part (controller) that configures different pacing and sensing algorithms and issues commands.

In this example, we only consider a single-chambered product line of pacemakers that does pacing and sensing in the heart's ventricles. More advanced pacemakers can be dual-chamber, and the pacing or sensing algorithms applied to each chamber can be different although highly coordinated. In our case study, we consider three different products within this product line:

BasePapcemaker: A BasePacemaker has the basic functionality shared by all pacemakers: generating a pulse whenever no heart beat is sensed during the sensing interval.

ModeTransitivePacemaker: A ModeTransitivePacemaker can switch between Inhibited Mode and Triggered Mode during runtime. In the Inhibited Mode, the pacemaker acts exactly like a BasePacemaker. In the Triggered Mode, a pulse follows every heartbeat. (Triggered Mode is mainly used in therapies for dual-chamber pacemakers.)

RateResponsivePacemaker: A RateResponsivePacemaker acts similarly to the BasePacemaker but can adjust its sensing interval according to the patient's current activity level: LRLrate, denoting the Lower Rate level for a patient's normal activities and URL rate, denoting the Upper Rate Level when a patient is exercising.

3.2 An Example of Crosscutting Variability

Many of the major components in a pacemaker have to log their critical events into an EventRecorder component for use in making therapy decisions either by the pacemaker or by the doctors [14]. However, different pacemakers log different events at different relative or absolute times. Event Logging is a crosscutting variability whose functionality is shared among different components in each pacemaker system. Requirements and features for this product line are specified in [27] using a Commonality and Variability Analysis (CVA), as part of the FAST approach [49]. The excerption of CVA for the event logging is presented in Table 1. The variabilities and commonalities are detailed in Table 2.

Table 1. Excerpts from pacemaker product line Commonality & Variability Analysis

Commonality 1. A pacemaker shall log average heart rate sensed every fixed recording interval at the BaseSensor component.
Commonality 2. A pacemaker operating in Inhibited mode shall record the pulse width of every pulse being generated at the PulseGenerator component.
Variability 1. A pacemaker operating in Triggered mode shall record the average number of pulses generated every fixed recording interval at the PulseGenerator component.
Variability 2. A pacemaker with an extra sensor shall record the percentage of the pacemaker sensing at LRLrate every fixed recording interval at the ExtraSensor component.

Table 2. Event Logging Variability & Commonality

Product Name	Component Name	Events to Log
Base Pacemaker	Base Sensor	Average heart rate sensed every fixed recording interval
	Pulse Generator	The pulse width of every pulse being made
Mode Transitive Pacemaker	Base Sensor	Average heart rate sensed every fixed recording interval
	Pulse Generator	1) In the Triggered mode, the average number of pulses generated every fixed recording interval 2) In the Inhibited mode, the pulse width of every pulse being generated
Rate Responsive Pacemaker	Base Sensor	Average heart rate sensed every fixed recording interval
	Pulse Generator	The pulse width of every pulse being made
	Extra Sensor	The percentage of the pacemaker sensing at LRLrate every fixed recording interval

3.3 Modeling using OO techniques

The Object Management Group (OMG) [33] uses UML [10] as a standard language for the Model-Driven Architecture [30]. In this section, we are using the UML 2.0 statechart [10] to model crosscutting variabilities. Statechart was preferred over other modeling artifacts for two reasons. First, it is particular suitable for detailed behavioral modeling. Second, it is close to implementation and is crucial in generating executable models to validate the design. The successful modeling in statecharts not only guides the implementation, but also provides assurance for later stages.

The following subsections describe the process of incremental modeling [27] of the crosscutting variabilities in different products. It is supported by the Rhapsody software modeling environment [13] from I-Logix. We start from the product that has the fewest variations (i.e., the BasePacemaker), and then incrementally build the model with variations of other products in the product line.

3.3.1 *BasePacemaker*

Based on the UML statechart model for the pacemaker product lines described in our previous work [27], we add the behavior of the EventRecorder of the BasePacemaker using the statechart shown in Fig. 1. It is composed of three orthogonal statecharts [10]: the BaseSensorCounting and BaseSensorRecording subcharts for recording the average heart rate at every recordingInterval, and the PulseGeneratorRecording subchart for recording the pulse width every time a pulse is generated.

In order to get the pulse width value (denoting how long the pulse lasts), which is a private attribute of the PulseGenerator Class, the PulseGenerator has to send this value explicitly as a parameter of the evPulseDone message (Fig. 2). The “show(params->width)” in Fig. 1 is a function that prints the value of the parameter named “width” (which is the parameter of “evPulseDone”).

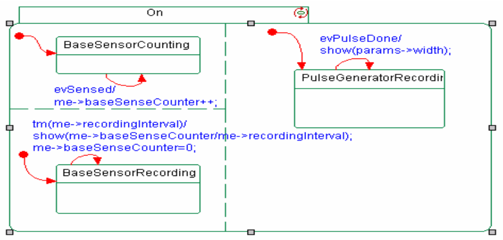


Figure 1. BasePacemaker’s EventRecorder

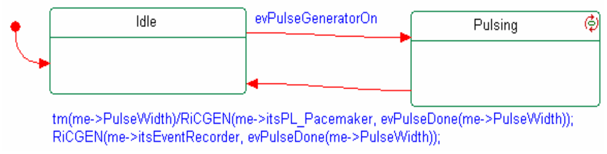

Idle
evPulseGeneratorOn
Pulsing
tm(me->PulseWidth)/RiCGEN(me->itsPL_Pacemaker, evPulseDone(me->PulseWidth));
RiCGEN(me->itsEventRecorder, evPulseDone(me->PulseWidth));

Figure 2. BasePacemaker’s PulseGenerator

3.3.2 *ModeTransitivePacemaker*

The statechart of EventRecorder in the ModeTransitivePacemaker, shown in Fig. 3, is created by inheriting [18] the EventRecorder’s statechart from the BasePacemaker. Variability 1 in Table 1 (mode transitive) is modeled by adding a condition connector [10] (the symbol of a circle with a “C” inside) in the sub-chart for pulse recording, and by adding a new subchart of pulse counting. The sub-chart of mode transitions (InhibitedMode and TriggeredMode) is created due to the need to keep the mode attributes local (as a private member, required by the modeling tool Rhapsody [13], as well as a common practice in Object- Oriented software development).

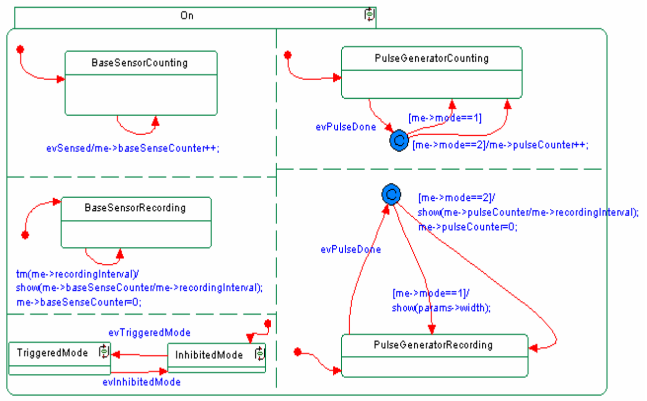


Figure 3. ModeTransitivePacemaker’s EventRecorder

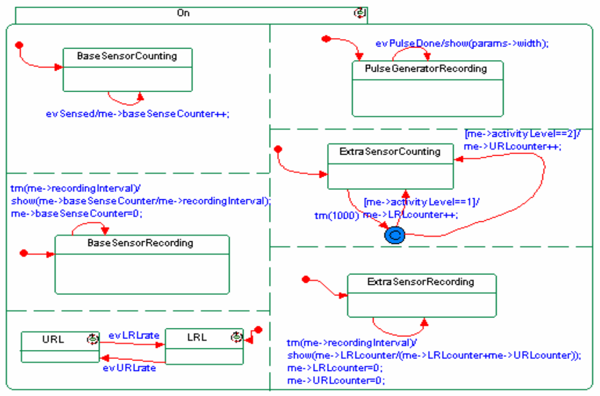


Figure 4. RateResponsivePacemaker’s EventRecorder

3.3.3 *RateResponsivePacemaker*

There are two ways to implement the statechart for the EventRecorder in the RateResponsivePacemaker. The first is to create an EventRecorder statechart for the whole product line (we call it PL_EventRecorder) by introducing the Variability 2 in Table 1 (the rate responsive variability) into the EventRecorder statechart of ModeTransitivePacemaker via transitions with condition connectors. This way the PL_EventRecorder becomes a parameterized state model [18] for the whole product line. This method is described in detail in our previous work [27]. The second way is to inherit the statechart of EventRecorder in BasePacemaker. As a result, each product member has its own statechart deriving from a base statechart (the BasePacemaker’s). These two ways are the common choices in modeling variabilities using statecharts in a software product line [18]. For ease of illustration of the variability we show the statechart generated using the second method in Fig. 4.

As seen in Fig. 4, Variability 2 in Table 1 is modeled by adding a sub-chart for ExtraSensor counting and recording separately. As in the ModeTransitivePacemaker, a sub-chart of activity level (URL and LRL) is created.

Thus, in the OO approach, the EventRecorder component acts similarly to an Observer Pattern [16]: it monitors all the triggering events and then dispatches them to their separate handlers (orthogonal sub-charts).

3.4 Modeling using AO techniques

Due to the lack of standard AO modeling techniques and support for weaving mechanism, we use UML sequence diagrams together with textual descriptions to demonstrate the behavior of an aspect. Sequence diagrams [10] capture the dynamic view of a system. They show a set of roles and the messages that are passed between instances of the roles. Sequence diagrams have been used before to demonstrate the behavior of aspects [6], [11], [43]. In this case, the sequence diagram serves as an abstraction to demonstrate the characteristics of common AO techniques.

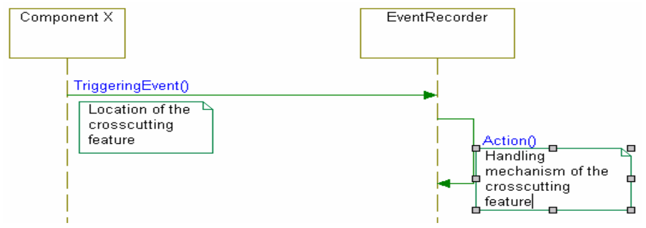

Component X
EventRecorder
TriggeringEvent()
Location of the crosscutting feature
Action()
Handling mechanism of the crosscutting feature

Figure 5. Generic Scenario of the EventRecorder Aspect

The EventRecorder component in our example system encapsulates the crosscutting variability of event logging. Therefore, we choose to model this component as an aspect. The generic scenario of the EventRecorder aspect is depicted in Fig. 5. It is composed of two parts: the *triggering event*, which is the location where the aspect crosscuts (call it "*location*"), and the *action*, which is the behavior of the aspect after being triggered (call it "*mechanism*"). Table 3 illustrates the different locations and mechanisms for the EventRecorder aspect in the product line (the events and action names are the abstraction of their counterparts in Fig. 1, 2 and 3). Table 3 shows that several locations share similar mechanisms. Table 4 helps demonstrate this in a clearer fashion.

We make the following observations by comparing Table 3 and Table 4:

1) Each group of locations that share a similar mechanism can be modeled as a "pointcut" [23], while the similar mechanism can be modeled as an "advice" [23]. By "similar" we mean that they behave the same except for the context to which they apply. For example, the counter incrementing behavior in different components is similar, except for the variable it increments.

In some cases, mechanisms differ significantly at different locations. For example, the mechanism for the location "RateResponsivePacemaker -> ExtraSensor -> recording interval timeout" differs from the second mechanism in Table 4 because the first takes the sum and the second takes the average. These mechanisms cannot be modeled as a single advice.

2) Here, where there is only a single crosscutting variability, the mechanisms do not overlap. This is because, even if two mechanisms apply to the same locations, they happen under different conditions. Thus, it does not make much difference whether we model each of the matching pointcut and advice pairs (as described above) in a separate aspect or model all of them in one aspect.

Table 3. Aspect Specification

Product Name	Component Name	Aspect	
		Join Point	**Advice**
Base Pacemaker	Base Sensor	Sensed	counter increases by one
		recording interval timeout	record the average counter value during the recording interval, then reset the counter
	Pulse Generator	Pulse	record the pulse width
Mode Transitive Pacemaker	Base Sensor	Same as in BasePacemaker	
	Pulse Generator	Pulse	1) if in Inhibited mode, same as BasePacemaker 2) if in Triggered mode, counter increases by one
		recording interval timeout	1) if in Inhibited mode, do nothing 2) if in Triggered mode, record the average counter value during the recording interval, then reset the counter
Rate Responsive Pacemaker	Base Sensor	Same as in BasePacemaker	
	Pulse Generator	Same as in BasePacemaker	
	Extra Sensor	1 msec timeout	1) if in LRLrate, LRLrate counter increases by one 2) if in URLrate, URLrate counter increases by one
		recording interval timeout	Record the ratio of the LRLrate counter value to the sum of the LRLrate and URLrate counter values, then reset the counters

Table 4. Mechanism Classification

Mechanism	Location	Condition
Counter residing in the same component as the location increases by one	1) BasePacemaker->BaseSensor->sensed event 2) ModeTransitivePacemaker->BaseSensor->sensed event 3) ModeTransitivePacemaker->PulseGenerator->sensed event 4) RateResponsivePacemaker -> BaseSensor ->sensed event 5) RateResponsivePacemaker -> ExtraSensor -> 1 msec timeout	3) if in Inhibited Mode 5) if in LRLrate, increase LRLrate counter; if in URLrate, increase URLrate counter
Record the average counter value during the recording interval, then reset the counter	1) BasePacemaker -> BaseSensor -> recording interval timeout 2) ModeTransitivePacemaker -> PulseGenerator -> recording interval timeout 3) RateResponsivePacemaker ->BaseSensor -> recording interval timeout	2) if in Triggered Mode
Record the pulse width	1) BasePacemaker -> PulseGenerator -> pulse event 2) ModeTransitivePacemaker -> PulseGenerator -> pulse event 3) ModeTransitivePacemaker -> PulseGenerator -> pulse event	2) if in Triggered Mode

However, if we introduce another crosscutting variability into the product line, it is likely that the mechanisms from the two variabilities will overlap in locations. In that case, conflict resolving techniques are needed. These could be similar to the feature interaction handling mechanisms [35] for local variabilities, but we have to bear in mind that such conflict resolution will apply invasively in the AO setting (rather than locally as in the OO setting). In fact, the tool support for aspect interaction at the programming level [3], [39] may be migrated to the modeling level.

3) The locations to which a crosscutting variability applies to can be fragmented within and across a product. For example, locations that share a similar mechanism can reside in different components of the same product, or in components from different products. This means that the scope of the join point (as well as the weaving) needs to be extended to the product-line level, rather than the product level as in traditional AOSD.

4) There are two ways that a condition can affect the mechanism. In the first way (seen in the first condition in the first mechanism group in Table 4) the condition serves as a *switch* to decide whether an event is able to trigger the action. In the second way (seen in the second condition in the same group) the condition uses *context* information passed to tell where the action should apply. Consequently, these two types of conditions need to be modeled differently. This remains an open problem for our future work.

3.5 Findings

Some similarities and differences between the OO approach and AO approach are observed as follows:

1. Both approaches handle the crosscutting variability in a centralized manner. The OO approach invokes the methods explicitly while the AO approach handles it implicitly [17], [44].

2. The OO approach requires each component being monitored to send its local variable values explicitly via messages, since the local variables are private in the OO paradigm. However, in the AO approach, the aspects are allowed reflective access to certain variables at the join points, such as the executing object, the target of a call, arguments of a method, etc. Explicitly sending these variables is not necessary in the AO approach.

3. In the OO approach, the location where the handling mechanism takes place (after the triggering event) must involve a component other than the component that sends the triggering event. However in the AO approach, there is no such restriction. This is due to the similar reason as above.

4. In the AO approach, if we treat different locations that share a similar mechanism as join points for the same aspect, modeling variabilities that have a common mechanism but differ in locations will be easier than in the OO approach, assuming automatic weaving mechanisms are provided. This is because in the OO approach, users have to manually adapt the variability into the local context, while in the AO approach users simply need to add some new join points. This is true for variabilities both within a product and across several products. In this situation, the AO approach makes the modeling of crosscutting features more reusable across the software product line.

5. The AO approach does not support more variability than the OO approach, since each different handling mechanism requires a separate advice. With many variations in the handling mechanism, both the AO approach and the OO approach incur significant overhead. Creating aspect templates or generic aspects helps reuse, but does not accommodate more variabilities.

4. DISCUSSION

In this section we give some suggestions for the weaving mechanism in the modeling level, as well as two open problems confronted in this work. Finally, a set of criteria for future empirical studies is proposed.

4.1 Weaving Mechanism

Without concrete weaving mechanisms, no executable models can be generated from the AO modeling. Weaving at the modeling level also provides a way to generate models independent of implementation languages. Based on our experiences using Rhapsody [13] as an OO modeling tool, we propose some suggestions for the weaving mechanism at the modeling level.

1. The effect of the aspect should be able to be demonstrated in the animation of the executable model. In other words, users should be able to model the aspect and the rest of the system separately and see the effect of weaving in the animation.

2. Users should be able to choose to implement the aspect weaver themselves by building it in the models, or to choose an existing weaver. For the latter, users should be able to turn it on or off.

3. Users should be able to view the marked join point, attributes and methods (advices) introduced by aspects statically in the system model, even though they cannot use them other than in the aspect.

4.2 Open Problems

The first open problem is about whether to model local features (e.g., switching to Inhibited Mode during runtime) at the product-line level using the AO approach. As suggested in Section 3.5, if we extend the scope of weaving and join point to the product line level, e.g., advising several product members using one aspect, we can achieve greater reuse of the crosscutting features. That raises the question of whether we can and should do the same for those local features. Some preliminary case studies can be found at [2] and [4].

The second problem is how much obliviousness a modeler should have about the weaving process. Unlike the coding stage, the modeling stage calls for exemplification of the design intent. Therefore we expect more knowledge about the weaving process to be exposed in the modeling level than in the AOP level. However, how much is enough remains a problem for future research.

4.3 Evaluation Criteria

In this section we propose the criteria for comparing the capability of different modeling techniques for crosscutting variabilities. The metrics introduced here, while preliminary and partial, identify some criteria that may be useful in subsequent, more empirical evaluations.

Feasibility

This criterion evaluates if it is easy or possible to model all types of crosscutting variabilities. In order to do this, a taxonomy of crosscutting variabilities needs to be provided. Anastasopoulos and Muthig [3] have done an initial step by classifying variations into two types: “positive” and “negative”, denoting the effect of variability on the system (i.e., adding vs. removing functionalities).

Degrees of variability

This denotes how flexible the modeling technique is for modeling the variability. Note that the OO and AO approaches can have different notions of “flexibility”. For instance, in the OO approach, binding time [46] is used to describe how late developers are able to change a variability (or select a variant at a specific variation point). However, this notion is not very meaningful for the AO approach as most aspects are bound at compilation time and the rest at load time or run time.

Therefore, we propose to measure the degree of crosscutting variability by evaluating the limitation of mechanisms and diversity of locations where variability can occur. (Point 4 of Section 3.5 provided such an example.)

Evolution

This is an important issue in software product lines. Specifically, we need to evaluate if an approach supports changing requirements and the addition of new product-line members. This can be done by checking the likely impact introduced by a change.

Executable model

As stated at the beginning of this paper, executable models are very important in clarifying the design intent and validating design logics. This is an indispensable part in MDD. We examine this criterion by checking if the modeling language provides sufficient support for describing behaviors and if code generation (for both the system and environment) is available.

Tool support

Tool support is crucial in making an approach scalable, especially in a product line setting. With sufficient tool support, the code generation should be automatically done. Moreover, users should be able to run the executable model and check it against the requirements scenarios [13].

Cost

Just as in product-line engineering, where too few products do not provide a gain via reuse [49], a new modeling technique for the crosscutting features does not necessarily always save time and money. We need to identify the situations when it will receive the biggest gain and maybe provide a pay-off model for such a technique.

5. RELATED WORK

Existing work that introduced the concept of aspects into software product-line development include [3], [5], [9], [19], [24], [25], and [39].

The work by Apel et. al. [5] combines the force of Feature Oriented Programming (FOP) and Aspect Oriented Programming (AOP) in the code level. Loughran and Rashid [24] propose ‘framed aspects’ as a technique combining AOP, frame technology and Feature-Oriented Domain Analysis (FODA). Both [5] and [24] compare the aspect-oriented approach with the approach they propose to combine with (mixin layers and frame respectively) and conclude that they complement each other. This also backs up our findings that OO and AO variability modeling techniques complement each other, such as AOP and OOP.

Anastasopoulos and Muthig [3], as well as Saleh and Gomaa [39], present evaluations of the use of AOP in the implementation of software product lines. Concrete tool support is provided for automatic weaving [39] or configuration [3].

Griss [19] proposes a feature-driven analysis to find aspects as crosscutting features at the high level and map them into code fragments in the components in the low level. The feature analysis provides the traceability document through the development cycle.

Loughran et. al. [25] introduce NAPLES, a tool that uses natural language processing and aspect-oriented techniques to derive feature-oriented models (including features, aspects, variabilities and commonalities in a given domain) from requirements.

Batory et. al. [9] models the components of distributed simulations as aspects, via the help of DSLs and GenVoca PLAs.

A significant amount of work has been devoted to aspect-oriented modeling for single systems, e.g., [6], [21], [41], [42], [43], and [51].

However, none of the above work addresses the role of aspects in the model-driven development of product lines in contrast to the traditional OO approach, as we do here.

6. CONCLUSION

The work described here provides a preliminary comparison of the OO and AO approaches in modeling crosscutting variabilities, based on experience with a product line case study. Several observations are made that may be helpful for future research. Possible future work includes tools for resolving aspect conflicts, more empirical evaluations of the two approaches, a rigid weaving mechanism, and its implementation in an existing MDD tool.

7. ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under grants 0204139, 0205588 and 0541163.

8. REFERENCES

[1] Aldawud, O., Bader, A., and Elrad T. Weaving with Statecharts. in *the Aspect-Oriented Modeling with UML workshop at the 1st Int'l Conf. on Aspect-Oriented Software Development* (Enschede, The Netherlands, 2002).

[2] Alves, V., Matos, P. Jr., and Borba, P. An Incremental Aspect-Oriented Product Line Method for J2ME Game Development *in the Workshop on Managing Variabilities Consistently in Design and Code at the 19th OOPSLA,* (Vancouver, Canada, 2004).

[3] Anastasopoulos, M., and Muthig, D., An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. in *Software Reuse: Methods, Techniques and Tools: 8th Int'l Conf., ICSR 2004* (Madrid, Spain, 2004), Springer Berlin / Heidelberg, 141-156.

[4] Apel, S., and Batory, D.,When to Use Features and Aspects? A Case Study. In *Proc. GPCE 2006* (Portland, USA, 2006).

[5] Apel, S., Leich, T., and Saake, G., Aspectual Mixin Layers: Aspects and Features in Concert. in *the 27th ICSE.* (Shanghai, China, 2006), ACM Press, 122 – 131.

[6] Araújo, J., Whittle, J., and Kim, D. Modeling and Composing Scenario-based Requirements with Aspects. in *the 12th IEEE International Requirements Engineering Conference* (Kyoto, Japan, 2004), IEEE Press, 58-67.

[7] Atkinson, C. et. al. *Component-based Product Line Engineering with UML*. Addison-Wesley Professional, 2001.

[8] Baniassad, E. et. al. Discovering Early Aspects. *IEEE Software, 23,* 1 (Jan. 2006), 61-70.

[9] Batory, D., et. al.. Achieving extensibility through product-lines and domain-specific languages: a case study. *ACM Transactions on Software Engineering and Methodology, 11,* 2 (April 2002), 191-214.

[10] Booch, G., Rumbaugh, J., and Jacobson, I. *The Unified Modeling Language User Guide*. Addison-Wesley Professional, 2005.

[11] Clarke, S., and Baniassad, E. *Aspect-oriented Analysis and Design: The Theme Approach,* Addison-Wesley, Upper Saddle River, 2005.

[12] Clements, P., and Northrop, L. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[13] Douglass, B. P. *Doing Hard Time Developing Real-Time Systems with UML, Objects, Frameworks and Patterns.* Addison-Wesley, 1999.

[14] Ellenbogen, K.A., and Wood M.A. *Cardiac Pacing and ICDs*. Blackwell Publishing, Malden, 2005.

[15] Filman R. E., Elrad, T., Clarke, S., and Aksit, M. et. *Aspect-Oriented Software Development*. Addison-Wesley Professional, 2004.

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

[17] Garlan, D., and Notkin, D. Formalizing Design Spaces: Implicit Invocation Mechanisms. in *VDM '91: Formal Software Development Methods,* (Noordwijkerhout, The Netherlands, 1991), Springer-Verlag, 31-44.

[18] Gomaa, H. *Designing Software Product Lines with UML: From Uses Cases to Pattern-Based Software Architectures.* Addison-Wesley, Boston, 2005.

[19] Griss, M. L., Implementing Product-line Features By Composing Component Aspects. in the *First International Software Product Line Conference* (Denver, USA, 2000). Kluwer 2000, 271-289.

[20] Guidant Corporation Keeps Its Rhythm With Statement MAGNUM, I-Logix, 2002. Retrieved August 7, 2006, from Iowa State University: http://www.ilogix.com/pdf/success/Statemate_GuidantCorporationKeepsItsRhythm.pdf.

[21] Jacobson, I., and Ng, P. *Aspect-Oriented Software Development with Use Cases.* Addison-Wesley Professional, Upper Saddle River, 2004.

[22] Katara, M., and Katz, S. Architectural Views of Aspects. in the 2nd *Int'l Conf. on Aspect-oriented Software Development* (Boston, USA, 2003), ACM Press, 1-10.

[23] Kiczales, G. et. al., Aspect-Oriented Programming. in *the 11th European Conference on Object-Oriented Programming* (Jyväskylä, Finland, 1997), Springer-Verlag, 220-242.

[24] Loughran, N., and Rashid, A., Framed Aspects: Supporting Variability and Configurability for AOP. *in the 8th International Conference on Software Reuse* (Madrid, Spain, 2004), Springer, 127-140.

[25] Loughran, N., Sampaio, A., and Rashid A., From Requirements Documents to Feature Models for Aspect Oriented Product Line Implementation. *MoDELS 2005 International Workshop on MDD in Product Lines* (Montego Bay, Jamaica, 2005), Springer, 262-271.

[26] Liu, J., Lutz, R., and Thompson J, Mapping Concern Space to Software Architecture: A Connector-Based Approach. in

ICSE 2005 Workshop on Modeling and Analysis of Concerns in Software (St. Louis, USA, 2005), ACM SIGSOFT Software Engineering Notes (Volume 30, Issue 4), 1 – 5.

[27] Liu, J., Dehlinger, J., and Lutz, R. Safety Analysis of Software Product Lines using State-based Modeling. in *the 16th IEEE International Symposium on Software Reliability Engineering* (Chicago, USA, 2005), IEEE Press, 21-35.

[28] McgGregor, J. and Korson, T. Understanding Object-Oriented: A Unifying Paradigm. *Communication of the ACM, 33,* 9 (Sept. 1990), 40-60.

[29] Model-Driven Software Development, May 2006. Retrieved August 7, 2006, from Iowa State University: http://www.mdsd.info/mdsd_cm/page.php?page=intro&id=5.

[30] Mukerji, J., and Miller, J. The MDA Guide v1.0.1. *OMG Papers on the MDA,* June 2003. Retrieved August 7, 2006, from Iowa State University: http://www.omg.org/docs/omg/03-06-01.pdf.

[31] Niemann, S. Executable Systems Design with UML 2.0. *OMG Whitepapers on UML*, I-Logix, August 2004. Retrieved August 7, 2006, from Iowa State University: http://www.omg.org/news/whitepapers/ Executable_System_Design_UML.pdf

[32] Nuseibeh, B., Crosscutting Requirements. in *the 3rd International Conference on Aspect-oriented Software Development* (Lancaster, UK, 2004), ACM Press, 3-4.

[33] The Object Management Group (OMG), August 2006. Retrieved August 8, 2006, from Iowa State University: http://www.omg.org/.

[34] Pohl, K., Böckle, G., and van der Linden, F. J. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, Berlin, 2005.

[35] Prehofer, C. An Object-Oriented Approach to Feature Interaction. in *the 4th IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems* (Montréal, Canada, 1997), IOS Press, 313-325.

[36] Rajan, H. and Sullivan, K, Classpects: Unifying Aspect- and Object-Oriented Language Design. *in the 27th International Conference on Software Engineering*1(St. Louis, USA, 2005), The ACM Digital Library, 59 – 68.

[37] Schmidt, D. C. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer 39,* 2 (Feb. 2006), 25-31.

[38] Rashid, A. et. al. Modularization and Composition of Aspectual Requirements. in the 2nd *International Conference on Aspect-oriented Software Development* (Boston, USA, 2003), ACM Press, 11-20.

[39] Saleh, M., and Gomaa, H., Separation of concerns in software product line engineering. in *ICSE 2005 Workshop on Modeling and Analysis of Concerns in Software* (St. Louis, USA, 2005), ACM SIGSOFT Software Engineering Notes (Volume 30, Issue 4), 1 – 5.

[40] Shomrat, M., and Yehudai, A. Obvious or not? Regulating architectural decisions using aspect-oriented programming. in the 1st *International Conference on Aspect-oriented Software Development* (Enschede, The Netherlands, 2002), ACM Press, 3-9.

[41] Sillito, J., Dutchyn, C., Eisenberg, A., and K. DeVolder. Use case level pointcuts. In *Proc. ECOOP 2004,* (Oslo, Norway, 2004).

[42] Stein, D., Hanenberg, S., and Unland, R., Position Paper on Aspect-Oriented Modeling: Issues on Representing Crosscutting Features. in *the 3rd International Workshop on Aspect-Oriented Modeling* (Boston, USA, 2003).

[43] Stein, D., Hanenberg, S., and Unland, R., On Representing Join Points in the UML. in *the 2nd International Workshop on Aspect-Oriented Modeling with UML* (Dresden, Germany, 2002).

[44] Sullivan, K., and Notkin, D. Reconciling Environment Integration and Software Evolution. *ACM Transaction on Software Engineering and Methodology, 1,* 3 (July 1992), 229-268.

[45] Sutton, S.M., and Rouvellou, I. Modeling of Software Concerns in Cosmos. in *the 1st International Conference on Aspect-oriented Software Development* (Enschede, The Netherlands, 2002), ACM Press, 127-133.

[46] Svahnberg, M., van Gurp, J., and Bosch, J. A taxonomy of variability realization techniques: Research Articles. *Software-Practice & Experience, 35,* 8 (July 2005), 705-754.

[47] Tarr, P., Ossher, H., Harrison, W., and Sutton, S. M. Jr., N Degrees of Separation: Multi-Dimensional Separation of Concerns. *in 21st Int'l Conf. on Software Engineering* (Los Angeles, USA, 1999), ACM Press, 107-119.

[48] Webber, D., and Gomaa, H. Modeling Variability in Software Product Lines with the Variation Point Model. *Science of Computer Programming, 53,* 3 (Dec. 2004), 305-331.

[49] Weiss, D. M., and Lai, C. T. R. *Software Product Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.

[50] Xie, T., and Zhao, J. A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs. in *the 5th International Conference on Aspect-oriented Software Development* (Bonn, Germany, 2006), ACM Press, 190-201.

[51] Xu, D., and Xu, W. State-based Incremental Testing of Aspect-oriented Programs. in *the 5th International Conference on Aspect-oriented Software Development* (Bonn, Germany, 2006), ACM Press, 180-189.

Using Graph-Rewriting for Model Weaving in the context of Aspect-Oriented Product Line Engineering

Florian Heidenreich
Dresden University of Technology
Software Technology Group
01062 Dresden, Germany
florian.heidenreich@inf.tu-dresden.de

Henrik Lochmann
SAP Research CEC Dresden
Chemnitzer Str. 48
01187 Dresden, Germany
henrik.lochmann@sap.com

ABSTRACT

In this paper we present the concept of combining feature models and solution models through aspect-oriented graph rewriting systems (AO-GRS) in the context of product-line engineering (PLE). Variable parts of software systems are often modelled by feature models in PLE. In Model Driven Development a feature is represented by means of model elements in a solution model, e.g. classes, methods or attributes in the context of UML models. The inclusion of a feature in the resulting software system may change the solution model of the product on multiple points and thereby crosscut the solution model. We use GRS-based model weaving to include specific features in a solution model based on the presence or absence of the feature in the variant model. We demonstrate the feasibility of our approach in a case study, which uses story diagrams as GRS.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering (CASE); D.2.2 [**Design Tools and Techniques**]: Object-oriented design methods

General Terms

Design, Languages

Keywords

AOSD, Product Line Engineering, Feature Models, Graph Rewriting

1. INTRODUCTION

Variable parts of software systems in PLE are often modelled by feature models [14], which describe the variabilities of the products in a product line. Enabling a feature often has a direct impact on the resulting solution model of the software product, because it may require including new model elements in the solution model. These changes are often not localised to one specific part of the model but are scattered over different packages or classes and thereby crosscut the solution model. For example, including a new attribute in a business class may also require changes on the graphical user interface of the application. This brings up the need for defining the necessary changes on the model as pluggable, traceable and well localised transformation aspects on model level.

Models, e.g. UML class diagrams, are graphs describing software systems. These models can be manipulated by graph rewrite systems (GRS) which have a strong formal background, such as criteria for termination, confluence and unique normal forms. GRS have been recognised as a powerful technique for specifying complex transformations that can be used in various stages of the software development process. Aßmann uses GRS for program optimisation [5], Radermacher for application of design patterns [30] while Christoph does transformation of software designs in the context of the *Model Driven Architecture* (MDA) via GRS [11]. Aßmann and Ludwig are using GRS for constructing aspect weavers that work on implementation level [7]. We use GRS to construct weavers to integrate aspectual features chosen from the feature model to the core solution model of a software system to build products in a product line.

We start with a simple example, which demonstrates that graph rewrite rules can construct model weavers. After an introduction to feature modelling and model-driven software engineering in Section 3, we present our idea of model weaving with GRS in Section 4. We explain the usefulness of our approach with a case study on using GRS for model weaving within the Fujaba tool suite [18] in Section 5. Section 6 presents some related work, before we draw our conclusion and discuss future work in Section 7.

2. INFORMAL EXAMPLE

Before we explain our approach in detail, we describe the idea of model weaving via GRS by means of a small example.

Assuming that a feature from a feature model requires the inclusion of a new method in a specific class of the solution model. For example, in a banking system, implementing a minimum balance rule may require a method for calculating the available balance in addition to extra attributes for representing the minimum balance. A model weaver should be able to find the specific class, create the required method and insert this method in the class. Figure 1 depicts a simple weaver, consisting of a graph rewrite system with only one rule. The graph rewrite rule is composed of a pattern, which matches a class with a specific name, and a pattern for creating a new method. Hence, it represents the concrete aspectual feature from the feature model. Whenever a class in the underlying model is found, whose class name corresponds to the name mentioned in the pattern, the method from the aspect pattern is linked to the methods of the matched class. The underlying graph is shown in Figure 2. We can use the graph rewrite rules to automatically generate model weavers as described in detail in Section 4.

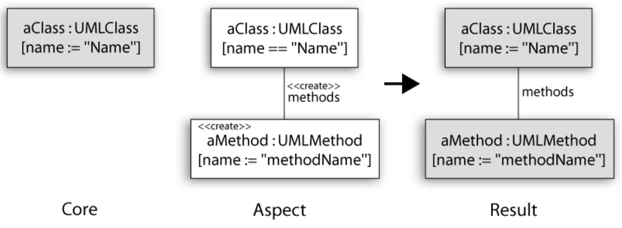


Figure 1: Graph rewrite rule for weaving a method into a class

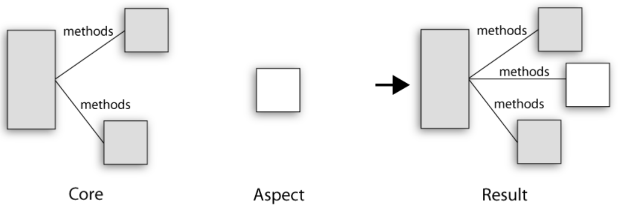


Figure 2: Graph visualisation of the weaving process

3. FEATURE AND SOLUTION MODELS

Product-line engineering in a software intensive context focuses on software engineering by developing and using core assets rather than creating products from scratch [24, 25]. To develop such assets, it is necessary to define and describe commonalities and variabilities in product lines. After the introduction of the *Feature-Oriented Domain Analysis* (FODA) [23] these product line variations were often expressed by *feature models.*

3.1 Feature models

A *feature model*, as the result of domain analysis, contains information about mandatory and optional parts of software and provides an abstract, concise and explicit representation of variability in a product line [14]. Feature models consist of concrete features, which are of certain feature types. The FODA feature types are: *mandatory*, *alternative*, *optional* and *or-features.* While *mandatory* features represent required parts of products in a product line, the remaining feature types describe variable ones. The interactions and dependencies between features are described in a hierarchical manner. With the help of Kang's graphical notation [23], feature models are shown as unranked trees, where the connection between parent and child features illustrates the type and relationship of child features in the same hierarchy level.

Figure 3 shows the very simple feature model we use in the case study described in Section 5. It consists of one *mandatory* root feature *communication*, followed by three *or-features email*, *postcard* and *SMS.* The *or-feature* in feature models establishes a dependency between parent and children in a [1-n] manner. That means at least one feature of a group of *or-features* must be included, while the other features of the group remain optional. Applied to the example in Figure 3, a certain product that follows the shown feature model has to implement at least *one* communication feature (e.g. *email*) and is allowed to implement other communication features as well, while this is not obligatory.

While a feature model describes the variability of a complete product line, a *variant model* can be understood as an instance of a feature model. Hence, variant models describe concrete products of a product line. Applied to the example above, a possible variant model would be one which includes the communication features *email* and *SMS* but does not include support for regular mail communication.

3.2 Solution models

In the last few years, omnipresent problems such as heterogenous platforms, code duplication, and insufficient documentation led to the idea of model-driven software engineering. The key idea is to develop solutions for a general problem domain instead of concrete, context specific scenarios that depend on underlying technologies. Therefore, abstract *solution models* of software systems are created for a certain problem domain. These models represent technology-independent systems, which form the basis for code generation and enable developers to understand unknown software systems more quickly. Different approaches have implemented this concept, the *Model Driven Architecture* [28] and *Model Driven Software Development* [33] in general along with supporting frameworks like the *Eclipse Modeling Framework* [15].

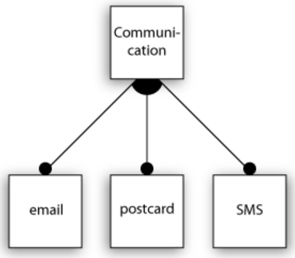


Figure 3: Example feature model in FODA feature diagram notation

In these approaches, system development starts with the creation of a model for a problem domain[1]. This model is either a *meta-model* or an instance of another meta-model. Meta-models form the basis for the creation of solution models by defining syntactic and semantic modelling constraints.

[1]As mentioned by Czarnecki in [12] regarding problem and solution space, the terms problem and solution domain are relative terms. A solution domain can be treated as problem domain on another stage of the software development process.

UML, as an example for a general purpose modelling language, defines the UML meta-model, which constrains the creation of domain specific models that may be instances of class, sequence or collaboration diagrams [29].

3.3 Need for integration

The abstract level of model-driven software engineering suits for an application to PLE because it allows a technology independent and high level way of developing software systems, which is demanded by feature models that describe variability in PLE. The realisation of products in a product line postulates the implementation of certain features in designed solution models and thus the combination of both feature models and solution models.

4. MODEL WEAVING WITH GRS

The main motivation behind weaving on the model level in the context of product-line engineering is combining the feature model and the solution model. The feature model thereby parameterises the core solution model, the variant-independent model (VIM), with features and extends the variability points of the model to form a variant-specific model (VSM) as shown in Figure 4. We use GRS for weaving aspectual features into the solution model. Before we describe our approach in detail, some basic graph rewriting terminology is presented in the next section.

4.1 Basic terminology

We choose *relational graph rewriting* as our graph rewrite system, in which graphs represent both entities of the solution model and aspects [4]. Nodes represent these entities and aspects and are linked by multiple relations, where a *context-free pattern* is a finite connected graph. A *context-sensitive pattern* is a graph of at least two disconnected subgraphs. For example matching a specific pattern in a graph may require the existence of another pattern. A *host graph* is the graph to be rewritten by a GRS. A *redex* is a subgraph of the host graph injectively homomorphic to a pattern. A *graph rewrite rule* r = (L,R) consists of a left-hand side pattern L and a right-hand side graph R. L is matched against the host graph, i.e. its redex located in the host graph. In a rewriting step, a redex is replaced by the parts specified in the right-hand side of the rule.

4.2 Model Weaving

As explained in [7], GRS can be used to construct aspect weavers. Model-based AO-GRS provide a simple formalism for weaving aspects, which rely on properties directly recognisable from the structure of the aspect specifications and the entities from the model. All nodes and edges as well as their properties form the join-point model of AO-GRS. These join points are addressable through the redexes of a context-free or context-sensitive pattern in the host graph. A graph rewrite rule can be considered as an aspect weaver. Most often, the left-hand pattern of a graph rewrite rule contains at least one pattern addressing join points in the host graph and one pattern describing the aspectual part of the graph rewrite rule, the aspect pattern. The weaving operation attaches the aspect pattern to the redexes of the host graph.

To combine the feature model and the solution model, we express each feature from the feature model as an aspect of the solution model. Since we use GRS, each aspect is represented by a graph rewrite rule. These rules are formulated on the meta-model level of the solution model, because we need access to the language constructs used to form the solution model. The graph rewrite rules can be just textual representations or built of graphical languages and modelled right beside the core solution model of the product line. We use the latter in our case study, which we describe in Section 5.

In model weaving for PLE, each aspect is woven according to the presence or absence of a feature in the variant model. If a feature is enabled and thereby present in the variant model, the corresponding graph rewrite rule is applied to the solution model. The graph rewrite rule can introduce new model elements by creating the representations of these new elements on meta-model level. This concept can be partially compared to the inter-type declarations of the AspectJ language [1], but furthermore it allows introducing all kinds of elements defined by the target model's meta-model.

We use the UML meta-model for class diagrams [29] as the graph model for our solution, but it is possible to exchange this by any other meta-model, e.g. the Ecore meta-model [15]. GRS are also not limited to models describing static behaviour of a system like class diagrams. Graph-rewriting can be applied to UML state, activity or sequence diagrams and even to the diagrams of domain specific languages (DSLs), which are based on a meta-model known by the software architect.

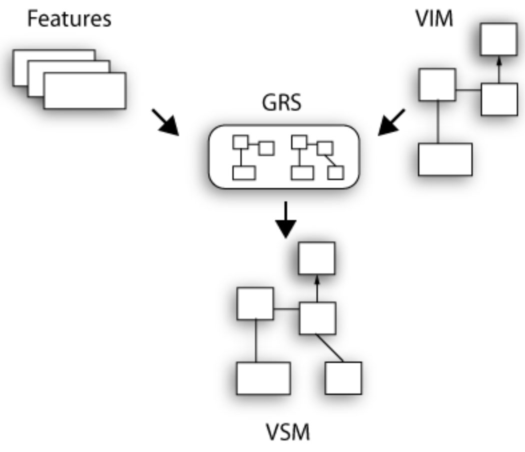


Figure 4: Parameterising the variant independent model (VIM) by features from a feature model to build a variant specific model (VSM)

5. CASE STUDY

We use the Fujaba Tool Suite [18] for building a small case study on using GRS for model weaving. Fujaba is a graph-based tool which uses UML for design and realisation of software projects. In the next section, we shortly introduce the concepts of Fujaba and especially its technique to define behavioural software parts with so called story diagrams. We demonstrate the feasibility of our approach by a small sample application.

5.1 The Fujaba Tool Suite

Fujaba, which means "From UML to Java and back again", was developed by the software engineering group of the University of Paderborn [2] and is supported by other German universities. It offers forward engineering as well as reengineering techniques, and code generation and was developed with the aim of helping non-professionals to develop software applications. The main concept, which shall help to reach this aim, is the concept of *Story-Driven Modelling* (SDM) [17]. With an extended type of UML collaboration diagrams combined with activity diagrams, the behaviour of methods can be defined by modelling complete method bodies, which are used for source code generation. With story diagrams, a UML-based graph rewrite language was supplied that should enable mainly students in an educational context to familiar with the paradigm of object-oriented software development more quickly and easily.

5.2 Sample application

To demonstrate our GRS-based approach, we created a short sample application, which implements the core concepts of communication systems. Our example focuses on the structural changes needed on the model level. Behavioural modelling can be done with appropriate models but is not covered by the example. The main idea is to realise several communication technologies which can be used by communication partners to exchange messages with each other. Figure 3 illustrates the feature model for this example, which contains only several communication features. As described above, the semantics of the feature model dictates at least one communication type but also allows the implementation of additional ones. According to the correspondent feature, the communication partners can exchange messages via email, SMS or regular mail. The latter can be understood as a kind of automatic submission of postcards in this software intensive context. A corresponding core solution model for a software system which covers these tasks, could look like the one depicted in Figure 5.

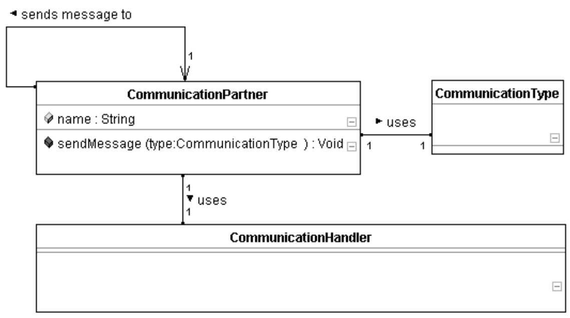


Figure 5: Core solution model before weaving

Besides the enumeration *CommunicationType*, the core solution model accomodates basically two classes, which shall be used for message exchange. The first one is the class *CommunicationPartner* which represents a communication party and contains information that is necessary to send messages according to a certain communication type. The message exchange via email for example needs information about the recipient's email address, while for SMS-based communication a mobile phone number has to be supplied. The only behavioural part in the class *CommunicationPartner* is covered by the *sendMessage()* method. This method triggers the message submission according to a certain communication type, which could be supplied e.g. by user interaction.

Due to the fact, that the solution model in Figure 5 illustrates just the core model of the system, the communication details for the class *CommunicationPartner* are not yet included. The same applies to the class *CommunicationHandler* and the enumeration *CommunicationType*, which are discussed below after showing the solution model as a result of a weaving transformation, as described above.

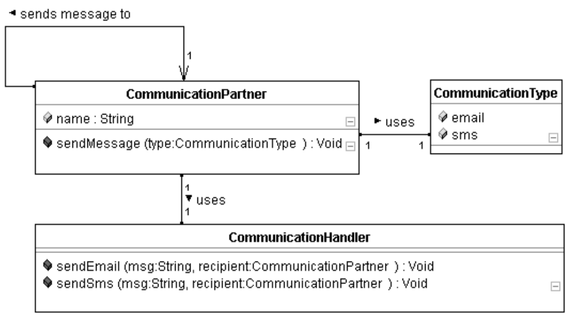

◂ sends message to
1
CommunicationPartner
name : String
sendMessage (type:CommunicationType) : Void
▸ uses
1
1
CommunicationType
email
sms
1
▾ uses
1
CommunicationHandler
sendEmail (msg:String, recipient:CommunicationPartner) : Void
sendSms (msg:String, recipient:CommunicationPartner) : Void

Figure 6: Solution model after weaving

Figure 6 depicts the solution model woven according to a certain variant model, assuming that the correspondent variant model for the shown solution model includes the communication types email and SMS but does not include regular mail sending. Hence, each aspect corresponding to the included features was introduced into the solution model. Now the necessary communication details are modelled in form of attributes in the class *CommunicationPartner* (e.g. phoneNumber). The class *CommunicationHandler* now contains methods to send messages via email or SMS. Additionaly, the enumeration *CommunicationType* contains constants that refer to all communication types that are possible and hence supported by the system.

The weaving applied to the core model was defined by correspondent graph rewrite rules, according to each included feature, as described in Section 4. In this case study we used the visual graph rewrite language of story diagrams, which was introduced in Section 5.1, to define the rewrite rules. Figure 7 shows the rule for including the email aspect.

As described in Section 4, the aspect-weaving behaviour has to be modelled on meta-model level, to be able to manipulate the underlying solution model which just instantiates the meta-model. Hence, the referenced classes in the story diagrams of our example refer to the UML meta-model. Based on the semantics of story diagrams, the upper part of Figure 7 defines the selection of the *UMLClass* with the name "CommunicationPartner", which is contained in the given solution model. This pattern defines the left-hand side of a graph rewrite rule. The connection between the objects *clazz* and *attribute* is stereotyped with the value *create*, which means, that a new attribute named "eMailAddress" has to be created for the selected class. This short story forms a rewrite rule that is sufficient to add the necessary details for email communication to the *CommunicationPartner* class. The story below implements the addition of the "sendMail" method to the class *CommunicationHandler*.

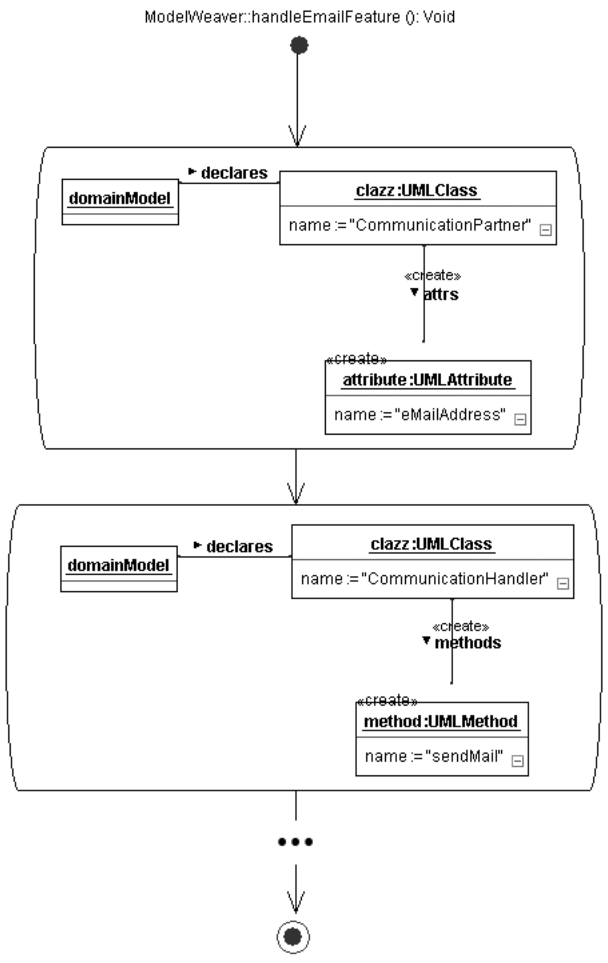


Figure 7: Partial story diagram for the email aspect

The story diagrams described above are now used to generate the actual aspect weaver, which transforms the solution model and thereby weaves the aspectual features. The weaving code is compiled on the fly and loaded into the Fujaba tool suite. Our solution also covers the integration of feature models by using a UML-based feature notation. Based on a specific variant model, the aspect weaver can be triggered via menu item and directly updates the solution model in the CASE tool. The whole solution is deployable as Fujaba plugin.

6. RELATED WORK

Many researchers applied aspect-oriented techniques to non-code artifacts, reaching from aspect-oriented requirements engineering (AORE) [32, 31] to aspect-oriented domain modelling (AODM) [20]. They invented methods to use and express AO concepts in the context of the UML [16] [9] and thereby raised the level of abstraction in AOSD.

It is well known that variabilities in product lines can be realised with aspect-oriented techniques. [19] analyzes different practices for implementing variabilities in PLE and also considers AOP as one possible solution. [26] uses feature oriented programming with collaborations in CaesarJ [27] while [22] introduces UML for Aspects, which supports aspectual collaborations as described in [21].

With AHEAD a more generalised approach is presented by Batory et al. [10] by introducing an equation-based model for refinement of code and non-code artifacts based on feature inclusion. The work on mapping features to models of Czarnecki et al. [13] is also related to our work, whereby the presented superimposition of variants requires all possible features - also conflicting ones - being modelled in the model.

In the domain of GRS, [7] describes the usage of GRS for aspect weaving and gives an overview of different GRS-based weaver classes. The research results presented in [8] are pointing towards integrating GRS tools in the standard software development tools using sublanguage projection. This refers to our work in terms of the integration of the GRS in the software tool as seen with Fujaba and our plugin. [3] uses graph-rewriting for applying transformation rules on domain-specific models defined in an UML-based meta-modelling language. [11] uses OPTIMIX [6] as GRS for the transformation of software designs in the context of the Model Driven Architecture, especially for transformation of platform independent models (PIM) to platform specific models (PSM). This approach has probably the closest relation to our work, because it also uses GRS for model transformation. It differs in regard to the amalgamation of feature models in PLE with solution models through generated GRS-based aspect weavers.

7. CONCLUSION AND FUTURE WORK

In this paper we showed that GRS can be used to weave aspectual features on model level. We explained AO-GRS and described the semantics of joint point, pointcut and advice to terms of graph rewriting. We pointed out, that our approach is not limited to models based on UML class diagrams, but can be used for UML state and sequence diagrams as well as for DSLs with an accessible meta-model. The usefulness of AO-GRS on the model-level was shown by a case study in the context of the Fujaba tool suite. We developed a plugin for Fujaba, which constructs model weavers based on story driven graph rewrite rules and applies them to solution models.

Our future work in this area will address the dependency between model-level aspects and implementation-level aspects in model weaving. Therefore, we plan to extend our approach to models describing dynamic behaviour of software systems and want to demonstrate the usefulness of our contribution in this area too.

Another possible research area is the integration of GRS in repositories, where both feature models and solution models reside together and are accessible to the GRS. This will reduce the effort in mapping the different model types, like feature models and solution models to graph models understandable by the GRS.

8. ACKNOWLEDGEMENTS

This work is supported by the feasiPLe project financed by the German Ministry of Education and Research (BMBF). We would like to thank Falk Hartmann, Steffen Zschaler, and the anonymous reviewers for their very valuable comments on previous versions of this paper.

9. REFERENCES

[1] AspectJ. http://www.eclipse.org/aspectj/.

[2] University of Paderborn, Germany. http://www.uni-paderborn.de/home/en/.

[3] A. Agrawal, G. Karsai, and A. Ledeczi. An end-to-end domain-driven software development framework. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 8–15, New York, NY, USA, 2003. ACM Press.

[4] U. Aßmann. Graph rewrite systems for program optimization. Technical Report RR-2955, INRIA Rocquencourt, 1996.

[5] U. Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*, pages 121–135, London, UK, 1996. Springer.

[6] U. Aßmann, A. Christoph, and J. Lövdahl. Optimix. http://optimix.sf.net.

[7] U. Aßmann and A. Ludwig. Aspect weaving with graph rewriting. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 24–36, London, UK, 2000. Springer.

[8] U. Aßmann and J. Lövdahl. Integrating graph rewriting and standard software tools. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 2003.

[9] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2004. IEEE Computer Society.

[10] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[11] A. Christoph. Graph rewrite systems for software design transformations. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 76–86, London, UK, 2003. Springer.

[12] K. Czarnecki. Overview of generative software development. In J.-P. Banâtre et al., editor, *UPP '04: Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 313–328, 2005.

[13] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In R. Glück and M. R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2005.

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

[15] Eclipse Foundation. Eclipse Modeling Framework Ecore meta-model. http://www.eclipse.org/emf/.

[16] T. Elrad, O. Aldawud, and A. Bader. Aspect-oriented modeling: Bridging the gap between implementation and design. In *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 189–201, London, UK, 2002. Springer.

[17] T. Fischer, J. Niere, L. Torunski, and A. Zuendorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, 1998.

[18] Fujaba Tool Suite Developer Team, University of Paderborn. Fujaba tool suite. http://www.fujaba.de/.

[19] C. Gacek and M. Anastasopoules. Implementing product line variabilities. In *SSR '01: Proceedings of the 2001 symposium on Software reusability*, pages 109–117, New York, NY, USA, 2001. ACM Press.

[20] J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. Gokhale, and B. Natarajan. An approach for supporting aspect-oriented domain modeling. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 151–168, New York, NY, USA, 2003. Springer.

[21] I. Groher, S. Bleicher, and C. Schwanninger. Model-driven development for pluggable collaborations. In O. Aldawud, T. Elrad, J. Gray, M. K. J. Kienzle, and D. Stein, editors, *7th International Workshop on Aspect-Oriented Modeling*, Oct. 2005.

[22] S. Herrmann. Composable designs with UFA. In *Workshop on Aspect-Oriented Modeling with UML at 1st Intl. Conference on Aspect Oriented Software Development*, 2002.

[23] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Software Engineering Institute, Carnegie Mellon University Pittsburgh, Pennsylvania 15213, 1990.

[24] K. Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *Software, IEEE*, 19:58– 65, Jul/Aug 2002.

[25] K. Lee, K. C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Lecture Notes in Computer Science*, volume Volume 2319, page Page 62, Januar 2002.

[26] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 127–136, New York, NY, USA, 2004. ACM Press.

[27] M. Mezini, K. Ostermann, and et al. CaesarJ. http://caesarj.org.

[28] Object Management Group. Model Driven Architecture. http://www.omg.org/mda/.

[29] Object Management Group. Unified Modeling Language 2.0. http://www.uml.org/.

[30] A. Radermacher. Support for design patterns through graph transformation tools. In *AGTIVE '99: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 111–126, London, UK, 2000. Springer.

[31] A. Rashid, A. Moreira, J. Araujo, P. Clements, E. Baniassad, and B. Tekinerdogan. Early aspects portal. http://www.early-aspects.net/.

[32] A. Rashid, A. Moreira, and B. Tekinerdogan. Special issue on early aspects: aspect-oriented requirements engineering and architecture design. *IEE Proceedings - Software*, 151(4):153–156, 2004.

[33] M. Völter and T. Stahl. *Model-Driven Software Development.* John Wiley & Sons, June 2006.

From Conditional Compilation to Aspects: A Case Study in Software Product Lines Migration

Vander Alves[*]
Informatics Center, UFPE
vander@acm.org

Alberto Costa Neto[*]
Informatics Center, UFPE
acn@cin.ufpe.br

Sérgio Soares[*]
Computing Systems Department, UPE
sergio@dsc.upe.br

Gustavo Santos
Meantime Mobile Creations
gustavo.santos@cesar.org.br

Fernando Calheiros
Meantime Mobile Creations
fernando.calheiros@cesar.org.br

Vilmar Nepomuceno
Meantime Mobile Creations
vsn@cesar.org.br

Davi Pires
Meantime Mobile Creations
davi.pires@cesar.org.br

Jorge Leal
Meantime Mobile Creations
jorge.leal@cesar.org.br

Paulo Borba[*]
Informatics Center, UFPE
phmb@cin.ufpe.br

ABSTRACT

Apart from adoption strategies, an existing Software Product Line (SPL) implemented using some variability mechanisms can be migrated to use another variability mechanism. In this paper, we present some migration strategies from one SPL implemented with conditional compilation to one using Aspect-Oriented Programming (AOP). The strategies present a variability pattern handled by the first mechanism and shows how it can be translated into a pattern using AOP constructs. We also show and discuss that some variability patterns cannot be migrated into AOP. The discussion centers around a commercial SPL in the mobile games domain.

1. INTRODUCTION

Adoption strategies for Software Product Lines (SPL) frequently involve bootstrapping existing products into a SPL (*extractive approach*) and extending an existing SPL to encompass another product (*reactive approach*), or their combination [8, 4]. The *proactive approach*, in which SPL design and implementation is accomplished for all products in the foreseeable horizon, may be less frequent in practice than the former approaches due to its incurred high upfront investment and risks. Extractive and reactive approaches can be enacted by the application of *program refactorings.*

Apart from adoption strategies, there may be a case when there is an existing SPL already implemented using some variability mechanisms and we would like to implement it using another variability mechanism. We refer to the process of accomplishing this as *migration strategy*, and reasons for accomplishing it include moving to a mechanism that better supports understandability, traceability, and further evolution of the SPL in the reactive scenario.

In this paper, we present some migration strategies from one SPL implemented with conditional compilation to one using Aspect-Oriented Programming (AOP) [7]. The strategies present a variability pattern handled by the first mechanism and shows how it can be translated into a pattern using AOP constructs. We also show and discuss that some variability patterns cannot be migrated into AOP. The discussion centers around a commercial SPL in the mobile games domain.

[*]Software Productivity Group http://www.cin.ufpe.br/spg.

Section 2 presents the case study used throughout the rest of the paper, motivating the need for migration strategies. Next, Section 3 presents migration strategies. Section 4 then addresses some mappings not possible in this migration strategy. Related work is considered in Section 5, and concluding remarks offered in Section 6.

2. THE CASE STUDY

The goal of the case study was to define and evaluate migration strategies for a mobile game SPL. The SPL considered was *Ronaldinho Total Control*[1], where player controls a soccer player in order to get the timing to keep the ball bouncing, make sequences of perfect hits to get bonuses, and get items to make his task easier and achieve the highest score. The game is running in a number of different devices. Devices differ in issues such as memory, screen sizes, additional keys, processing power, which ultimately constrains the features available in each of them. Thus, there are a number of versions of the game running in each device, where each version has slightly different features. The number of instances of this SPL is 16. Figure 1 illustrates the game screen of the game running in two of these different devices.

In Figure 1, the screen on the right is from a power end device, whereas the screen on the left is from a resource-constrained device. Apart from screen dimensions, we can also notice the existence of a bird and a cloud on the right. These are actually scenery objects that move in the background, called *croma* feature. This feature is optional in the SPL and is not present in the device on the left, since due to its constraint on bytecode size.

In terms of implementation, the variabilities involved have different granularity: some relate to the existence or not of particular proprietary drawing API, whereas others happen

[1]Access provided by our industrial partner

Figure 1: Game screen of the game in different devices.

within classes involving addition/removal of fields and code blocks inside methods.

The original variability mechanism of the SPL is conditional compilation, which is still largely used in the industry, especially in the mobile games domain. Nevertheless, this mechanism is not an appropriate substitute for proper programming language support. Also, such mechanism has poor legibility and leads to lower maintainability.

The variability of this domain shows considerable tangling and crosscutting. Therefore, it is worth investigating the usefulness of AOP mechanism in handling them. In this context, we propose a number of migration strategies discussed in the next section.

3. MIGRATION STRATEGIES

In this section, we present some migration strategies. They were adopted in our case study to migrate from a mobile game SPL implemented using conditional compilation to AspectJ constructs. Some kinds of variations could not be solved using the current version of AspectJ (1.5) and are presented later in Section 4. Each strategy is first described in the context of a concrete example; then it is generalized in a template form, so that it can support automation, which is reagarded as future work.

3.1 Super Class Variation

In the case study, there were variations in the super class of some classes. These variations occur, for example, when defining a `Canvas` class that is used to draw shapes and images on the screen. For Nokia devices, it is required that these classes extend the Nokia API class `com.nokia.mid.ui-.FullCanvas` instead of MIDP [12] class `javax.microedition-.lcdui.Canvas`. If the device supports MIDP 2.0 or is a Siemens mobile device, `Canvas` super classes are also different, called respectively `javax.microedition2.lcdui.game.-GameCanvas` and `com.siemens.mp.color_game.GameCanvas`. As a consequence, dealing with this variation requires changing an `import` declaration and the corresponding super class name in the `extends` clause. Using conditional compilation tags, it is possible to define a different import and extends declaration for each of those variations. The following piece of code shows how this variability mechanism is employed to address such variations for configurations corresponding to Nokia and MIDP devices.

```
//#ifdef nokia_device
//#  import com.nokia.mid.ui.FullCanvas;
//#else
//#  import javax.microedition.lcdui.Canvas;
//#endif
...
//#ifdef nokia_device
//#  public class MainCanvas extends FullCanvas {
//#else
//#  public class MainCanvas extends Canvas {
//#endif
...
```

Using AspectJ, such variability can be addressed by declaring an aspect for each possible super class alternative, corresponding to a different configuration. A `declare parents` clause with the required class name is defined in the aspect. Additionally, the corresponding import declaration is transferred to the aspect. The piece of code below shows the result of applying this strategy to the example above.

```
//core
public class MainCanvas {...}

//Nokia configuration
import com.nokia.mid.ui.FullCanvas;
public aspect NokiaCanvasAspect {
  declare parents: MainCanvas extends FullCanvas;
  ...
}

//MIDP configuration
import javax.microedition.lcdui.Canvas;
public aspect MIDPCanvasAspect {
   declare parents: MainCanvas extends Canvas;
   ...
}
```

The approach presented above only works because `FullCanvas` is a subclass of `Canvas`, which is a precondition of declare parents. The classes `GameCanvas` (MIDP 2.0 and Siemens) also respect this rule.

This strategy can be generalized by a pair of source and target templates specifying a transformation on code assets of the SPL. The source template is as follows:

```
//#ifdef TAG
//#  ts'
//#else
//#  ts''
//#endif

//#ifdef TAG
//#  public class C extends C' {
//#else
//#  public class C extends C'' {
//#endif
       fs
       ms
     }
```

Where `TAG` is a conditional compilation tag, whose selection in the SPL configuration binds the superclass of `C` to `C'`, including the corresponding import. When not selected in the SPL configuration, the superclass of `C` is bound to `C''`, also including its corresponding import. We denote the set of type declarations by `ts'` and `ts''`. Also, `fs` and `ms` denote field declarations and method declarations, respectively.

Code assets matching the source template are transformed according to the following target template, where aspect `A` binds the superclass of `C` to `C'`. The import required by `C'` is in `ts'` and is moved aspect `A`.

```
//core
public class C {
  fs
  ms
}

//configuration 1
ts'
public aspect A {
  declare parents: C extends C';
}

//configuration 2
ts''
public aspect B {
  declare parents: C extends C'';
}
```

3.2 Interface Implementation Variation

Another kind of variation in hierarchy addressed in the case study was to make a class implement a different interface. It usually happens due to the use of different APIs requiring the implementation of specific interfaces. This variability issue is similar to the one presented in the previous subsection and can be handled similarly in the migration strategy. The main difference is that it uses `declare implements` instead of `declare parents`.

3.3 If Condition Variation

A common variation in mobile devices is the number and type of keys in the keypad. Additionally, the values that represent key pressing events differ between mobile devices families. This latter variability is usually implemented through blocks of constant definitions with different values subject to conditional compilation. Other possible implementations include macro and configuration files.

When migrating to AspectJ, it is possible to introduce constants via inter-type declarations with the appropriated values. Additionally, there are variations in `if` conditions responsible for checking whether an specific key has been pressed and launch the code that treats the event. These variations usually required to add more `or-conditions` to treat the additional keys. The following code shows an example of this situation.

```
public class MainCanvas extends Canvas {
  protected void keyPressed(int keyCode) {...
    if (keyCode == LEFT_SOFT_KEY
//#ifdef device_keys_motorola
//#     ||  keyCode == -softKey
//#endif
    ) {
      // handle key event
    }
    ...
  }
}
```

The previous example shows that an additional `or-condition` can activate the code inside `if` command for Motorola mobile devices. With conditional compilation, using one or more `ifdef`'s addresses this variability issue.

We defined a migration strategy that involved 1) the extraction of `if-condition` to a new method defined in the class containing the base condition; 2) the use of an around advice in an aspect to enhance the base condition. The result is as follows:

```
public class MainCanvas extends Canvas {
  protected void keyPressed(int keyCode) {...
    if(compareEquals(keyCode, softkey)) {
       // handle key event
    }
  }
  private boolean compareEquals(int keyCode,
                                int softKey) {
    return keyCode == softKey;
  }...
}

//Motorola device configuration
public privileged aspect DeviceKeysMotorola {
  boolean around(int keyCode, int softKey) :
    execution(private boolean MainCanvas.compareEquals(..))
    && args(keyCode, softKey)
  {
    return keyCode == softKey || keyCode == -softKey;
  }
  ...
}
```

The source template of the migration strategy is shown next:

```
ts
public class C {
 fs
 ms
 T m(ps) {
   body
   if (cond
//#ifdef TAG
//#    op cond'
//#endif
    ) {
      body'
    }
    body''
  }
}
```

where `cond` represents the base condition and the variation is an additional expression `op cond'`. The expression `op` represents binary operators and `cond'`, any boolean expression. Also, `body`, `body'`, and `body''` denote blocks of statements in a method. The target template of this strategy is presented next:

```
ts
public class C
 fs
 ms
 T m(ps) {
   body
   if (getCond(ps')) {
     body'
   }
   body''
 }
 boolean getCond(ps') {
   return cond;
 }
}

//SPL configuration handling variability issue
public aspect A {...
  boolean around(ps') :
    execution(boolean C.getCond(..))
    && args(ps')
  {
    return cond || cond';
  }
}
```

It is important to notice that using an `around-advice` allows substituting or complementing the original condition specified in the `if` statement, by executing or not a `proceed` statement.

3.4 Feature Dependency

This section presents the strategy employed to migrate a feature depending on others features. In the case study, there is a feature called Arena, that allows posting game results to a public server for ranking purposes. This feature also presents results on the device screen. Since screen size is variable across devices, it would be necessary to develop an Arena feature to each appropriated screen size. Using conditional compilation, this feature implementation is spread in many classes and tangled with other functionalities.

In the following code, if the tag `feature_arena_enabled` is enabled during SPL instantiation, some common constants to paint the scroll bar are defined, but the constants `ARENA_SCROLL_HEIGHT` and `ARENA_SCROLL_POS_Y` have different values depending on the device's screen size.

```
public class MainScreen {
//#if feature_arena_enabled
   /** Constants to paint the scroll bar */
    //#if device_screen_128x128
    //#  public static final int ARENA_SCROLL_HEIGHT = 92;
    //#  public static final int ARENA_SCROLL_POS_Y = 17;
    //#elif device_screen_128x117
    //#  public static final int ARENA_SCROLL_HEIGHT = 81;
    //#  public static final int ARENA_SCROLL_POS_Y = 16;
    //#endif
//#endif
...
}
```

The strategy adopted to implement this feature dependency was to define an aspect called `ArenaAspect` to handle the core of the feature and, for each screen size variation inside Arena, define others aspects, `ArenaScreen128x128` and `ArenaScreen128x117`. Additionally, there is the following constraint on the SPL configuration knowledge: when the optional feature Arena is enabled, one of the aspects `ArenaScreenWxH` is automatically selected depending on the screen size of the device. The piece of code below shows the result of applying this strategy to the class `MainScreen` mentioned previously.

```
public class MainScreen {... }

public aspect ArenaAspect {
   /** Constants to paint the scroll bar */
}

public aspect ArenaScreen128x128 {
  public static final int
      MainScreen.ARENA_SCROLL_HEIGHT = 92;
  public static final int
      MainScreen.ARENA_SCROLL_POS_Y = 17;
}

public aspect ArenaScreen128x117
  public static final int
      MainScreen.ARENA_SCROLL_HEIGHT = 81;
  public static final int
      MainScreen.ARENA_SCROLL_POS_Y = 16;
}
```

The template generalizing this migration strategy is presented next. It is important to notice that `TAG_A` represents an optional feature and tags `TAG_B1` and `TAG_B2` represent features depending on `TAG_A`.

```
public class C {
  fs
  ms
 ...
//#if TAG_A
//#  fs'
//#  ms'
   //#if TAG_B1
   //#  fs''
   //#  ms''
   //#elif TAG_B2
   //#  fs'''
   //#  ms'''
   //#endif
//#endif
}
```

The target template of this strategy is presented next, where `C.fs'`, `C.fs''` and `C.fs'''` are the sets of fields introduced via inter-type declaration into class `C` by the aspects composed with `C`. The same pattern is used for methods, but they are named `C.ms'`, `C.ms''` and `C.ms'''` instead. Aspect `A` is included in the SPL instance iff feature `A` is selected; aspects `AB1` and `AB2` are present in the SPL instance iff their corresponding features are present and feature `A` is also selected.

```
public class C {
  fs
  ms
}
public aspect A {
  C.fs'
  C.ms'
}
public aspect AB1 {
  C.fs''
  C.ms''
}
public aspect AB2 {
  C.fs'''
  C.ms'''
}
```

3.5 Discussion

Some of the strategies presented previously could benefit from general OO techniques (e.g. using abstract methods and subclassing, patterns and so forth), but this would imply having a subclass for each possible device, thus leading to complex class hierarchies. Additionally, many more classes would be involved, thus incurring into a penalty in terms of bytecode size, a critical issue in the mobile application domain.

The strategies replace the scattered ifdefs by a number of aspects, which have to be managed. This is addressed by a configuration knowledge, relating device configurations to configurations involving sets of aspects and core classes. The AO advantage lies in the fact that the extracted variability can be used elsewhere without replicating code, whereas the ifdef variability can only be used in that context.

Although some variabilities addressed are very fine-grained, they are crosscutting, because they can be logically grouped together with other fine-grained variability affectting other join points, such that this unit–the aspect–implements a feature. More generally, we could further cluster crosscutting variability so that it can be more broad in a module-classes and aspects–implementing a given feature. This is regarded as future work.

Some strategies not shown involved handling variability in the definition and usage of constants. The usage of constants can certainly benefit from using final static variables, an appraoch we have used; however, variability in the definition constants themselves was addressed by a migration strategy to use inter-type declaration. On the other hand, mode-driven approach is not appropriate in this domain because device constraints such as memory imply constraints on game features, thus preventing the definition of a pure platform independent model.

4. OPEN ISSUES

In addition to the migration strategies already presented, there are some variations for which we could not define a migration strategy using AspectJ. In this section, we address those by showing how AspectJ's current implementation does not support them. In some cases, we provide alternative solution using other approaches; in others, we present candidate extensions to the AspectJ language.

4.1 Import Variation

In the performed case study, there are variations between device families that use different APIs. These APIs define types with the same name and the same interfaces to facilitate the porting task. However, those types are defined in different name spaces, since each API has its own package name. For instance, the following piece of code depicts an example of such variation. The code originally written with conditional compilation tags imports a `Sprite` type from `javax.microedition.lcdui.game` package or from `com.meantime.j2me.util.game` depending on the MIDP version it uses. The latter is used when generating a release to device families that use MIDP 2.0, and the former otherwise.

```
//#ifdef game_sprite_api_midp2
//#   import javax.microedition.lcdui.game.Sprite;
//#elif
//#   import com.meantime.j2me.util.game.Sprite;
//#endif
...
```

Since the AspectJ language in its current version (1.5) does not handle variability at the import clauses granularity, there is not a solution to migrate this conditional compilation code to AspectJ code. One alternative for such kind of variations would be extending AspectJ with inter-type declarations that insert an import clause in a type. Another possibility would be using a transformation system [3] that uses generative techniques allowing to control such kind of elements in the source code.

This concrete example can be generalized to variations that demand different imports clauses, regardless of the types' name. The form of such problem is presented in the following piece of code.

```
...
//#if TAG_1
//#   import I_1;
//#elif TAG2
//#   import I_2;
...
//#elif TAG_n
      import I_n;
//#endif
...
```

where `TAG_1`, `TAG_2`, and `TAG_n` are conditional compilation tags that define variation code and `I_1`, `I_2`, and `I_n` are the imports expressions.

4.2 Superclass Constructor Call

Another example of conditional compilation code that could not be migrated to AspectJ is a call to a superclass constructor. In this example, two variants demands calling the superclass constructor with the parameter `false` if the device uses MIDP 2.0 or if it is a Siemens device; otherwise, no explicit super call is needed, thus implying an implicit to the empty superclass constructor.

```
...
  public MainCanvas() {
//#if device_graphics_canvas_midp2 ||
//#   device_graphics_canvas_siemens
//#      super(false);
//#endif
         ...
  }
...
```

AspectJ does not support such migration since an advice cannot call a constructor using neither `super` nor `this`. In fact, it is possible to write a code that prevents the superclass constructor to execute, but not a code that executes one constructor instead of another.

A possible solution would be extending AspectJ to allow writing an advice that executes first in a constructor call and can call the superclass constructor or another constructor in the same class, or using the transformation system mentioned before to add such constructor call.

This issue can be generalized to any variation that demands a different superclass constructor call:

```
...
  CONSTRUCTOR(PARS) {
//#if TAG
//#   super(ARGS);
//#endif
      ...
  }
```

or a change in the inline calls of class constructors.

```
...
  CONSTRUCTOR(PARS) {
//#if TAG
//#   this(ARGS);
//#endif
      ...
  }
```

where PARS is the constructor parameter list, which can be empty, and ARGS is the argument list, possible empty, of the class or superclass constructor call.

4.3 Adding an else-if Block

Another migration issue occurs when a variation demands the insertion of new else-if blocks in a conditional statement. This case is common with feature variations that add new screens to the game. The code that paints the current screen must check the type of the current screen in a long if-else-if structure; therefore, new screen type checks are added as else-if's to the end of this structure.

```
     ...
     if (this.screenMode ==
           Resources.MAIN_SCREEN_MODE_SPLASH) {
       //code
     } else if (this.screenMode ==
           Resources.MAIN_SCREEN_MODE_LOGO) {
       //code
     }
//#ifdef feature_arena_enabled
//#  else if (this.screenMode ==
//#        Resources.MAIN_SCREEN_MODE_ARENA_WELCOME) {
//#    //code
//#  } else if (this.screenMode ==
//#        Resources.MAIN_SCREEN_MODE_ARENA_LOGIN) {
//#    //code
//#  }
//#endif
```

There is no construction in AspectJ that deals with conditional statements or any similar that would address this issue. The alternative would be again using the transformation system to generate the code to be added. An AspectJ extension that intercepts conditional statements does not seam very useful, since the conditional statements are not named, which leads to ambiguity when a method has more than one conditional statement.

This issue can be generalized by the following form:

```
    ...
    if(EXP_1) {
      // code
    } else if (EXP_2) {
      // code
    }
    ...
//#ifdef TAG
    else if(EXP_n) {
      // variation
    }
//#endif
```

where EXP_1, EXP_2, and EXP_n are boolean expressions.

5. RELATED WORK

We have previously explored SPL adoption strategies at the implementation level [2] and at the feature model level [1]. In this work, instead of adoption strategies, we address migration of variability mechanism in an existing SPL.

Lopez-Herrejon et al [10] have evaluated the use of different variability mechanisms in providing support for modularization of features. Differently, our work focuses on the migration of one technique to another by providing strategies specified by means of templates. Our work could benefit from theirs by considering migration strategies to other target variability mechanisms in cases where AspectJ does not have appropriate constructs.

Another work [6] explores the application of refactoring to SPL Architectures. They present metrics for diagnosing structural problems in a SPL Architecture, and introduce a set of architectural refactorings that can be used to resolve those problems. These metrics could be useful for detecting bad smells and guiding the application of our migration strategies.

Monteiro et al [11], Laddad [9], and Cole et al [5] discuss refactoring from Java to AspectJ programs. Although these works are not directly related to SPLs, we can use several OO to AO refactorings to extract variations of a mobile application in a extractive approach to define a SPL. In the particular example addressed in this paper, we worked on an SPL that was already implemented using conditional compilation and extracted the variations to use aspects, following an approach which was neither extractive, nor proactive nor reactive.

6. CONCLUSION AND FUTURE WORK

We have presented migration strategies from one SPL implemented with conditional compilation to one using AOP. The strategies present a variability pattern handled by the first mechanism and shows how it can be translated into a pattern using AOP constructs. We also show and discuss that some variability patterns cannot be migrated into AOP. The discussion centers around a commercial SPL in the mobile games domain.

As future work, we intend to explore other target variability mechanisms and also to enhance AspectJ to overcome the mappings not currently possible in our migration strategy. We also plan provide automation support for the templates and to asses them in different product lines. Finally, we will explore clustering the variability into a broader context.

7. ACKNOWLEDGMENTS

We gratefully acknowledge our industrial partner, Meantime Mobile Creations, for granting us access to the game in this case study. This research was partially sponsored by CNPq (grants 481575/2004-9, 141247/2003-7, 552068/2002-0) and MCT/FINEP/CT-INFO (grant 01/2005 0105089400).

8. REFERENCES

[1] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring product lines. In *Proceedings of the 5th ACM International Conference on Generative Programming and Component Engineering (GPCE'06)*. ACM Press, Oct 2006. To appear.

[2] Vander Alves, Pedro Matos Jr., Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and evolving mobile games product lines. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, volume 3714 of *Lecture Notes in Computer Science*, pages 70–81. Springer-Verlag, Sep 2005.

[3] Fernando Castor, Kellen Oliveira, Adeline Souza, Gustavo Santos, and Paulo Borba. JaTS: A Java transformation system. In *XV Brazilian Symposium on Software Engineering*, pages 374–379, Rio de Janeiro, Brazil, October 2001.

[4] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[5] Leonardo Cole and Paulo Borba. Deriving refactorings for aspectj. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development - AOSD'05*. ACM press, March 2005.

[6] Matt Critchlow, Kevin Dodd, Jessica Chou, and André van der Hoek. Refactoring product line architectures. In *IWR: Achievements, Challenges, and Effects*, pages 23–26, 2003.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect–Oriented Programming. In *European Conference on Object–Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, 1997.

[8] Charles Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th Workshop on Software Product-Family Engineering*, pages 282–293, 2001.

[9] Ramnivas Laddad. Aspect oriented refactoring series. In *TheServerSide.com*, December 2003.

[10] Roberto E. Lopez-Herrejon, Don S. Batory, and William R. Cook. Evaluating support for features in advanced modularization technologies. In *ECOOP*, pages 169–194, 2005.

[11] M. P. Monteiro and J. M. Fernandes. Object-to-aspect refactorings for feature extraction. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development - AOSD'04*. ACM press, March 2004.

[12] Java Community Process. *Mobile Information Device Profile 2.0*. http://jcp.org/aboutJava/communityprocess/-final/jsr118/index.html, 2004.