The Role of Features and Aspects in Software Development

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik

der Otto-von-Guericke-Universität Magdeburg

von: Diplom-Informatiker Sven Apel

geboren am 21.04.1977 in Osterburg

Gutachter:

Prof. Dr. Gunter Saake

Prof. Don Batory, Ph.D.

Prof. Christian Lengauer, Ph.D.

Promotionskolloquium: Magdeburg, Germany, 21.03.2007

Apel, Sven:
The Role of Features and Aspects in Software Development
Dissertation, Otto-von-Guericke-Universität
Magdeburg, Germany, 2006.

Abstract

In the 60s and 70s the *software engineering* offensive emerged from long-standing problems in software development, which are captured by the term *software crisis*. Though there has been significant progress since then, the current situation is far from satisfactory. According to the recent report of the Standish Group, still only 34% of all software projects succeed.

Since the early days, two fundamental principles drive software engineering research to cope with the software crisis: *separation of concerns* and *modularity*. Building software according to these principles is supposed to improve its understandability, maintainability, reusability, and customizability. But it turned out that providing adequate concepts, methods, formalisms, and tools is difficult.

This dissertation aspires to contribute to this field. Specifically, we target the two novel programming paradigms *feature-oriented programming (FOP)* and *aspect-oriented programming (AOP)* that have been discussed intensively in the literature. Both paradigms focus on a specific class of design and implementation problems, which are called *crosscutting concerns*. A crosscutting concern is a single design decision or issue whose implementation typically is scattered throughout the modules of a software system. Hence, crosscutting concerns contradict and violate the principles of separation of concerns and modularity.

Though FOP and AOP provide method-level, language-level, and tool-supported means to deal with crosscutting concerns, they do so in different ways. In this dissertation we demonstrate that FOP and AOP are not competing approaches but that their combination can overcome their individual limitations. We underpin this insight by a classification of crosscutting concerns and an evaluation of FOP and AOP with respect to different classes of crosscutting concerns. The result is a set of programming guidelines in form of a catalog that contrasts the strengths and weaknesses of FOP and AOP.

In order to profit from their individual strengths, we propose the symbiosis of FOP and AOP. To this end, we present *aspectual feature modules (AFMs)* that realize the symbiosis by the integration of concepts, design rationales, languages constructs, and tools for FOP and AOP. An evaluation and comparison with traditional FOP and AOP corroborates that AFMs largely profit from either's strengths.

Furthermore, we emphasize that current AOP languages are not suited to be combined with the stepwise development style of FOP. Consequently, we introduce the notion of *aspect refinement (AR)* that unifies AOP and stepwise software development and that is underpinned by a set of accompanying language constructs and tools.

A non-trivial case study demonstrates the practical applicability of AFMs and AR to a medium-sized software project. This study reveals a further fundamental issue: Given the programming guidelines, how are mechanisms related to AOP and FOP used in contemporary programs? The background is that a specific class of crosscutting concerns, called *collaborations*, is connected naturally with FOP. Due to the missing support in main stream programming languages today, AOP has frequently been used to implement collaborations.

However, with the advent of languages that support collaborations and the classification and evaluation contributed by this dissertation, we ask: What fraction of aspect-oriented code implements collaborations? What fraction implements crosscutting concerns beyond collaborations? A quantitative analysis of 8 AspectJ programs of different size reveals that on average 98% of the code base is associated with collaborations and only 2% exploits the advanced capabilities of AOP. Furthermore, we observed that the impact of AOP decreases as the program size increases.

Finally, the dissertation discusses why this (dis)proportion of code related to AOP and FOP is not surprising and whether and how the impact of AOP can be increased.

Zusammenfassung

Der Begriff *Softwaretechnik* und die damit verbundene Offensive erwuchs in den 60ern und 70ern aus den anhaltenden Problemen bei der Entwicklung von Software, welche unter dem Begriff *Softwarekrise* zusammengefasst werden. Obwohl sich seitdem einiges bewegt hat, ist die derzeitige Situation in der Softwareentwicklung alles andere als zufrieden stellend. Laut dem aktuellen Bericht der Standish Group werden nur 34% aller Softwareprojekte erfolgreich zum Abschluss gebracht.

Seit dem werden zwei Prinzipien eng mit der Überwindung der Softwarekrise in Verbindung gebracht: *Trennung von Belangen (separation of concerns)* und *Modularität (modularity)*. Finden diese Prinzipien in der Entwicklung von Software Beachtung, lässt sich die Verständlichkeit, Wartbarkeit, Wiederverwendbarkeit und Maßschneiderbarkeit von Software signifikant verbessern. Allerdings stellte sich schnell heraus, dass es weit komplizierter ist, adäquate Konzepte, Methoden, Formalismen und Werkzeuge zu entwickeln, als zunächst angenommen.

Diese Dissertation hat zum Ziel, zu diesem Bereich der Forschung beizutragen. Im Speziellen beschäftigt sich die Arbeit mit zwei derzeitig diskutierten Programmierparadigmen, der *Feature-orientierten Programmierung (FOP)* und der *Aspekt-orientierten Programmierung (AOP)*. Beide Paradigmen konzentrieren sich auf eine bestimmte Klasse von Entwurfs- und Implementierungsproblemen, die so genannten *querschneidenden Belange (crosscutting concerns)*. Ein querschneidender Belang entspricht einer einzelnen Entwurfs- oder Implementierungsentscheidung bzw. einer Fragestellung oder eines Ansinnens, dessen Implementierung typischerweise über weite Teile eines Softwaresystems verstreut ist. Aus diesem Grund widersprechen querschneidene Belange den Prinzipien der Trennung von Belangen und der Modularität.

FOP und AOP stellen beide methodische und programmiersprachliche Mittel und Werkzeuge bereit, gehen das Problem der querschneidenden Belange aber auf sehr unterschiedliche Weise an. In dieser Dissertation wird jedoch festgestellt, dass FOP und AOP keine konkurrierenden Ansätze sind, sondern dass ihre Kombination die individuellen Schwächen überwinden kann. Diese Einsicht wird untermauert durch eine Klassifikation von querschneidenden Belangen und eine Evaluierung von FOP und AOP hinsichtlich der verschiedenen Klassen querschneidender Belange. Ergebnis ist ein Satz von Pro-

grammierrichtlinien in Form eines Katalogs, der die Stärken und Schwächen von FOP und AOP gegenüberstellt.

Um von den individuellen Stärken beider Paradigmen zu profitieren, wird in dieser Dissertation die Symbiose von FOP und AOP vorgeschlagen. Insbesondere präsentieren wir den Ansatz der *Aspekt-basierten Featuremodule (aspectual feature modules – AFMs)*, welche die Symbiose umsetzen, indem sie die Entwurfsphilosophien, Sprachmechanismen und Werkzeuge von FOP und AOP kombinieren. Eine Evaluierung und eine Gegenüberstellung mit traditioneller FOP und AOP demonstrieren die Überlegenheit von AFMs.

Des Weiteren wird in der Dissertation herausgestellt, dass derzeitige AOP-Sprachen nicht uneingeschränkt geeignet sind, in die schrittweise Entwurfsphilosophie von FOP integriert zu werden. Konsequenterweise wird der Ansatz der *Aspektverfeinerung (aspect refinement – AR)* vorgestellt, welcher AOP und schrittweise Softwareentwicklung à la FOP vereinheitlicht. Weiterhin werden entsprechende Sprachkonstrukte und Werkzeuge zur Verfügung gestellt.

Mittels einer nicht-trivialen Fallstudie wird die praktische Anwendbarkeit von AFMs und AR auf ein mittelgroßes Softwareprojekt demonstriert. Die Studie wirft weiterhin eine fundamentale Frage auf: Wie werden Mechanismen von FOP und AOP heutzutage verwendet. Hintergrund ist, dass eine spezielle Klasse von querschneidenden Belangen eng mit FOP verknüpft ist, die so genannten *Kollaborationen (collaborations)*. Durch die fehlende Unterstützung von Kollaborationen in aktuellen Programmiersprachen wird dafür heute oft AOP benutzt.

Durch das Aufkommen von Programmiersprachen, die Kollaborationen explizit unterstützen, sowie durch die in dieser Dissertation präsentierte Klassifikation und Evaluierung, stellen sich jedoch folgende Fragen: Welcher Anteil von Aspektcode implementiert Kollaborationen? Welcher Anteil implementiert querschneidene Belange, die darüber hinaus AOP benötigen? Eine quantitative Analyse von 8 AspectJ-Programmen unterschiedlicher Größe ergibt, dass durchschnittlich 98% der Codebasis der analysierten Programme mit Kollaborationen verknüpft sind und nur 2% die erweiterten Mittel von AOP jenseits von Kollaborationen ausnutzen. Weiterhin wird beobachtet, dass mit steigender Programmgröße der Einfluss von AOP sinkt.

In der Dissertation wird die Frage beantwortet, warum dieses (Miss)Verhältnis zwischen AOP und FOP-Code besteht, und warum dies nicht überrascht. Weiterhin wird diskutiert, ob und wie der positive Einfluss von AOP gesteigert werden kann.

Acknowledgements

Pursuing a Ph.D. is an endeavor that is not only steeped in sudden inspiration and pure scientific beauty, but is often a path with many obstacles, difficulties, and throwbacks. However, I was able to make my way because many people accompanied and supported me. First, I want to thank Susi my partner and wife-to-be for her encouragement, love, and "mental support" in good and bad times. I am also grateful to my parents who supported me, my dreams, ideas, and plans from the very beginning of my life. Furthermore, I want to acknowledge my brother, my grandparents, my grandaunt, my mother-in-law-to-be, and Ayla (a.k.a. "The Queen of Saba") who all helped and believed in me, in one or the other way.

In my life I had many teachers, mentors, and professors – too many to mention here. Certainly, my advisers played an important role in my dissertation. It is worth noting that I had not just one adviser, but two advisers who supported me in different ways.

First, Don Batory helped me understand the "big picture" of my work, and helped me refine my style in scientific thinking, working, and writing. In my time at the University of Texas at Austin we had many long and emotional discussions about fundamental issues of my research and philosophical issues of science. It was an honor and a true pleasure working with him. His support, patience, encouragement, and advice has gone further than I would have imagined and expected.

Second, I want to thank my adviser Gunter Saake who called me in his group at the University of Magdeburg and gave me the opportunity for doing my Ph.D. in the first place. In the face of numerous organizational, operational, and financial problems, he always came up with an unconventional and practical solution that ensured my freedom of research and scientific work. He always believed in my abilities and supported me without asking. I learned many invaluable truths from him about the amenities and dangers of the world of research, science, and academics.

As a further mentor I want to thank Christian Lengauer. I first met him at the University of Texas at Austin. We quickly realized that despite our largely different views of the world of software engineering and programming, we were able to work in harmony. I profited always from our controversial discussions that emerged from our different

perspectives and backgrounds. This raised my awareness of the importance of formal foundations and precise definitions.

During the years of my Ph.D. studies I worked and discussed with many other researchers that contributed to the evolution of my thinking and understanding of many problems in computer science. The most influential persons were Erik Buchmann, Klemens Böhm, Thomas Leich, Roberto Lopez-Herrejon, Olaf Spinczyk, DeLesley Hutchins, Klaus Ostermann, Sahil Thaker, Walter Cazzola, and Jia Liu.

A special thank-you goes to the members of the Metop research institute that supported me financially and organizationally over long times. Particularly, I want to mention Thomas Leich and Marco Plack that granted support in many situations.

Furthermore, I want to thank several students and colleagues that were involved in many activities related to this dissertation: Christian Kästner, Marko Rosenmüller, Martin Kuhlemann, Helge Sichting, Holger Steinhaus, Laura Marnitz, and Karl-Heinz Deutinger.

Finally, I want to thank all members of the Database Group at the University of Magdeburg and the Product-Line Architecture Research Group at the University of Texas at Austin, as well as all my relatives and friends.

Contents

List of Figures

List of Tables

Abbreviations

AFM	Aspectual Feature Module
AHEAD	Algebraic Hierarchical Equations for Application Design
AOP	Aspect-Oriented Programming
AOR	Aspect-Oriented Refactoring
AR	Aspect Refinement
BAC	Basic and Advanced Dynamic Crosscuts
CIA	Classes, Interfaces, and Aspects
CORBA	Common Object Request Broker Architecture
FODA	Feature-Oriented Domain Analysis
FOP	Feature-Oriented Programming
FOR	Feature-Oriented Refactoring
HHC	Heterogeneous and Homogeneous Crosscuts
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
ITD	Inter-Type Declaration
LOC	Lines of Code
MDSoC	Multi-Dimensional Separation of Concerns
NOO	Number of Occurrences
OOP	Object-Oriented Programming
P2P-PL	Peer-to-Peer Product Line
SEI	Software Engineering Institute
SPL	Software Product Line
SWD	Stepwise Development
SWR	Stepwise Refinement
XML	Extensible Markup Language

CHAPTER 1

Introduction

1.1 Overview

software engineering and software crisis

The term '*software engineering*' was introduced in the NATO Working Conference on Software Engineering in 1968 [NR69]. Though there are alternative definitions we use the following: *software engineering is the analysis, design, implementation, documentation, customization, deployment, and maintenance of software by combining and applying technologies and practices from several fields, e.g., computer science, project management, engineering.* The software engineering offensive was started to cope with a whole class of phenomena observed in software development that were summarized by the term '*software crisis*'. The software crisis became manifest in projects running over-time, projects running over-budget, low-quality software, software that did not meet its requirements, projects that were unmanageable, and code that was difficult to maintain.

causes for the software crisis

Edsger Dijkstra, a pioneer of software engineering, explained the major cause for the software crisis as follows [Dij72]:

> *...machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*

progress and disillusion

Since the 60s, tremendous progress has been made in dealing with the software crisis. It became possible to construct increasingly complex software systems. However, the progress in developing concepts, methods, and tools for software engineering did not keep track with the enormous boost of the complexity and the sheer size of contemporary software systems. That is, the aspiration to establish software development as

an engineering discipline is, to a significant extent, still an aspiration [FGG+06]. The current software science and technology base is inadequate to meet current and future needs in software construction [Jac06, Boe06].

the Standish Group reports

According to the Standish Group in 1995, only about 16% of software projects were successful, 53% were fraught with problems (cost or budget overruns, content deficiencies), and 31% were cancelled; the average software project ran 222% late, 189% over budget and delivered only 61% of the specified functions [Gro95]. According to the Standish Group's most recent report, only 34% of all software projects were deemed to be successful [Gro03]. Evidence suggests that despite the improvement from 1995 to 2003 the current situation in software development is far from adequate [FGG+06, Jac06, Boe06, Gla05, Gla06].

separation of concerns and modularity

Fundamental principles that drive the research on software engineering since the early days are *separation of concerns* and *modularity*, which are highly related to each other. Building software according to these principles makes it more manageable and understandable and consequently software reuse, evolution, and maintenance is improved.

Separation of concerns means to break down a software into pieces [Dij82, Dij76, Par76, Par79]. These pieces are the *concerns* of a software system, in which a concern is a semantically coherent issue of a problem domain that is of interest. A concern may be a requirement such as 'realtime operation', a program feature such as 'RSA encryption', a data structure such as a B-tree, or even a tiny issue like implementing a length counter as long integer or as short integer. Concerns are the primary criteria for decomposing software into smaller, more manageable, and comprehensible parts, which is embodied by the principle of separation of concerns.

However, the definition of separation of concerns does not provide guidance on how to identify and arrange concerns. *Cohesion* proved to be an appropriate criterion. Cohesion is the grade of functional relatedness of the pieces of code that implement a concern [YC79]. High cohesion is preferable because it is associated with several desirable properties of software, e.g., robustness, reliability, reusability, and understandability. Structuring software on the basis of this criterion enables the software developer to concentrate on the issues regarding one concern in isolation, thus minimizing the distraction by implementation details of other concerns. Parnas describes this approach as *design for change* [Par79]: a programmer structures software such that the concern implementations encapsulate code that is likely to change. Following this approach, separation of concerns enables the change of a concern's implementation without affecting or depending on other concerns.

Modularity is the principle to structure software into modules or, expressed more quantitatively, it measures the extent to which modules are used in a software system.

The idea of modules emerged from several tracks of research, in particular, *modular programming* [Con68], *program specification* [Par72a, Par72b], *structured programming* [DDH72, Dij76], and *structured design* [SMC74, YC79]. Though there are various definitions, it has been agreed that a module must be a part of a larger system and inter-operate with other modules. Modules are self-contained, cohesive building blocks of software. A module is a device to implement a concern and modularity is a consequence of separation of concerns.

A module provides and communicates via an *interface* to hide specific details of the concern it implements (*information hiding*) [Par72b]. Interfaces decouple concern implementations from each other and minimize concern interdependencies. Modules with interfaces provide an enabling mechanism for separation of concerns and design for change.

challenges

The history of software engineering and programming language research is to a significant extent the history of supporting and improving separation of concerns and modularity. The challenge for the research community and the industry is to provide the right languages, abstractions, models, methods, and tools to assist software developers in building well-structured and modular software. This would be a major step to overcome the software crisis. Unfortunately, it turned out that this is a difficult task.

aim of the dissertation

This dissertation aspires to make a contribution to this field, i.e., to provide conceptual, methodological, practical, and tool-related means to improve the separation of concerns and modularity in software. Specifically, this dissertation focuses on two novel programming and software development paradigms, *feature-oriented programming (FOP)* [Pre97, BSR04] and *aspect-oriented programming (AOP)* [KLM+97, EFB01].

crosscutting concerns

Both, FOP and AOP target a specific class of design and implementation problems, which are called *crosscutting concerns* [KLM+97]. A crosscutting concern is a single design or implementation decision or issue whose implementation typically must be scattered throughout the modules of a software system, that results in inter-mingled code, and that leads to code replication. Crosscutting concerns are special as they challenge traditional programming and development paradigms such as *object-oriented programming (OOP)*. It has been observed that crosscutting concerns lead to inherently suboptimally structured code that decreases understandability and manageability of software [KLM+97, EFB01, TOHSMS99].

tyranny of the dominant decomposition

The problem of crosscutting is not a matter of a good or bad programming style or software design. It emerges directly from the missing support of traditional programming paradigms (e.g., OOP) to decompose software in multiple ways (along multiple dimensions), which is called the *tyranny of the dominant decomposition* [TOHSMS99]. That is, a program can be modularized in only one way at a time (along one dimension), and the many kinds of concerns that do not align with that modularization end up in

scattered, *tangled*, and *replicated code*. FOP and AOP address this issue explicitly and provide mechanisms for decomposing software along more than one dimension.

Although both FOP and AOP aim at modularizing crosscutting concerns, they approach this problem from different sides. While FOP deals with the automated synthesis of software out of features, AOP provides meta-level[1] language constructs that enable to reason about and manipulate base programs. In both FOP and AOP a programmer defines the points in a program to be extended (a.k.a. *join points*) and a set of actions, extensions, or transformations to be performed at these points.

AOP and FOP can profit from each other

Though it seems that FOP and AOP are competing approaches, in this dissertation we observe that FOP and AOP are complementary techniques. They decompose and structure software in different ways, along different dimensions, which leads to different program designs. We demonstrate how the combination of FOP and AOP can overcome their individual limitations. The different strengths and weaknesses revealed and systematized in this dissertation call for a symbiosis of both programming paradigms in order to profit from their advantages and to minimize their shortcomings.

programming guidelines

Given the numerous, individual strengths and weaknesses of FOP and AOP, we need guidelines to assist programmers in choosing the right technique for the right problem. The entire dissertation is steeped in these guidelines and can be understood as a historical overview of the author's investigations in this problem field: the programming guidelines have been derived from the evaluation of FOP and AOP and drive the proposal of the symbiosis of both; they have been evaluated in a non-trivial case study, and help to identify the current practice of using mechanisms of FOP and AOP.

In a nutshell, the guidelines for using FOP and AOP based on their strengths and weaknesses are the essence for comparing, combining, and unifying FOP and AOP. They guide the way to a better understanding of crosscutting concerns and of the corresponding implementation mechanisms, which, taken by itself, is a contribution to the debate about modularity and separation of concerns.

1.2 Contribution

1. We evaluate FOP and AOP with respect to their performance in facilitating separation and modularization of crosscutting concerns, as well as related evaluation criteria. This evaluation is preceded by a systematic classification of crosscutting concerns on the basis of their structural properties, which enables to systematize

[1] Steimann shows that AOP languages are essentially second-order languages. The processing of an aspect requires reasoning about and involves manipulation of a program, i.e., AOP is de facto a meta-programming technique [Ste05].

the evaluation process. The result is a set of programming guidelines that emphasizes the individual strengths and weaknesses of FOP and AOP.

2. We propose the symbiosis of FOP and AOP. We discuss its design space, present a concrete realization at the implementation level, and contribute several tools to assist programmers in combing FOP and AOP mechanisms.

3. Given the combination of FOP and AOP, we present a unification of AOP and the stepwise development methodology of FOP. This unification enables the uniform treatment of all implementation artifacts of a program feature (i.e., classes and aspects). This follows directly from the *principle of uniformity* that states that program features consist of various types of software artifacts and *all* artifacts can be subject of subsequent refinement [BSR04].

4. We demonstrate the practical applicability of our proposal by applying the core language constructs and tools to a medium-sized case study. This provides first insights into how FOP and AOP techniques would be combined in a non-trivial setting.

5. Finally, we present our investigations in how AOP and FOP mechanisms are used in third-party software projects. Background is that our programming guidelines devise in which situations AOP mechanisms outperform FOP mechanisms, and vice versa. By defining a set of code metrics, appropriate tool support, and an analysis of a set of third-party programs, we shed light on the questions: What is the current practice of using AOP and FOP? And to what extent related design and implementation problems occur?

1.3 Outline

Chapter 2 lays the foundations for understanding the central ideas of this dissertation. It limits its focus on essential concepts related to separation of concerns, modularity, FOP, AOP, and their connection to software engineering. Consciously, we avoid getting into much detail; we do not give a comprehensive or historical overview of related programming and software development approaches.

Chapter 3 introduces a classification framework for crosscutting concerns. This classification forms a systematic basis for the evaluation and comparison of FOP and AOP; it is essential to infer programming guidelines for choosing the right implementation technique for the right class of crosscutting concerns.

Chapter 4 presents the evaluation of FOP and AOP. For this purpose, we define a set of evaluation criteria that is applied in a comparison of FOP and AOP. The result

is a catalog that contrasts the strengths and weaknesses of FOP and AOP, which can be understood as a set of programming guidelines.

Chapter 5 elaborates on the symbiosis of FOP and AOP. After a brief discussion of the design space, the chapter introduces the notion of an *aspectual feature module (AFM)* that realizes the symbiosis. AFMs are evaluated using our criteria and compared to traditional FOP and AOP. Finally, we give an overview of several tools that have been developed in this dissertation and discuss related approaches.

Chapter 6 introduces the notion of *aspect refinement (AR)*, which unifies aspects and the stepwise development methodology of FOP. After a discussion we point to a tool developed in this dissertation and discuss related work.

Chapter 7 reviews the results of the application of AFMs and AR to a product line for overlay networks. We examine the collected data and discuss open issues and related studies.

Chapter 8 reflects on the experiences gained in the case study and extracts a problem statement. We define a set of code metrics and provide tool support for program analysis. We discuss the results of applying our metrics to 8 small-sized to large-sized AspectJ programs.

Chapter 9 summarizes the dissertation, puts the results into perspective, and lists suggestions for further work.

CHAPTER 2

Design and Implementation Techniques for Separation of Concerns and Modularity

This chapter lays the foundations for understanding the central ideas of this dissertation. It is not intended as a historical overview or as a comprehensive survey on design and implementation techniques for separation of concerns and modularity.

2.1 Separation of Concerns

Separation of concerns (SoC) is a fundamental principle of software engineering. It is credited to Dijkstra [Dij76] and Parnas [Par76, Par79] who applied the principle of *divide-and-conquer* to software development: *it is easier to manage a problem by breaking it down into smaller pieces than to solve the problem as is.* Such pieces are the concerns of a software system, where a concern is a semantically coherent issue of a problem domain that is of interest. *Cohesion* is the grade of functional relatedness of the pieces of code that implement a concern [YC79]. High cohesion is preferable because it is associated with several desirable properties of software, e.g., robustness, reliability, reusability, and understandability.

software decomposition

In software development, separation of concerns is related to the decomposition mechanisms of design and implementation. Concerns are the primary criteria for decomposing software into smaller, more manageable and comprehensible parts. The resulting pieces are not the concerns themselves but their representations at design and implementation levels. For example, a concern may be a requirement such as 'realtime operation', a program feature such as 'RSA encryption', a data structure such as a B-tree, or even a tiny issue like implementing a length counter as long integer or as short integer. For simplicity, we equate concerns and their representations in the remaining dissertation.

benefits of separation of concerns

The goal of separation of concerns is to localize, untangle, separate, and encapsulate the representations of concerns in a software system. The following benefits are attributed to software with well separated concerns:

Comprehension: A well structured system is easier to understand [Par79, Dij76]. A localized and separated concern representation enables the programmer to concentrate on that concern in isolation without getting distracted by details of other concerns. Dijkstra formulates this as follows:

> *Our heads are so small that we cannot deal with multiple aspects simultaneously without getting confused.*

Comprehensibility is a critical requirement for tasks like software reuse, customization, and maintenance. Thus, achieving comprehensibility is the primary goal of separation of concerns.

Reuse: Software reuse is the process of creating software systems from existing software rather than building software systems from scratch [Kru92]. Separated concerns can be more easily reused in different contexts than intermingled ones. The more independent a concern is, the easier it can be detached from or attached to a software system. The spectrum of reuse reaches from reusing a concern, i.e., its implementation, in different variants of one software product (e.g., a component) to reusing a concern in different, unrelated software systems (e.g., a library function) [Big98].

Maintenance: Updating, debugging, and evolving a software system are frequent tasks in software maintenance. They usually boil down to adding, removing or changing concern implementations. Parnas was the first to proclaim that change should be considered when designing software; this concept is called *design for change* [Par79]. The idealized goal is to change software as much as possible in a non-invasive way, i.e., by applying new pieces that implement the change and removing unneeded ones instead of modifying existing pieces [OH92, VN96a].

Structuring software along concerns enables (1) the addition of new concerns in form of distinct pieces of software and (2) the modification or exchange of existing concerns in isolation.

Customization: Typically, different stakeholders have different requirements on a software system. Thus, there is a need to customize software to meet the specific needs of stakeholders. Ideally, a software design and implementation is variable, i.e., it supports the easy derivation of system variants. Customizing a software system means adjusting the given system structure in the boundaries of the supported variability [vGBS01]. Separation of concerns is beneficial in that the implementation of a concern can come in different variants and concerns can be combined in

different ways. Customizing software means then to choose the concerns desired and to select those implementations that fit a requirement specification best.

software decomposition and composition

Concerns are separated by decomposing software along concern representations. That is, in all phases of the software life cycle, concerns of a software system are separate pieces, distinguishable from other concerns. However, such separation is non-trivial to achieve, especially in large-scale and evolved software. Design and implementation techniques have to support separation of concerns explicitly by providing appropriate (de)composition mechanisms. *Decomposition* means to break down a software design into pieces; *composition* ties these pieces together to get a complete software product. Design and implementation techniques have to provide different kinds of (de)composition mechanisms at different levels of abstraction in order to account for the diversity of possible concerns. Prominent examples are the concepts of *functions* in *structured programming* and *classes* in OOP. While functions decompose a software system along its instructions, classes decompose a software system along the data to be encapsulated.

The exploration and analysis of (de)composition mechanisms is a major subject of research in software engineering and programming languages. Early work addressed issues like structured programming and information hiding. Recent work aims at software structures at a larger scale and occurring in all phases of the software life cycle. The following sections introduce the design and implementation techniques relevant for this dissertation.

2.2 Stepwise Software Development

Stepwise refinement [Wir71] and *program families* [Par76] are two design methodologies that are fundamental to software engineering. Both address explicitly the issue of separation of concerns. They support the incremental development of software over time by implementing a series of design decisions being applied in several development steps, which is called *stepwise development (SWD)*. This way, the resulting software forms a layered design such that each layer implements a concern that corresponds to a design decision and a development step; subsequently applied layers build up on previously applied layers.

2.2.1 Stepwise Refinement

Wirth was the first to articulate the role of stepwise refinement in program design [Wir71]. According to his view a program (or its specification) is gradually developed in a sequence of *refinement steps*. In each step, the structural elements of the given program (instructions and data) are decomposed into more detailed elements. That is, refinement is the

revealing of design and implementation details that have not yet been exposed and each refinement step implies a *design decision*. The successive decomposition or refinement of program specifications terminates when all structural program elements are expressed in terms of an underlying programming language. Hence, the process of stepwise refinement is a mapping between two representations of a program, where the representation that is refined is more abstract than the representation that results.

A program specification could be written informally as natural language text, e.g.,

> *given an array A of size N, permute the elements of A in such a way that A is sorted in increasing order [Wir76].*

Alternatively a specification could be expressed in a formal (programming or mathematical) language that is usually tailored to a specific problem domain, e.g., information system development [JSHS96], interactive systems [BS01], object modeling [Jac02], or network services [Bow96], to name a few.

refinement tree

Since for each refinement step alternative design decisions are possible, the overall refinement process results in a *refinement tree*. The leaves of a refinement tree define different implementations of the considered program. The path from the root of the tree to a leaf expresses the program's design and implementation – it is a series of refinements that explains how a program implements its specification.

Figure 2.1 depicts an example refinement tree, adopted from [Bax92]. The root of the tree is a program specification in form of an abstract syntax tree, which represents an arithmetic expression $(3 * (y + z) + 4)$. By applying the two transformation rules *dist* and *com* that implement familiar distributivity and commutativity laws the original specification is refined into two new specifications: '$3 * y + 3 * z + 4$' and '$4 + 3 * (z + y)$'. These two alternative refinement steps result in two new leaves of the refinement tree, which are two alternative abstract syntax trees.

With stepwise refinement the programmer makes decisions how to derive a more concrete representation of the program starting from a more general one. The resulting refinement tree contains all alternative design decisions (in our example, applying the distributivity and the commutativity law) made during the refinement process.

2.2.2 Program Family Development

Parnas proposed a related methodology for SWD: a *program family* is a set of similar programs [Par76]. The idea is to concentrate on the commonalities of a set of programs instead of their differences with the goal of sharing functionality between program family members. To achieve the needed degree of reusability within a program family, Parnas and others [Dij68, Dij76] proposed implementing software starting from a *minimal base*

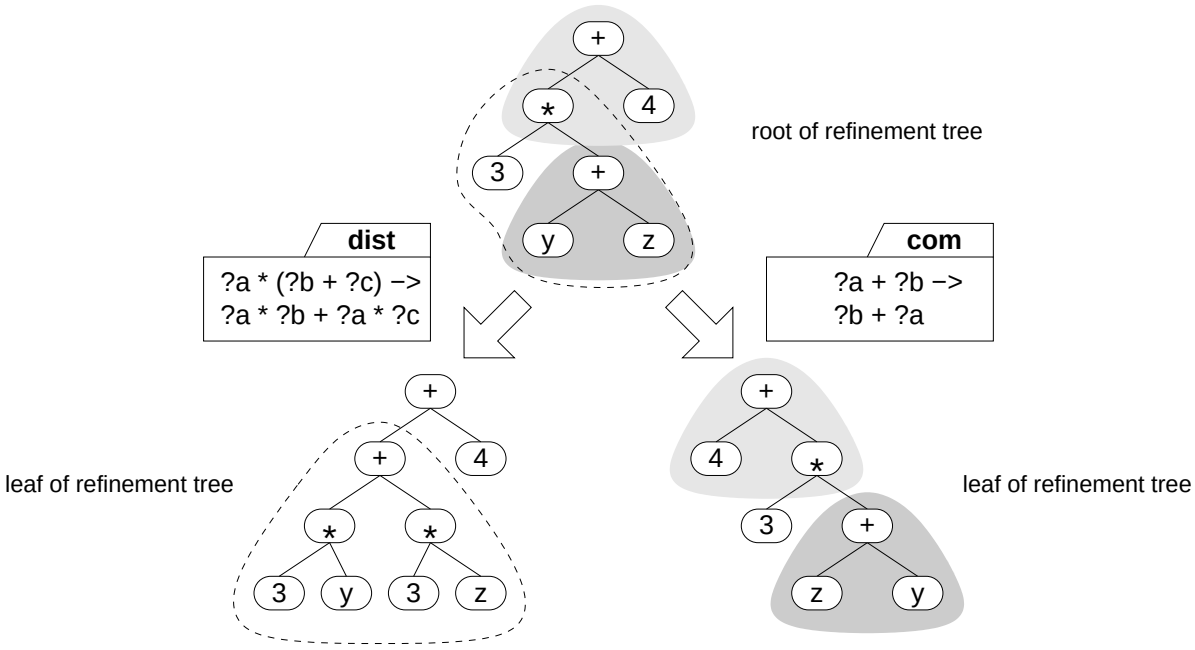


Figure 2.1: Implementing two design decisions by applying two refinements [Bax92]

of functionality and evolving the functionality by adding *minimal extensions* in a series of *development steps*, which leads to conceptually layered designs. Parnas further proposed the concept of modules that implement layers, which we explain soon (Sec. 2.3).

operating system family development

Figure 2.2 depicts the design of a family of operating systems [HFC76]. In contrast to Wirth's refinements, the layers of a program family are displayed in bottom-up order. Starting from the layer '*hardware*', which is the base of the operating system family, the subsequent layers extend previous layers, e.g., layer '*synchronization*' extends layer '*process management*'. Note that one layer can be extended by multiple other layers, e.g., layer '*synchronization*' is extended by '*special devices*' and '*address space creation*'. Different family members consist of different sets of layers. In our example, three family members can be derived, i.e., three operating systems: a batch system, a process control system, and a time sharing system. Adding a layer means extending a whole family of programs because each family member may potentially use this new layer.

2.2.3 Stepwise Refinement Versus Program Families

While Parnas' and Wirth's approaches are not equivalent there are certain fundamental similarities.

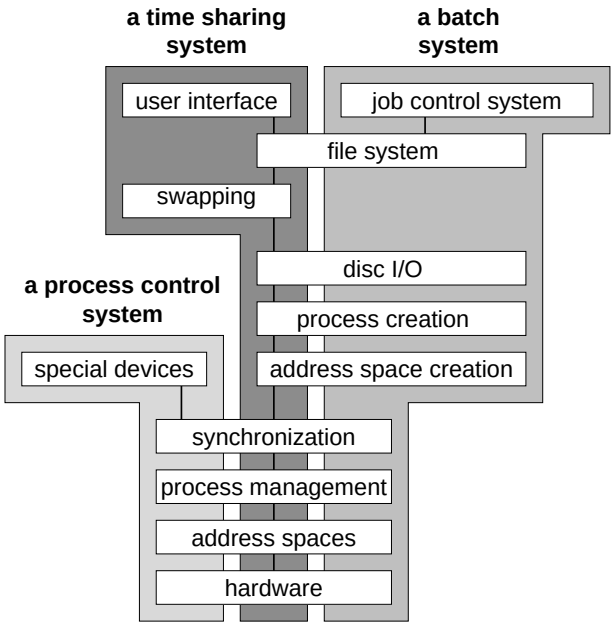


Figure 2.2: A program family of operating systems [HFC76].

Wirth's refinement

Wirth's stepwise refinement has been associated historically with the progressive rewriting of a formal specification of a program into executable code. With each step the program becomes more concrete and eliminates nondeterminism of program behavior. Thus, a refinement does not extend the program behavior but makes it more concrete, e.g., by refining the specification to strengthen the condition '$x > 0$' to '$x = 10$'. With each step the set of possible programs that satisfy a specification decreases.

Parnas' program families

Following Parnas' approach, a family of programs is developed incrementally. The difference to Wirth's approach is that this process starts with a minimal base and proceeds by extending the functionality in order to encapsulate design decisions step by step. The evolution of a program family does not start with a complete specification but with a possibly empty base program. With each step, the set of possible programs that can be derived from the program family increases, which is in contrast to Wirth's approach in which the number of potential programs decreases with each step.

unification of Wirth's and Parnas' worlds

However, an alternative interpretation of Parnas's work is that a programmer starts with a *domain model* that is implemented by a program family. A domain model captures and relates all the knowledge that is of interest to a group of stakeholders [CE00]. By adding successively new extensions to a base the scope of possible programs that share these extensions narrows, i.e., the program family becomes more concrete. We and others [Big98, BSR04] favor this view since it unifies the early work of Wirth and Parnas on stepwise software development.

Adopting this interpretation we define a *refinement* as a set of changes applied to a program. That is, a refinement extends a program by adding new constructs and it modifies the existing structures of a program. This excludes the mere removal of existing structural elements. A refinement is associated with a *development step* and can be understood as concern being implemented.

2.2.4 Software Product Lines

Research on *software product line (SPL)* development is related to SWD (especially to program families) with a special focus on economics. The Carnegie Mellon Software Engineering Institute (SEI) describes a SPL as follows[1]:

> *A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying 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.*

Furthermore, the SEI makes the following statement as to why SPLs are important:

> *Software product lines are rapidly emerging as a viable and important software development paradigm allowing companies to realize order-of-magnitude improvements in time to market, cost, productivity, quality, and other business drivers. Software product line engineering can also enable rapid market entry and flexible response, and provide a capability for mass customization.*

structural features

To achieve the advantages stated above, Czarnecki argues that the ideal way of SPL development is to implement a SPL as a program family [CE00]. That is, each layer (or a set of layers) of a program family implements a feature of the corresponding SPL, where a feature corresponds to a (set of) core asset(s). Furthermore, it is assumed that the considered features are *structural features* [LBN05]. A structural feature is a feature that has an explicit representation at design and implementation level. That is, the assets of a feature are physically or visually represented, e.g., by files, program text, design documents.

emergent features

This definition excludes those features – if they are even features in the sense of domain modeling – that implement program behaviors that emerge indirectly from the combination of other features *at runtime*, which is in science widely known as *emergent behavior* [Mog06, Lod04]. For example, security characteristics of software emerge from the concrete composition of features when the program is running [Lip05]; there is not one or a set of assets that represent the security feature.

[1] http://www.sei.cmu.edu/productlines/

SPLs and program families

The approach of implementing SPLs as program families leads to a small time to market and a high degree of reusability and customizability since new, tailored products can be derived more easily by composing the layers that correspond the desired features [CN02, GS04, CE00]. This also implies that SPLs are implemented in a stepwise manner, true to the motto of SWD.

2.3 Modules

What is a module?

A *module* is a structural mechanism that facilitates separation of concerns. The idea of modules emerged from several tracks of research, namely *modular programming* [Con68], *program specification* [Par72a, Par72b], *structured programming* [DDH72, Dij76], and *structured design* [SMC74, YC79]. Today it has been agreed that modules are self-contained, cohesive pieces of a software system, where *cohesive* refers to the ability of a module to localize program and data structures physically, e.g., in program text or in the file system. A module has a well-defined interface for communicating with other modules and it can be compiled separately. *Modularity* is the principle to structure software into modules. A more quantitative definition is that modularity measures the extent to which modules are used in a software system.

information hiding and encapsulation

Modules embody the principle of *information hiding* [Par72b]. This principle states programmers should hide those design decisions in a software system that are most likely to change (*design for change*), thus protecting other parts of the program from modification if the design decision is changed. Often, information hiding is used synonymously with *encapsulation*, where a module encapsulates data and program structures. Information hiding and modules facilitate separation of concerns since a concern implementation (module) becomes decoupled from other concern implementations. Due to the encapsulation property, modules can be modified or even exchanged without affecting other modules.

modules vs. classes

The concept of modules has evolved to object-oriented language constructs such as classes. Their primary focus is not on separate development but on structuring software to improve comprehensibility, reusability, maintainability, customizability, and evolvability [Boo93, GHJV95]. Like a module a class encapsulates data and program structures and provides an interface (information hiding). Classes can be aggregated hierarchically to form compound classes. In contrast to the early idea of modules, classes can be instantiated and support inheritance and subtype polymorphism. Hence, with respect to its static properties, a class (or a set of classes) can be understood as a traditional module.

2.4 Feature-Oriented Programming

2.4.1 Features, Concerns, and Collaborations

What is a feature?

Research on *feature-oriented programming (FOP)* studies the modularity of *features* in software product lines, where a feature is an increment in program functionality [BSR04]. The concept of features is closely related to that of concerns – some researchers even equate them [MLWR01]. We prefer a different view: while features reflect directly the requirements of the stakeholders and are used to specify and distinguish different software products [KCH+90, CE00], concerns are at a lower level, more fine-grained, and not in any case of interest to stakeholders. Features are concerns, but not all concerns are features.

feature modules

Feature modules are modules that realize features at design and implementation levels. They support information hiding by exploiting underlying OOP mechanisms. They are be composed statically and can be compiled independently. Typically, features modules refine the content of other features modules in an incremental fashion. This follows directly the early principles of SWD. The goal of FOP is to synthesize software (individual programs) by composing a series of desired feature modules. As feature modules reflect the requirements on a software, FOP bridges the gap between analysis, design, and implementation. We use the terms feature and feature module in the remaining dissertation interchangeable.

collaborations

An important observation is that features are implemented seldomly by single classes but instead by a whole set of collaborating classes, where a *collaboration* is a set of classes that communicate with one another to implement a feature [RAB+92, VN96c, MO04, LLO03, BSR04, SB02, OZ05, Ern01, Ern03]. Feature modules abstract and explicitly represent such collaborations. Hence, FOP stands in the long line of prior work on object-oriented design and role modeling, as surveyed in [Ste00].

roles

Classes play different *roles* in different collaborations [VN96c]. A role encapsulates the behavior or functionality that a class provides when a corresponding collaboration with other classes is established – or in context of FOP, when a corresponding feature module is present. That is, a role is that part of a class that implements the communication protocol with other classes participating in a particular collaboration. Figure 2.3 shows four classes participating in three collaborations. For example, class *A* participates in collaboration *I* and *II*, i.e., two distinct roles implement the communication protocol necessary for these collaborations.

From the FOP perspective, each role is implemented by a refinement (declared by the keyword `refines`). That is, a role adds new elements to a class and extends existing elements, such as methods. Usually features extend a program by adding several new

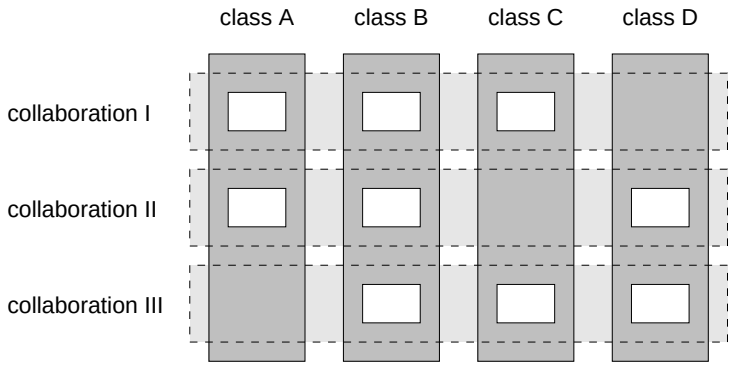


Figure 2.3: Collaboration-based design.

classes *and* by applying several new roles to existing classes simultaneously. Hence, the implementation of a feature cuts across several places in the base program.

Figure 2.4 depicts the collaboration-based design of a simple program that deals with graph data structures. The diagram uses the *UML* notation [BRJ05] with some extensions: white boxes represent classes *or* roles; gray boxes denote collaborations; solid arrows denote refinement, i.e., to add a new role to a class.

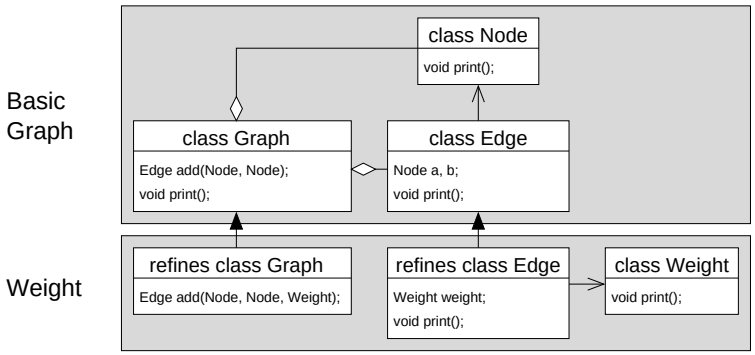


Figure 2.4: Collaboration-based design of a graph implementation.

The feature *BasicGraph* consists of the classes `Graph`, `Node`, and `Edge` that together provide functionality to construct and display graph structures[2]. The feature *Weight* adds roles to `Graph` and to `Edge` as well as a class `Weight` to implement a weighted graph, i.e., a graph that assigns to each edge a specific weight value.

[2] In this dissertation we write feature names in *italic* fonts and names of internal elements of features (e.g., classes, methods, fields) in `typewriter` fonts.

2.4.2 Jak: FOP for Java

Jak[3] is an extension of Java for FOP. It supports a special language construct to express refinements of classes, e.g., for implementing roles. Classes in Jak are implemented as standard Java classes. Figure 2.5 depicts our feature *BasicGraph* implemented in Jak[4]. It consists of the classes `Graph` (Lines 1-15), `Node` (Lines 16-20), and `Edge` (Lines 21-28). A programmer can add nodes (Lines 3-7) and print out the graph structure (Lines 8-14).

Jak constants

```
1  class Graph {
2      Vector nodes = new Vector(); Vector edges = new Vector();
3      Edge add(Node n, Node m) {
4          Edge e = new Edge(n, m);
5          nodes.add(n); nodes.add(m);
6          edges.add(e); return e;
7      }
8      void print() {
9          for(int i = 0; i < edges.size(); i++) {
10             ((Edge)edges.get(i)).print();
11             if(i < edges.size() - 1)
12               System.out.print(", ");
13         }
14     }
15 }
16 class Node {
17   int id = 0;
18   Node(int _id) { id = _id; }
19   void print() { System.out.print(id); }
20 }
21 class Edge {
22   Node a, b;
23   Edge(Node _a, Node _b) { a = _a; b = _b; }
24   void print() {
25     System.out.print(" ("); a.print(); System.out.print(", ");
26     b.print(); System.out.print(") ");
27   }
28 }
```

Figure 2.5: A simple graph implementation (*BasicGraph*).

A refinement in Jak encapsulates the changes a feature applies to a class. It is declared by the keyword `refines`. A sequence of refinements applied to a class is called *refinement chain*, i.e., a class composed with a series of refinements forms a new class.

Jak refinements

A refinement in Jak is implemented by a *mixin* [BC90, SB02]. A mixin is an *abstract subclass* that can be applied to various classes to form a new classes. Composing a mixin and a class is called *mixin composition*; the relationship between mixin and superclass

mixin composition

[3] http://www.cs.utexas.edu/users/schwartz/ATS.html

[4] For simplicity, we merge in code listings all classes and all refinements of a feature into one piece of code; in truth each class or refinement is located in a distinct file.

is called *mixin-based inheritance*, a form of inheritance that delays the coupling between subclass and superclass until composition time (a.k.a. *mixin instantiation*). Alternative implementation mechanisms for refinements are *virtual classes* [MMP89, EOC06, OZ05], *traits* [DNS+06], or *nested inheritance* [NCM04, NQM06].

Figure 2.6 depicts the feature *Weight* implemented in Jak: it introduces a class that represents the weight of an edge (Lines 15-19); it refines the class `Graph` (Lines 1-6) by introducing a new method `add` that assigns a weight value to an edge (Lines 2-5); it refines the class `Edge` (Lines 7-14) by adding a field (Line 8) and a method for assigning the weight value (Line 9) and by extending the `print` method to display the weight (Lines 10-13).

A *method extension* is implemented by overriding the method to be extended, adding code, and calling the overridden method via the keyword `Super`[5] (Lines 3,11).

```
1  refines class Graph {
2    Edge add(Node n, Node m, Weight w) {
3      Edge res = Super.add(n, m);
4      res.setWeight(w); return res;
5    }
6  }
7  refines class Edge {
8    Weight w = new Weight(0);
9    void setWeight(Weight _w) { w = _w; }
10   void print() {
11     Super.print();
12     System.out.print(" ["); w.print(); System.out.print("] ");
13   }
14 }
15 class Weight {
16   int w = 0;
17   Weight(int _w) { w = _w; }
18   void print() { System.out.print(w); }
19 }
```

Figure 2.6: Adding support for weighted graphs (*Weight*).

Jak feature modules

Jak's feature modules are represented by file system directories. Thus, they have no textual representation at the code level. The artifacts, i.e., classes and refinements found inside a directory are members (assets) of the enclosing feature. Figure 2.7 shows the directory hierarchy of our graph example, including the features *BasicGraph*, *Weight*, and *Color*.

In its current version, Jak supports separate compilation of feature modules but does not support explicit interfaces, i.e., the interface of a feature module is the sum of the

[5] We capitalize `Super` to emphasize the difference to the Java keyword `super`, which refers to the parent type of a class (traditional inheritance). For brevity we write `Super` instead of `Super(`<*argument types*>`)`, which is used actually in Jak.

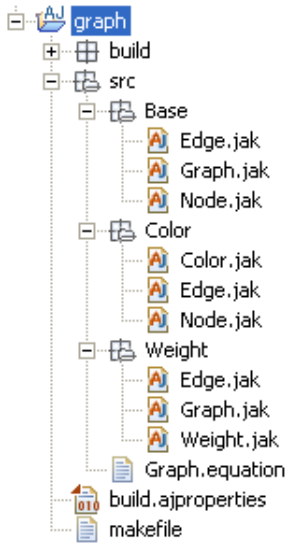


Figure 2.7: Directory structure of a graph implementation.

interfaces of the participants of the encapsulated collaboration. However, other FOP languages support collaboration interfaces [MO02].

2.4.3 GenVoca

GenVoca[6] is an algebraic model for FOP [BO92]. Features are modeled as operations of an algebra. Each SPL is modeled by one associated algebra, which is called a *GenVoca model*. For example, '*Graph = {BasicGraph, Weight, Color}*' denotes a model *Graph* that has the features *BasicGraph*, *Weight*, and *Color*.

constants and functions

Features are modeled as *functions*. A *constant function* (a.k.a. *constant*) represents a base program. All other functions receive programs as input and return modified programs as output. That is, functions represent program refinements that implement program features. For example, '*Weight* • *X*' and '*Color* • *X*' add features to program *X*, where '•' denotes function composition. The design of a software product is a named *feature expression*, e.g., '*WeightedGraph = Weight* • *BasicGraph*' and '*ColoredWeightedGraph = Color* • *Weight* • *BasicGraph*'. Note that not all possible feature expressions must be valid, i.e., there may be expressions represent syntactically or semantically in-

[6] The name *GenVoca* is derived from the systems *Genesis* [BBG+88, Bat88] and *Avoca* [PHOA89] that demonstrated first the duality between refinement and modules in different domains (i.e., data management and network protocols); GenVoca refers to the underlying domain-independent methodology to develop software by stepwise refinement.

correct programs [BG97, Bat05]. The set of all valid feature expressions corresponds to the SPL, i.e., all derivable products of a given GenVoca model.

2.4.4 AHEAD

principle of uniformity

AHEAD (Algebraic Hierarchical Equations for Application Design) is an architectural model for large-scale program composition and the successor of GenVoca [BSR04]. It scales the ideas of GenVoca to all kinds of software artifacts. That is, features do not only consist of source code but of all artifacts that contribute to that feature, e.g., documentation, test cases, design documents, makefiles, performance profiles, mathematical models. Furthermore, the *principle of uniformity* states that every kind of software artifact that is part of a feature can be subject of subsequent refinement [BSR04].

containment hierarchy

With AHEAD, each feature is represented by a *containment hierarchy*, which is a directory that maintains a subdirectory structure to organize the feature's artifacts. Composing features means composing containment hierarchies and, to this end, composing corresponding artifacts by *hierarchy combination* [OH92] (a.k.a. *mixin composition* [BC90, SB02, OZ05], *hierarchy inheritance* [Ern03], or *superimposition* [Bos99, BF88, CM86, Kat93]). Hence, for each artifact type a different implementation of the *composition operator* has to be provided.

Figure 2.8 shows the features *BasicGraph* and *Weight*; each consists of several source code files as well as an HTML documentation; *BasicGraph* contains additionally an XML build script. The feature expression '$WeightedGraph = Weight \bullet BasicGraph$' combines both features, which is implemented as a recursive combination of their containment hierarchies. For example, the resulting file *Edge.jak* is composed of its counterparts in *BasicGraph* and in *Weight*. The composition is specific to the type of the software artifact, e.g., composing HTML is different from composing XML or Java.

The *AHEAD Tool Suite (ATS)*[7] implements the ideas of AHEAD. It contains several tools for developing, debugging, and composing source code and non-source code artifacts. The Jak language is integrated into the ATS and there are tools to compose Java-based source code artifacts.

[7] http://www.cs.utexas.edu/users/schwartz/ATS.html

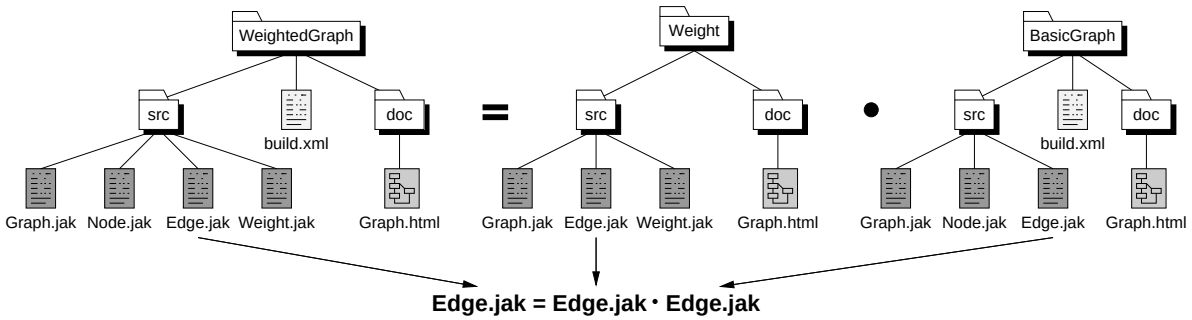


Figure 2.8: Combining the containment hierarchies of two features.

2.5 Aspect-Oriented Programming

2.5.1 Crosscutting Concerns

Aspect-oriented programming (AOP) is a programming paradigm that aims at modularizing *crosscutting concerns* [KLM+97, EFB01]. Crosscutting is a structural relationship between the representations of two concerns. In other words, a representation of a concern crosscuts the representation of another concern. Crosscutting is an alternative structural relationship to hierarchical and block structure. It is not defined between concerns but between their representations, i.e., the modules that implement the concerns.

collaborations are crosscuts

In our remarks on FOP, we have already considered a kind of crosscutting concern: collaborations extend a program at different places, thus cutting across the module boundaries introduced by classes. Feature modules modularize collaborations, which implement features. AOP considers crosscutting concerns in general, without special focus on feature modularity or collaborations.

tyranny of the dominant decomposition

Traditional languages and modularization mechanisms suffer from a limitation that is referred to as the *tyranny of the dominant decomposition*, which seems to be the cause of crosscutting [TOHSMS99]: a program can be modularized in only one way (along one dimension) at a time, and the many kinds of concerns that do not align with that modularization end up in *scattered*, *tangled*, and *replicated code*. Figure 2.9 illustrates different dimensions of separation of concerns, e.g., along the feature dimension or the object dimension[8].

[8] http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm

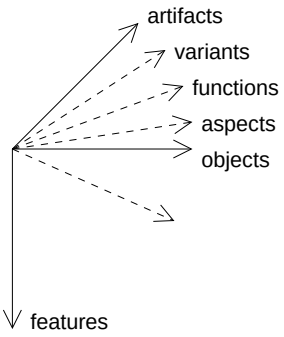


Figure 2.9: Dimensions of separation of concerns.

code scattering, tangling

Code scattering refers to a concern implementation that is scattered across many other concerns implementations; *code tangling* refers to the intermingled implementation of several concerns within a module. Both decrease modularity and violate the principle of information hiding [KLM+97, EFB01, Kic06].

Figure 2.10 shows how the implemention the feature *Color* crosscuts our basic graph implementation (the code associated with the feature *Color* is underlined). The classes `Node` and `Edge` get a field `color` (Lines 3,14) and two methods `setColor` (Lines 4,15) and `getColor` (Lines 5,16). Further on, the `print` methods of `Node` and `Edge` are modified to display the colors appropriately (Lines 9,20). The implementation of the feature *Color* is scattered across three classes (`Color`, `Node`, `Edge`) and within these classes it changes two methods. Moreover, it is tangled with the feature *Display* for displaying the graph structure, which is itself scattered over `Graph`, `Node`, and `Edge`.

code scattering and tangling degrade comprehensibility

Code scattering and tangling degrade a program's comprehensibility. The programmer becomes distracted when dealing with tangled code, i.e., code that addresses multiple concerns. Scattered code forces the programmer to reason about a concern in multiple places of a program. Overall, scattered and tangled code decreases reusability, maintainability, and customizability since the concerns become coupled – short their implementation violates the principle of separation of concerns [KLM+97, EFB01].

code replication

A further negative effect of crosscutting is code replication, which occurs typically when a concern interacts with multiple concerns and all interactions are implemented identically. For example, the implementation of our feature *Color* results in code for managing and changing colors that is replicated in the classes `Edge` and `Node`. It has been observed that code replication is a serious problem: beside the handicap of reimplementing the same functionality again and again, code replication reduces software maintainability [FR99] and is a potential substrate for errors caused by copy and paste of code fragments [LLM06].

```
1  class Graph { /* ... */ }
2  class Node {
3    Color color;
4    void setColor(Color c) { color = c; }
5    Color getColor() { return color; }
6    int id = 0;
7    Node(int _id) { id = _id; }
8    void print() {
9      Color.changeDisplayColor(getColor());
10     System.out.print(id);
11   }
12 }
13 class Edge {
14   Color color;
15   void setColor(Color c) { color = c; }
16   Color getColor() { return color; }
17   Node a, b;
18   Edge(Node _a, Node _b) { a = _a; b = _b; }
19   void print() {
20     Color.changeDisplayColor(getColor());
21     System.out.print(" ("); a.print(); System.out.print(", ");
22     b.print(); System.out.print(") ");
23   }
24 }
25 class Color {
26   static void changeDisplayColor(Color c) { /* ... */ }
27 }
```

Figure 2.10: Implementing the feature *Color* leads to code scattering, tangling, and replication (code associated to the feature *Color* is underlined).

2.5.2 Aspects: An Alternative Modularization Mechanism

AOP addresses the problems caused by crosscutting concerns as follows: concerns that can be modularized well using the given decomposition mechanisms of a programming language (a.k.a. *host programming language*) are implemented using these mechanisms. All other concerns that crosscut the implementation of other concerns are implemented as so-called *aspects*.

An aspect is a kind of module that encapsulates the implementation of a crosscutting concern. It enables code that is associated with one crosscutting concern to be encapsulated into one module, thereby eliminating code scattering and tangling. Moreover, aspects can affect multiple other concerns via one piece of code, thereby avoiding code replication.

aspect weaving

An *aspect weaver* merges the separate aspects of a program and the remaining program elements at predefined *join points*. This process is called *aspect weaving*. Join points can be syntactical elements of a program, e.g., a class declaration, or events in the dynamic execution of the program, e.g., a call to a method in the control flow of another method.

Figure 2.11 illustrates the weaving of two aspects into a base program consisting of three components.

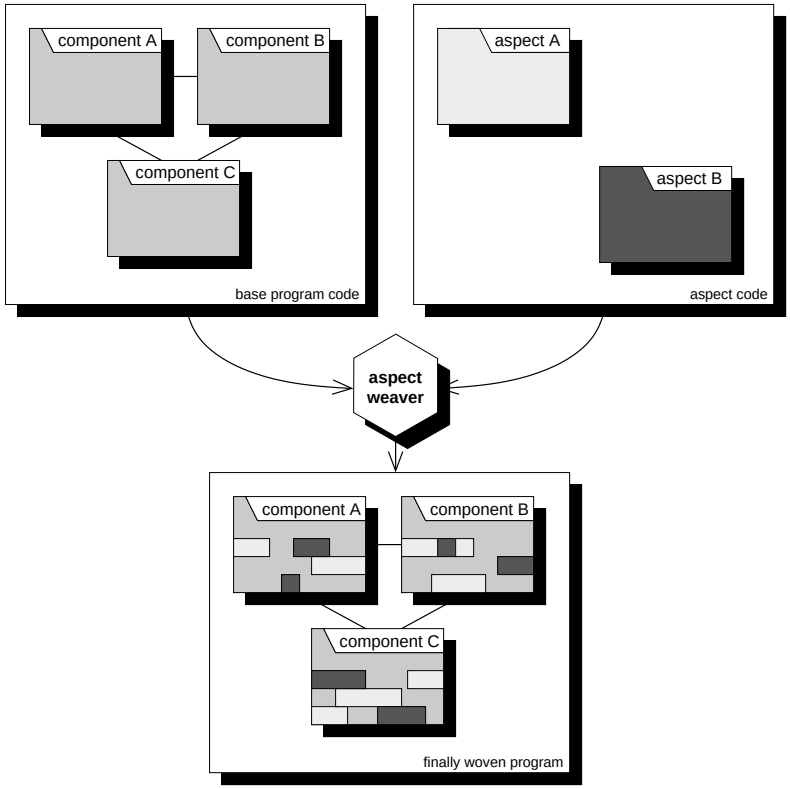


Figure 2.11: Aspect weaving.

aspects violate information hiding

Although often referred to as modularization mechanism, the traditional aspect violates the principle of information hiding [LLO03, Ald05, SGS+05, DW06]: while the aspect itself has an interface, it affects other modules directly, without the indirection of an interface. This precludes developing and modifying modules independently. However, it has been argued that traditional modularization mechanisms themselves do not perform well with respect to crosscutting concerns [LLO03, Kic06]. Hence, aspects seem to be a pragmatic alternative. There are several efforts that aim at restoring information hiding in AOP [Ald05, OAT+06, DW06, SGS+05].

aspects vs. classes

In most AOP languages the concept of an aspect extends the concept of a class. Besides structural elements known from OOP, e.g., methods and fields, aspects may contain also *pointcuts*, *advice*, and *inter-type declarations*.

Pointcuts: A *pointcut* is a declarative specification of the join points that an aspect will be woven into, i.e., it is an expression (quantification) that determines whether a given join point matches.

Advice: An *advice* is a method-like element of an aspect that encapsulates the instructions that are supposed to be executed at a set of join points. Pieces of advice are bound to pointcuts that define the set of join points being *advised*.

Inter-type declarations: An inter-type declaration adds methods, fields, or interfaces to existing classes from inside an aspect.

2.5.3 AspectJ: AOP for Java

AspectJ[9] is an AOP language extension of Java. Figure 2.12 illustrates how an aspect in concert with a class and an interface implements our *Color* feature. The dashed arrows denote the structural elements of the graph implementation affected by the aspect (only a subset is depicted). The AspectJ weaver merges the aspect implementation and the basic graph implementation.

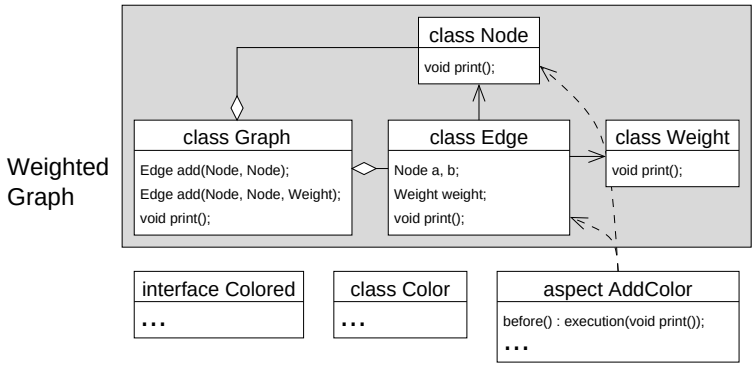


Figure 2.12: Implementing the *Color* feature as aspect.

Figure 2.13 depicts one possible implementation of the *Color* feature in AspectJ. The aspect `AddColor` defines an interface `Colored` for all classes that maintain a color (Line 2) and declares via inter-type declaration that `Node` and `Edge` implement that interface (Line 4). Furthermore, it introduces via inter-type declarations a field `color` and two accessor methods to `Node` and `Edge`. Finally, it advises the execution of the method `print` of all colored entities, i.e., `Edge` and `Node`, to change the display color.

inter-type declarations for multiple types

Note that, in AspectJ, one cannot declare one field or method for multiple types simultaneously. This leads to a replication of code in our *Color* feature: the code for introducing the field `color` and the two accessor methods is replicated (Fig. 2.13, Lines 6-8 and 10-12). To overcome this limitation of AspectJ, we prefer the following syntax in the remaining dissertation: `Color (Node || Edge).color` introduces a field `color` to the

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

```
1  aspect AddColor {
2      interface Colored { Color getColor(); }
3
4      declare parents: (Node || Edge) implements Colored;
5
6      Color Node.color;
7      void Node.setColor(Color c) { color = c; }
8      public Color Node.getColor() { return color; }
9
10     Color Edge.color;
11     void Edge.setColor(Color c) { color = c; }
12     public Color Edge.getColor() { return color; }
13
14     before(Colored c) : execution(void *.print()) && this(c) {
15         Color.changeDisplayColor(c.getColor());
16     }
17 }
```

Figure 2.13: Implementing the *Color* feature using AspectJ (excerpt).

types `Node` and `Edge`. Using this syntax we can eliminate the redundant code caused by inter-type declarations in our aspect `AddColor`, as shown in Figure 2.14.

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

Figure 2.14: A more compact syntax for inter-type declarations in AspectJ.

2.6 Terminology Used in this Dissertation

In the remaining dissertation we use the following terminology and conventions. We assume that a SPL is implemented as a program family in a SWD manner. A series of features refines a given base program in several development steps. A refinement encapsulates a set of changes made to a program, i.e., it adds new structures and modifies existing ones.

We consider AOP and FOP techniques for SWD of SPLs with the primary goal of separation of concerns and feature modularity. Consequentially, aspects and feature modules

implement structural features of a SPL. An aspect is a class-like implementation mechanism that contains additionally pointcuts, advice, and inter-type declarations, as exemplified by the AspectJ programming language. A feature module is an implementation mechanism that supports the encapsulation of a collaboration of several software artifacts, as exemplified by the Jak programming language. Furthermore, feature modules are composed by mixin composition.

CHAPTER 3

A Classification Framework for Crosscutting Concerns

This chapter shares material with the ICSE'06 paper 'Aspectual Mixin Layers: Aspects and Features in Concert' [ALS06], and the AOPLE'06 paper 'On the Structure of Crosscutting Concerns: Using Aspects or Collaborations?' [ABR06].

In order to compare FOP and AOP, we present a classification framework for crosscutting concerns. Subsequently, we demonstrate that AOP and FOP perform differently in modularizing the different classes of crosscutting concerns.

Within our framework, we classify crosscutting concerns (*crosscuts* for short) along two dimensions: (1) the structure of a crosscut can be homogeneous or heterogeneous and (2) concerns can crosscut the static structure or the dynamic structure of a program.

3.1 Homogeneous and Heterogeneous Crosscutting Concerns

homogeneous crosscuts

A *homogeneous crosscut* extends a program at multiple join points by adding one *extension*, which is a modular piece of code [CRB04]. For example, our *Color* feature is a homogeneous crosscut. It extends the two classes `Node` and `Edge` in the same way (cf. Fig. 2.12 and Fig. 2.13): the aspect `AddColor` contains an advice that advises two method executions (`print` in `Node` and `Edge`) and four inter-type declarations that introduce members and an interface to both classes, `Node` and `Edge`.

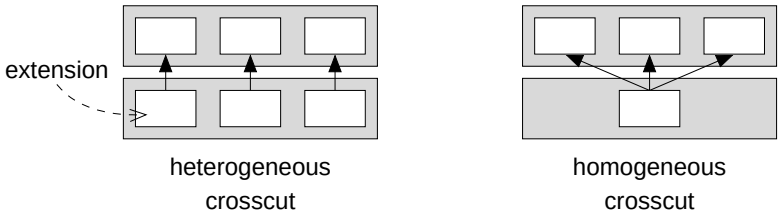


Figure 3.1: Homogeneous and heterogeneous crosscuts.

heterogeneous crosscuts

A *heterogeneous crosscut* extends multiple join points by adding multiple extensions, where each individual extension is implemented by a distinct piece of code, which affects exactly one join point [CRB04]. For example, our *Weight* feature is a heterogeneous crosscut (cf. Fig. 2.4 and Fig. 2.6). It extends the classes `Graph` and `Edge` but each in a different way: the refinement of `Graph` introduces the method `add`; the refinement of `Edge` introduces the method `setWeight` and the field `weight`, and it extends the method `print`.

comparison

Figure 3.1 illustrates the difference between homogeneous and heterogeneous crosscuts. White boxes denote the individual extensions made to a program, e.g., encapsulated in classes, methods, or advice. Gray boxes denote the program and the crosscut that affects the program. Figure 3.1 indicates that a homogeneous crosscut can be implemented using a set of distinct extensions, like a heterogeneous crosscut; but this results in code replication. For example, Figure 3.2 depicts an aspect with one piece of advice that advises three methods; Figure 3.3 depicts an equivalent aspect but with three distinct pieces of advice that advise only one method each – all with an identical advice body.

```
1 aspect FooAspect {
2   after() : execution(void A.foo()) ||
3             execution(void B.foo()) ||
4             execution(void C.foo()) {
5     /* do something */
6   }
7 }
```

Figure 3.2: A homogeneous crosscut implemented using one piece of advice.

```
1  aspect FooAspect {
2    after() : execution(void A.foo()) {
3      /* do something */
4    }
5    after() : execution(void B.foo()) {
6      /* do something */
7    }
8    after() : execution(void C.foo()) {
9      /* do something */
10   }
11 }
```

Figure 3.3: A homogeneous crosscut implemented using three pieces of advice.

3.2 Static and Dynamic Crosscutting Concerns

static crosscuts

A *static crosscut* extends the structure of a program statically [MO04], i.e., it adds new classes and interfaces and injects new fields, methods, interfaces, etc. Note that method extensions are not static crosscuts, as we will explain soon. AspectJ's inter-type declarations and Jak's refinements that introduce new members are examples of implementations of static crosscuts (Fig. 3.4).

```
1 refines class Edge {
2   Weight w = new Weight(0);
3   void setWeight(Weight _w) {
4     w = _w;
5   }
6 }
```

```
1 aspect AddWeight {
2   Weight Edge.w = new Weight(0);
3   void Edge.setWeight(Weight _w) {
4     w = _w;
5   }
6 }
```

Figure 3.4: Implementing static crosscuts in Jak (left) and AspectJ (right).

dynamic crosscuts

A *dynamic crosscut* affects the runtime control flow of a program [MO04]. The semantics of a dynamic crosscut can be understood and defined in terms of an event-based model [WKD04, Läm99]: a dynamic crosscuts runs additional code when predefined events occur during the program execution. Such events are also called *dynamic join points* [MK03b, WKD04, OMB05]. Examples of programming constructs that implement dynamic crosscuts are method extensions in Jak (via overriding) and advice in AspectJ (Fig. 3.5). While the former is limited to method-related join points [MO04], the latter may advise a more sophisticated set of events.

```
1 refines class Edge {
2   void print() {
3     Super.print();
4     System.out.print(" [");
5     w.print();
6     System.out.print("] ");
7   }
8   /* ... */
9 }
```

```
1 aspect AddWeight {
2   after(Edge e) :
3     execution(void Edge.print()) && this(e) {
4       System.out.print(" [");
5       e.w.print();
6       System.out.print("] ");
7   }
8   /* ... */
9 }
```

Figure 3.5: Implementing dynamic crosscuts in Jak (left) and AspectJ (right).

comparison

Figure 3.6 illustrates the difference between static and dynamic crosscuts. The left shows a static crosscut. It crosscuts the static structure of a program, here represented as a class graph. White boxes denote classes or their extensions; empty arrows denote inheritance and filled arrows denote the application of extensions; an extension to a dashed box means the introduction of a new class.

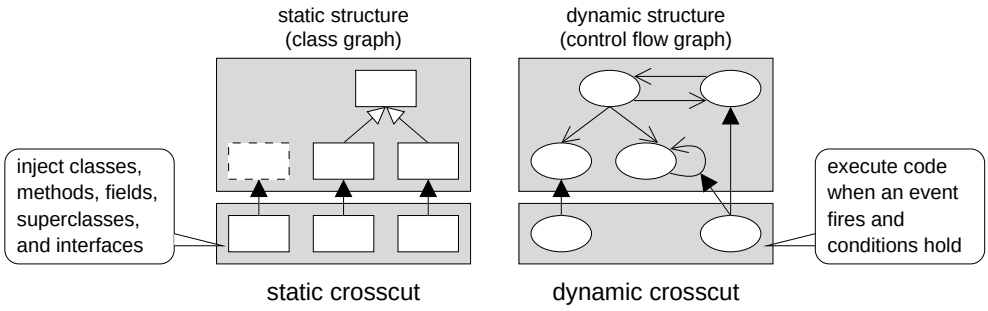


Figure 3.6: Static and dynamic crosscuts.

The right shows a dynamic crosscut. It affects the dynamic structure of the program, here represented as a control flow graph. The extensions are applied to join points that are events in the dynamic control flow. In our example, the elements of the control flow graph are method executions and the arrows between them are calls. An extension may by applied to a method call or a method execution.

Note that dynamic crosscutting should not be confused with *dynamic weaving*, which refers to the weaving of code at loadtime or runtime [PGA02, SCT03, BHMO04].

Basic and Advanced Dynamic Crosscuts

Dynamic crosscuts are especially interesting when they not only affect 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. Capturing and advising these events using traditional OOP mechanisms results in complicated workarounds. There are many proposals for new language constructs for defining and catching new kinds of events during the program execution [OMB05, HG06, MK03a]. In order to distinguish these new kinds of events and the novel language mechanisms that support them from simpler events known from OOP, we distinguish between *basic dynamic crosscuts* and *advanced dynamic crosscuts*, which we define as follows:

1. A basic dynamic crosscut addresses only events that are related to method calls and executions; advanced dynamic crosscuts address all kinds of events, e.g., throwing an exception or assigning a value to a field.

2. A basic dynamic crosscut accesses only runtime variables that are related to the method call or execution that is advised, i.e., arguments, result value, and enclosing object instance of the advised method; advanced dynamic crosscuts can expose more information related to a join point, e.g., the runtime type of the caller of a method.

3. Basic dynamic crosscuts affect a program control flow unconditionally; advanced dynamic crosscuts may specify a condition that is evaluated at runtime, e.g., a method execution is only affected if it occurs in the control flow of another method execution.

4. Basic dynamic crosscuts address only simple events; advanced dynamic crosscuts can specify composite events and event patterns, e.g., *trace matches* are executed when events fire in a specific pattern, thus, involving the history of computation [AAC+05].

basic dynamic crosscuts are method extensions

Principally, basic dynamic crosscuts can be implemented as method extensions using traditional OOP. They extend a method execution/call unconditionally and access only information that is available in method extensions, i.e., the arguments, the result, and the enclosing runtime object.

3.3 Summary: Classification Matrix

Table 3.1 contrasts several examples of the different classes of crosscutting concerns written in AspectJ. It can be seen that our classification framework lays out the set of possible crosscutting concerns in a two-dimensional space. Homogeneous as well as heterogeneous crosscuts can be either static or dynamic. Dynamic crosscuts can be basic dynamic and advanced dynamic.

	homogeneous	**heterogeneous**
static	** Introducing a method to two classes **/ `void (Point || Shape).setX(int x)` `{ /*...*/ }`	*/* Introducing a method to one class */* `void Point.setX(int x)` `{ /*...*/ }`
basic dynamic	*/* Advising a set of method executions */* `before() : execution(* set*(..))` `{ /*...*/ }`	*/* Advising one method execution */* `before() : execution(void Point.setX(int))` `{ /*...*/ }`
advanced dynamic	*/* Advising a set of method executions dependently on the program control flow */* `before() : execution(* set*(..)) && !cflow(execution(* rotate(..)))` `{ /*...*/ }`	*/* Advising one method execution dependently on the program control flow */* `before() : execution(void Point.setX(int)) && !cflow(execution(void Line.rotate(double)))` `{ /*...*/ }`

Table 3.1: Classification matrix with AspectJ examples.

issues to address

Given these different classes of crosscutting concerns it is straightforward to ask whether and how FOP and AOP support their modularization. Also it is interesting to contemplate how often these different kinds of concerns occur when implementing program features and which mechanisms are beneficial for implementation. This kind of knowledge

helps (1) build better tools that reflect the programmer's needs; (2) provide programming guidelines for exploiting programming mechanisms better; (3) discover misuse of programming mechanisms.

CHAPTER 4

A Conceptual Evaluation of Aspect-Oriented and Feature-Oriented Programming

This chapter shares material with the ICSE'06 paper 'Aspectual Mixin Layers: Aspects and Features in Concert' [ALS06].

This chapter presents a conceptual evaluation and comparison of AOP and FOP with respect to implementing program features. First, we propose a set of evaluation criteria that build upon our classification framework for crosscutting concerns. Then, we apply our criteria to evaluate and compare AOP and FOP. Finally, we put our results in perspective and formulate a goal statement for this dissertation.

focus on programming support

In our evaluation we focus exclusively on the implementation mechanisms associated to FOP and AOP. We do not take software development methodologies, tool support, type systems, or mathematical foundations discussed in context of AOP and FOP into account. For FOP this means that a feature module encapsulates a collaboration of software artifacts that are composed by mixin composition. For AOP this means that an aspect is a class-like entity that contains additionally pointcuts, advice, and inter-type declarations.

4.1 Evaluation Criteria

4.1.1 Abstraction

Abstraction in computer science helps to manage the complexity of software [Sha84]. Abstraction is the process of emphasizing and hiding the details of software at different

levels and to different degrees. Abstraction refers also to a construct or concept that has been subjected to the process of abstraction [Kru92]. Separation of concerns and modules are the enabling technologies for abstraction. But abstraction is more than breaking down a system into modules. Abstracting from details means to introduce new concepts or constructs and to introduce new descriptions or formalizations that condense relevant information and that reduce complexity. A principal goal of abstraction is to express a design or implementation issue in terms of abstractions that are close to what the programmer has in mind when thinking about this issue.

An abstraction of a software artifact consists of a high-level, intuitive, and useful specification that maps to a realization at a lower level; the specification describes "what" the abstraction does, whereas the realization of the abstraction describes "how" it is done [Kru92].

In our evaluation we examine the abstraction capabilities of FOP and AOP for implementing program features. Since both AOP and FOP rely on OOP, we focus only on those abstraction mechanisms that exceed the level of traditional OOP (e.g., classes, methods) and on how they differ.

4.1.2 Crosscutting Modularity

Modularity is the property of software systems that measures the extent to which they have been composed of modules. We focus exclusively on crosscutting modularity since FOP and AOP are equal with respect to modularization mechanisms known from OOP. Specifically, we use the results of the previous chapter to examine how aspects and feature modules perform in modularizing the different classes of crosscutting concerns that occur when implementing features, which are classified by our framework. That is, we evaluate how AOP and FOP perform in modularizing homogeneous and heterogeneous as well as static and dynamic crosscutting concerns.

4.1.3 Feature Cohesion

Cohesion is the ability of a feature to encapsulate all implementation details that define the feature in one unit [BK06, LHBC05]. While modularity addresses the internal structure of a feature, i.e., the modular implementation of the artifacts that implement a feature, cohesion addresses the feature as a whole, i.e., the encapsulation of all artifacts that contribute to the feature. The highest degree of cohesion is achieved by a one-to-one mapping of requirements to corresponding units at implementation level [CE00].

For example, it is easier and more intuitive to plug a cohesive data management component to a cohesive network driver in one step than to connect the data management and the network software in many places by hand.

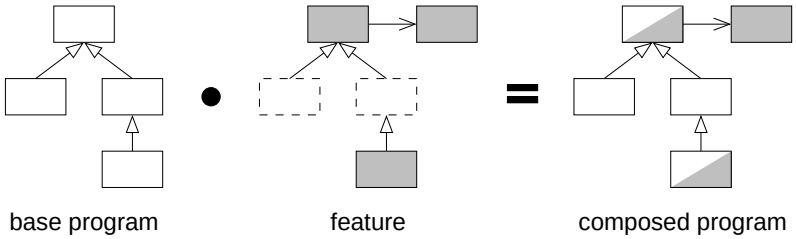


Figure 4.1: Integrating features by superimposition.

4.1.4 Feature Integration

Once a set of desired features has been selected, they are integrated to form a tailored software product. Principally, we distinguish between two types of feature integration:

Superimposition: One way to apply a feature to a program is to *superimpose* the program structure with the structure of the feature [OH92, Ern03, SB02, BSR04, OZ05, Bos99]. The concept of superimposition was first proposed for combining control structures of distributed processes [BF88, CM86, Kat93]. In terms of object-orientation, superimposition means that the class hierarchy of the program is merged with the class hierarchy of the feature [OH92], where the latter hierarchy is a sparse version of the former. The merging is applied recursively and structural elements are merged by name and type; merging classes is implemented by set union and merging methods is implemented by overriding.

Figure 4.1 illustrates the process of superimposition by example: on the left side is the class hierarchy of the base program; classes of the base program are depicted as white boxes. The program's class hierarchy is superimposed (denoted by '•') by a sparse class hierarchy of a feature; gray boxes are the classes of the feature and dashed white boxes mark the not-affected classes of the base program. On the right side the result of superimposing the structures of the base program and the feature is depicted; white boxes are the unmodified classes of the base program; gray boxes are the classes introduced by the applied feature; boxes that are half white and half gray denote the merged classes of base program and feature.

Crosscutting integration: Superimposition as feature integration technique is not always sufficient [MO02, MO03, LLO03]. Sometimes the structure of a feature does not fit the structure of the base program. This happens (1) when a feature is reused in different base programs that have different structures and (2) when a programmer wants to express a new feature in terms of abstractions that differ from those in the base program [Nov00]. For example, suppose a network software is refined by an application protocol. The protocol at application level can be expressed more easily in terms of producer (server), consumer (client), and product

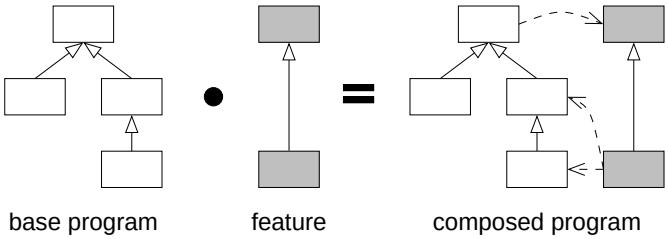


Figure 4.2: Crosscutting integration of features.

(delivered data) than by using the basic network abstractions such as sockets and streams. Since there are no structural counterparts between the two components it is complicated to achieve a clean mapping, i.e., it is not possible to superimpose their structures. Thus, such integration results in code scattering and tangling.

Figure 4.2 illustrates the process of a crosscutting integration of features: the left side shows the base program, the middle a feature, and the right side the composed program. Within the composed program the original base program and the applied feature are integrated via a set of *links* (denoted by dashed arrows) that connect the structural elements of both sides, e.g., object references, method invocations, advice, wrappers. In contrast to superimposing features, the integration pattern is cluttered. The links between base program and feature crosscut the program's as well as the feature's structure. Moreover, additional code for establishing the links is necessary.

4.1.5 Feature Composition

Features can be composed to form a new features. Technically, features are composed by superimposition or crosscutting integration. Using feature composition a programmer reuses code, which is beneficial because thinking in terms of existing features is often easier than building features from scratch. For example, constructing a data management feature out of simpler features such a storage management, query evaluation, and caching, is more efficient than constructing a tailored data management component for each use case from scratch.

4.2 Evaluation of AOP and FOP

We apply our catalog of criteria to evaluate and compare AOP and FOP.

4.2.1 Abstraction

FOP and AOP provide different abstraction mechanisms beyond those used in OOP.

FOP abstracts from composition details

FOP's feature modules encapsulate all software artifacts that implement a feature, which may be of different types. The definition of a feature module is separated from the composition specification, which enumerates the desired features by name. This hides the details of composing the internal artifacts from the programmer. The keyword `refines` denotes the composition of artifacts by set union (e.g., classes) and sequence combination (e.g., method extensions). It abstracts from a concrete implementation (e.g., *mixin-based inheritance* [BC90, SB02, BSR04], *jampack composition* [BSR04], *virtual classes* [MMP89, EOC06, OZ05], *nested inheritance* [NCM04, NQM06], *traits* [DNS+06], or *classboxes* [BDN05]).

AOP abstracts from the control flow

AOP increases the level of abstraction by introducing the concept of join points. A join point refers either to a lexical point in the static program structure or to an event in the dynamic flow of a program. This way programmers specify program extensions with respect to the dynamic program semantics [WKD04]. The programmer can think in terms of events and actions without being aware of the details that enable event handling and action triggering. For example, the pointcut `cflow` refers to the dynamic control flow of a program; it can be used to limit a set of join points to those that occur in the control flow of another join point. Of course, `cflow` can be implemented using standard OOP techniques [LHBL06] – but this obscures the programmers intention and leads to code scattering and tangling. AOP's pattern-matching and wildcard mechanisms abstract from workarounds necessary for refining each join point by a separate extension.

different abstraction mechanisms

The bottomline is that both, FOP and AOP, provide sophisticated but different mechanisms that exceed the capabilities of OOP. While feature modules abstract from details about composition and refinement, aspects provide abstractions for the control flow and for selecting and refining multiple join points.

4.2.2 Crosscutting Modularity

Homogeneous and Heterogeneous Crosscuts

A significant body of work has exposed that collaborations of classes are predominantly of a heterogeneous structure [VN96c, MO04, LLO03, Ern01, OZ05, Ost02, TVJ+01, EOC06, TOHSMS99, BSR04, SB02, Ste00, Ste05, Bos99]. That is, the roles and classes added to a program differ in their functionality, as in our graph example. A collaboration is a heterogeneous crosscut and a heterogeneous crosscut can be understood as collabo-

ration applied to a program. Hence, a feature module is well qualified to implement a heterogeneous crosscut.

In contrast to feature modules, 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. This way programmers avoid code replication. The more join points are captured by a homogeneous crosscut, the higher the pay-off of AOP.

Although both approaches support the implementation of the crosscuts the other approach focuses on, they cannot do so elegantly [MO04].

```
1  refines class Node {
2    Color color;
3    Color getColor() { return color; }
4    void setColor(Color c) { color = c; }
5    void print() {
6      Color.changeDisplayColor(getColor());
7    }
8  }
9  refines class Edge {
10   Color color;
11   Color getColor() { return color; }
12   void setColor(Color c) { color = c; }
13   void print() {
14     Color.changeDisplayColor(getColor());
15   }
16 }
17 class Color { /* ... */ }
```

Figure 4.3: Implementing the *Color* feature as a feature module.

using collaborations instead of aspects

Implementing our *Color* feature (a homogeneous crosscut) using FOP we would introduce two refinements to the classes `Node` and `Edge`, which introduce exactly the same code (Fig. 4.3). Our AOP-based solution proposed previously avoids this code replication (Fig. 4.4).

using aspects instead of collaborations

Conversely, an aspect may implement a collaboration (a heterogeneous crosscut) by bundling a set of inter-type declarations and advice, as shown in Figure 4.5. The aspect `AddWeight` introduces the method `add` and the field `weight` via inter-type declarations (Fig 4.6, Lines 2-5; Line 6) and extends the `print` method via advice (Lines 7-11). Hence, it implements a heterogeneous crosscut, which is a collaboration of `Weight` and two roles, a role of `Edge` and a role of `Graph`. We have noticed and so have others [Ste05, MO04, Bos99] that not expressing a collaboration in terms of object-oriented design (i.e., roles implemented as refinements) decreases program comprehensibility. This is because programmers cannot recognize the original structure of the base program within a subsequent refinement – in our example the structuring in `Graph`, `Node`, and `Edge`.

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

Figure 4.4: Implementing the *Color* feature as an aspect.

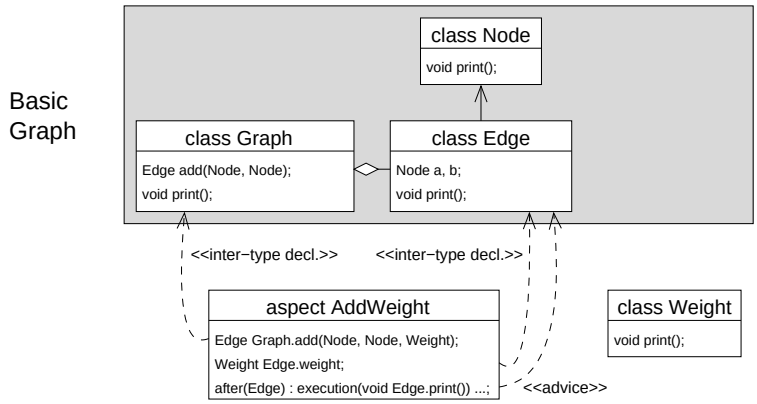


Figure 4.5: Implementing a collaboration as an aspect.

large-scale features

One may argue that, for this simple example, it does not really matter whether one uses feature modules or aspects. But the difference between FOP and AOP becomes more obvious when considering features at a larger scale. Then it become clear that aspects lack scalability. Suppose a base program consists of many classes and a feature extends most of them. In a FOP solution the programmer defines, per class to be extended, a new role with the same name (Fig. 4.7). This way the programmer is able to retrieve the program structure within the new feature. There is a one-to-one mapping between the structural elements of the base program and the elements of the feature; base program and feature are merged recursively by name and type.

In an AOP solution one would merge all participating roles into one (or more) aspect(s) (Fig. 4.8). While this is possible, it flattens the inherent object-oriented structure of the feature and makes it hard to trace the mapping between base program and feature [Ste05, MO04]. Note that the difference between AOP and FOP, as shown in the Figures 4.7 and 4.8, is not only a matter of visualization. The point is that the inner structure of the aspect does not reflect the structure of the base program; there is no natural mapping between structural elements of the base program and the feature. So it is no coincidence

```
1  aspect AddWeight {
2    Edge Graph.add(Node n, Node m, Weight w) {
3      Edge res = add(n, m);
4      res.weight = w; return res;
5    }
6    Weight Edge.weight;
7    after(Edge e) : this(e) && execution(void Edge.print()) {
8      System.out.print(" [");
9      e.weight.print();
10     System.out.print("] ");
11   }
12 }
```

Figure 4.6: An AspectJ aspect that implements a collaboration.

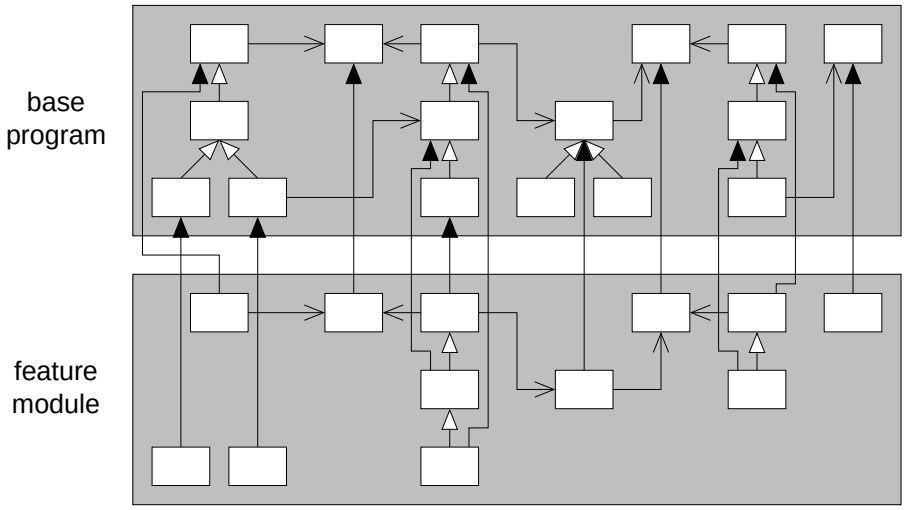


Figure 4.7: Implementing a large-scale feature using a feature module.

that the mapping is complicated and hard to trace for the programmer. The one-to-one mapping of the FOP solution is easier to understand especially for large-scale features.

roles and aspects

Implementing each role as a distinct aspect, as suggested by Hanenberg et al. [HU02], Kendall [Ken99], and Sihman et al. [SK03], would obscure the object-oriented structure as well. In our example we would implement the refinements of `Graph` and `Edge` as two distinct aspects. This approach would enable to establish a one-to-one mapping between the structural elements of the base program and the elements of the feature (provided reasonable naming conventions). However, this way inheritance and refinement is replaced simply by aspect weaving without any further benefit. We and others [Ste05, MO04] argue that such a replacement of object-oriented techniques without any benefit is questionable, especially with respect to the additional complexity introduced by aspect weaving [Ste06, Ale03].

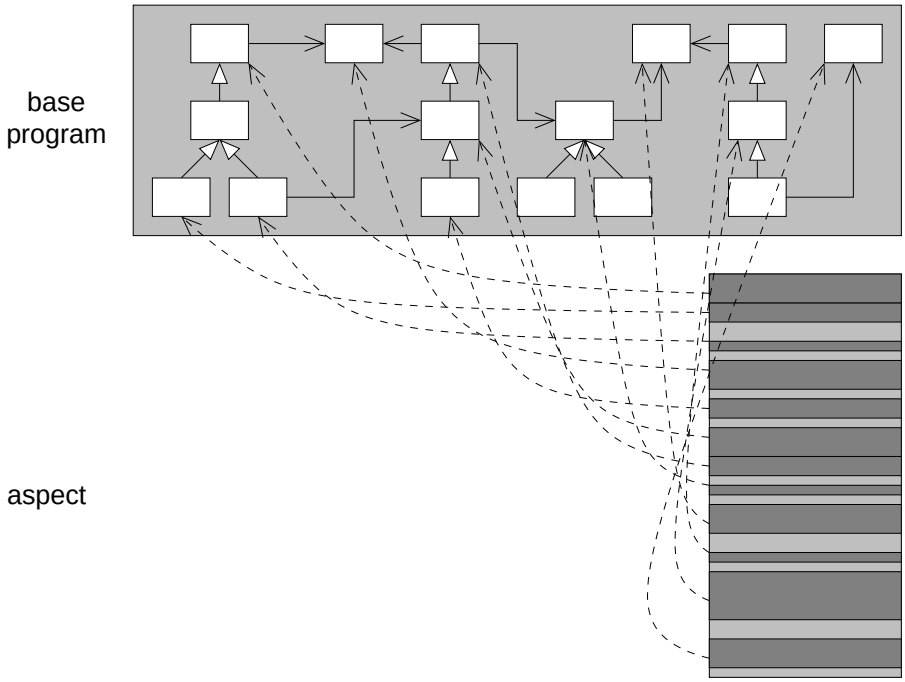


Figure 4.8: Implementing a large-scale feature using an aspect.

The reason why aspects fail in expressing collaborations appropriately is that roles are closely connected to their classes; role-based design is inherently object-oriented [Ste00].

Static and Dynamic Crosscuts

static crosscuts

Features and aspects may extend the structure of a base program statically, i.e., by injecting new members and introducing new superclasses and interfaces to existing classes. Figure 4.9 depicts a refinement and an aspect, which both inject a method and a field as well as introduce a new interface to the class `Edge`.

```
1 refines class Edge
2   implements Comparable {
3     boolean compare(Edge e) {
4       /* ... */
5     }
6 }
```

```
1 aspect ComparableEdge {
2   declare parents: Edge implements Comparable;
3   boolean Edge.compare(Edge e) {
4     /* ... */
5   }
6 }
```

Figure 4.9: Implementing a static crosscut via refinement (left) and via aspect (right).

Additionally, features are able to encapsulate and introduce new classes. Traditional aspects, as exemplified by AspectJ, are not able to introduce independent classes – at least not as part of an encapsulated feature. While it is correct that one can just add

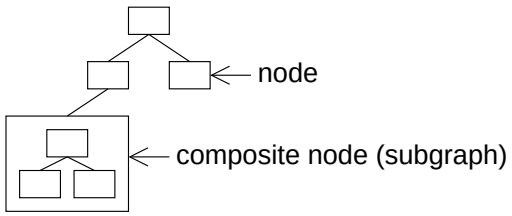


Figure 4.10: A recursive graph data structure.

another class to an environment, e.g., using AspectJ, this is at the tool level and not at a model level. The programmer has to build his own mechanisms (outside of the tool) to implement feature modularity [LHBC05], e.g., in *FACET*, an AspectJ implementation of a CORBA event channel, the programmers implemented a non-trivial mechanism for feature management [HC02].

dynamic crosscuts

As opposed to AOP, FOP provides no extra language support for implementing dynamic crosscuts. That is, dynamic crosscuts can be implemented but there are no tailored abstraction mechanisms to express them in a more intuitive way, e.g., by an event-condition-action pattern. Without depending on a workaround, FOP supports just basic dynamic crosscuts, i.e., method extensions [MO04]. While this works for many implementation problems, there are certain situations in which a programmer may want to express a new program feature in terms of the dynamic semantics of the base program, i.e., to implement an advanced dynamic crosscut (cf. Sec. 3.2). Aspects are intended exactly for this kind of crosscut. They provide a sophisticated set of mechanisms to refine a base program based upon its execution, e.g., mechanisms for tracing the dynamic control flow and for accessing the runtime context of join points.

extending recursive data structures demands aspects

When extending the printing mechanism of our graph implementation, we can take advantage of these sophisticated mechanisms of AOP. Background is that the `print` methods of the participants of the graph implementation call each other, – especially, when nodes of a graph may be (sub)graphs themselves (Fig. 4.10).

Generally, recursive data structures are an appropriate use case for AOP. The AOP language constructs for advanced dynamic crosscuts (e.g., `cflow`, `cflowbelow`) enable to advise only selected join points within the control flow of a program. For example, 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 condition evaluated at runtime (Fig. 4.11). The advice shown is an example of an advanced dynamic crosscut.

```
1 aspect PrintHeader {
2   before() : execution(void *.print()) && !cflowbelow(execution(void *.print())) {
3     printHeader();
4   }
5   void printHeader() {
6     System.out.print("header: ");
7   }
8 }
```

Figure 4.11: Advising the printing mechanism using advanced advice.

using FOP for advanced dynamic crosscuts

Though language abstractions such as `cflow` and `cflowbelow` can be implemented (emulated) by FOP, this usually results in code replication, tangling, and scattering. For example, Figure 4.12 depicts the above extension to the printing mechanism implemented using FOP. Omitting AOP constructs results in a complicated workaround (underlined) for tracing the control flow (Lines 2,6,8) and executing the actual extension conditionally (Lines 4-5). Compared to the FOP solution, the AOP solution captures the intension of the programmer more precisely and explicitly (cf. Fig. 4.11).

```
1  refines class Node {
2    static int count = 0;
3    void print() {
4      if(count == 0)
5        printHeader();
6      count++;
7      Super.print();
8      count--;
9    }
10   void printHeader() { /* ... */ }
11 }
```

Figure 4.12: Implementing the extended printing mechanism via refinement.

The bottomline is that FOP and AOP are complementary with respect to crosscutting modularity. FOP is strong in modularizing collaborations, which are heterogeneous and basic dynamic crosscuts. AOP performs well in modularizing homogeneous and advanced dynamic crosscuts.

4.2.3 Feature Cohesion

Features implemented via feature modules have an explicit representation at the design and implementation level. All structural elements that contribute to the feature are encapsulated inside the feature module. Hence, a high degree of feature cohesion is achieved.

Using AOP, a programmer expresses new features by aspects, but in many cases features cannot be expressed using one single aspect, especially not in complex programs [LHBC05, MO04]. Often the programmer introduces several aspects and additional classes, e.g., the *Weight* feature consists of the aspect `AddWeight` and the class `Weight`. One may argue that we could express every feature using only one aspect, but this violates the principle of separation of concerns – it destroys the inner structure of a feature's implementation, as explained in Section 4.2.2. Classes and aspects are too small units of modularity and therefore not suitable for implementing features [VN96c, MO04, LLO03, Ern01, OZ05, Ost02, TVJ+01, EOC06, TOHSMS99, BSR04, SB02, Ste00, Ste05, Bos99].

Nevertheless, aspects can be encapsulated in packages or may contain nested classes but there are no mechanisms for refining and composing these constructs. However, hybrid approaches like *Caesar* [MO02, MO03, MO04] exploit the mechanisms of collaboration-based designs such as mixin composition and virtual classes.

In summary, feature modules provide appropriate means for the cohesive implementation of program features. The reason for that is they encapsulate collaborations of artifacts and they can be composed. An aspect should not implement an entire feature because in traditional AOP it is a class-like entity that cannot express a collaboration. What follows is that aspects can be a part of a feature implementation, as we will show in Chapter 5.

4.2.4 Feature Integration

When applying a feature to a program, an FOP compiler superimposes the structure of the feature module with the structure of the base program. Superimposition is implemented by merging recursively the hierarchical structures of feature modules by name and type (mixin composition) [BSR04, OZ05, AGMO06, Ern03].

If a feature module is of a different structure than the base program, the code for integrating the feature and the base program has to be implemented by hand. Usually, this results in code scattering and tangling [MO04, LLO03]. The bottomline is that FOP does not support crosscutting integration very well.

It has been shown that aspects, in collaboration with other mechanisms such as wrappers, can help in integrating structurally independent components, i.e., features that differ in their inner structure [MO02, MO03, LLO03]. Pointcuts and advice are hereby used to modularize the crosscutting integration code, which would otherwise lead to code tangling, scattering, and replication. Hence, aspects facilitate a well modularized crosscutting integration of features.

But AOP does not support superimposition. Indeed, aspects may implement roles or even entire collaborations but they always have to specify explicitly where the base

program is to be modified. There is no matching by name, type, and/or structure, as advocated in [SB02, BSR04, TOHSMS99, OZ05, AGMO06, Ern03].

The bottomline is that FOP is appropriate for superimposition and AOP for crosscutting integration of features.

4.2.5 Feature Composition

Feature modules can be composed to form new features modules. This enables the programmer to generate compound features out of basic ones. A feature module is implemented as a containment hierarchy, which can be nested hierarchically. The algebraic theory behind FOP models a feature as a function; applying a feature to a program is modeled as function application and composing features as function composition.

It has been observed that composing aspects is non-trivial or even impossible [LHBC05, LHBL06]. While aspects can be applied to a program individually, two aspects cannot be composed to form a new aspect. The composition is further complicated by the hard-to-understand precedence rules for ordering the application of aspects.

4.3 Summary, Perspective, and Goals

Table 4.1 summarizes the results of our conceptual evaluation. It reveals that both programming paradigms complement one another. That is, both have strengths where the respective other is weak. For example, while FOP is sufficient to encapsulate collaborations, which are heterogeneous crosscuts, AOP suffices in expressing homogeneous crosscuts, thus avoiding code replication. Furthermore, AOP is strong in abstracting the dynamic control flow and FOP in abstracting the composition of features. The benefits of using both AOP and FOP together offer rewards that neither of them could accomplish in isolation.

symbiosis of FOP & AOP

A clever symbiosis of both paradigms might replace the weaknesses of one paradigm with the strengths of the other. However, an unfavorable symbiosis might lead to even worse results. The following chapters address this issue in greater depth.

A further crucial issue that arises from the symbiosis proposed is to what extent the individual mechanisms of AOP and FOP are really needed. In this dissertation we discuss first results of analyzing a series of case studies to address this issue.

evaluation criteria		FOP	AOP
abstraction		**good support**: FOP provides abstraction mechanisms and tool support for feature composition and program refinement	**good support**: AOP has an event-based model and abstracts from details of refining multiple join points
crosscutting modularity	heterogeneous crosscuts	**good support**: feature modules encapsulate and compose collaborations of classes and refinements	**limited support**: aspects bundle sets of inter-type declarations and advice, but lack of abstracting and expressing collaborations
	homogeneous crosscuts	**no support**: feature modules provides no explicit language constructs for refining multiple join points simultaneously	**good support**: aspects provide wildcards and pattern matching mechanisms to refine multiple join points simultaneously
	static crosscuts	**good support**: feature modules can inject new fields, methods, and classes as well as declare new superclasses/interfaces	**limited support**: aspects can inject new fields and methods – but no classes – as well as declare new superclasses/interfaces
	dynamic crosscuts	**weak support**: feature modules can implement only basic dynamic crosscuts via overriding (method extensions); there is no support for advanced dynamic crosscuts	**good support**: aspects provide sophisticated mechanisms for advising a program based on its dynamic semantics (basic and advanced dynamic crosscuts)
feature cohesion		**high degree**: feature modules encapsulate all artifacts that contribute to a feature	**low degree**: aspects cannot encapsulate collaborations of multiple artifacts that contribute to a feature
feature integration	superimposition	**good support**: FOP provides explicit support for superimposition – merging hierarchical structures recursively by name and type	**no support**: AOP does not provide any mechanisms for superimposing hierarchical structures of software artifacts
	crosscutting integration	**no support**: no mechanisms for expressing and modularizing crosscutting integration code	**good support**: aspects can connect feature implementations and modularize the integration code
feature composition		**good support**: feature modules can be composed to form new features; this is modeled by function composition	**no support**: aspects cannot be composed; difficult composition rules

Table 4.1: A comparison of FOP and AOP.

CHAPTER 5

The Symbiosis of Feature Modules and Aspects

This chapter shares material with the ICSE'06 paper 'Aspectual Mixin Layers: Aspects and Features in Concert' [ALS06] and the GPCE'05 paper 'FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming' [ALRS05].

In this chapter we address the following issues: (1) how to combine FOP and AOP and (2) does their combination outperform FOP and AOP in isolation?

First, we explore the space for achieving the symbiosis of FOP and AOP. Then, we present our approach for integrating feature modules and aspects, which we call *aspectual feature modules (AFMs)*. Finally, we present our attempts to provide adequate tool support and discuss related approaches.

5.1 Design Space

FOP and AOP can be combined principally in two ways: (1) design a programming language that combines the mechanisms of AOP and FOP, which we call an *in-language approach*, and (2) integrate aspects as software artifacts into the development style of FOP and SWD, which we call an *architectural approach*.

in-language approach

The in-language approach enables to explore the language properties of FOP and AOP as well as their possible integration. As our evaluation will reveal, some language mechanisms of AOP and FOP are redundant. It is an interesting research question what a novel language should look like that integrates AOP and FOP, but in an aggregated and stripped-down form. To put it in other words, with the in-language approach we can explore the minimal core language for implementing features, true to the motto: what

is not essential will be omitted. Moreover, it would be possible to address advanced language level issues such as type systems and soundness.

architectural approach

The architectural approach is a software engineering approach. It takes into account that FOP is also a design method to develop SPLs in a SWD manner. AHEAD as architectural model comprises all kinds of software artifacts and lays an algebraic foundation for features and SWD. In this sense, aspects are just a new software artifact that should be integrated into the architectural model as well, however, with special characteristics and individual support at the language level. Choosing this approach would combine the implementation mechanisms of AOP and FOP. We could explore the relationship of feature modules and aspects with respect to the implementation of the large-scale building blocks of SPLs and their impact on software design.

While both approaches promise interesting insights, we can choose only one in order not to exceed the scope of this dissertation. Since we aim at SPLs and SWD it is reasonable to explore the architectural approach first. Though we address one or the other language level issue (e.g., in Chapter 6), an in-depth analysis of what a minimal and efficient FOP/AOP language would look like is relegated to future work.

5.2 The Integration of Feature Modules and Aspects

Since AHEAD provides an architectural model for FOP, we describe our integration of FOP and AOP on top of the AHEAD model.

feature modules decompose object-oriented designs

When designing and implementing SPLs in a feature-oriented way, a programmer starts usually by modeling and abstracting real-world entities in terms of classes and objects and their collaborations. The result is an object-oriented design (left side of Fig. 5.1). FOP further structures this design along collaborations that classes undergo. Only the subsets of classes (roles) that participate in a collaboration to implement a certain feature are encapsulated inside the corresponding feature module, i.e., features crosscut the object-oriented design (right side of Fig. 5.1). Subsequent features refine existing features by superimposing their structure (collaborations) [OH92, BSR04, SB02, Bos99, OZ05, Ern03]. Hence, a feature module is a mechanism that decomposes an object-oriented design along a further dimension, i.e., the features of a program.

feature modules lack crosscutting modularity

Our evaluation pointed us to the fact that in some situations the implementation of a feature cannot be modularized appropriately by using a traditional feature module implemented for instance in Jak, i.e., attempts to do so result in code replication, and code scattering and tangling. Typically, these situations are related to crosscutting phenomena. We argue that the shortcomings of FOP revealed by our evaluation are directly responsible for this issue.

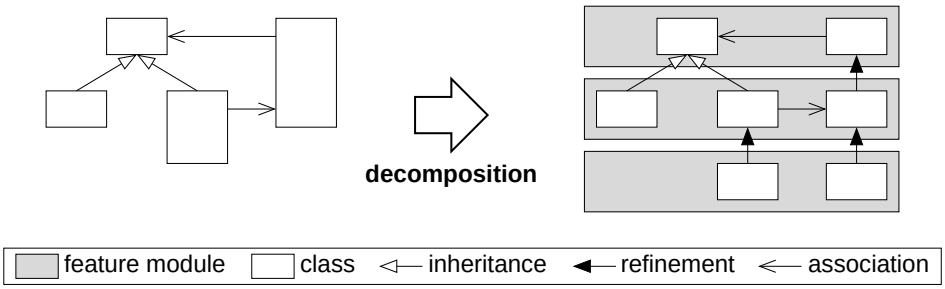


Figure 5.1: Feature-driven decomposition of an object-oriented design.

integrating aspects to improve crosscutting modularity

To address this issue, we propose to employ AOP since it provides powerful mechanisms to modularize crosscutting concerns. Nevertheless, as our evaluation revealed, simply using aspects instead of feature modules for implementing program features is not appropriate either, e.g., because of the lack of feature cohesion and the missing abstraction mechanisms for collaborations. Instead, we propose to use aspects to implement only concerns that crosscut a given object-oriented design and that cannot be modularized well using feature modules, thus creating an aspect-oriented design, i.e., a hierarchy of classes and aspects (left side of Fig. 5.2).

decomposing aspect-oriented designs

In order not to forgo the benefits of feature modules, we suggest further to decompose such aspect-oriented design using the mechanisms of FOP: *While the aspect-oriented design serves as a substructure, feature modules decompose this design further, along the features of the program.*. Hence, a feature is implemented by a collaboration of classes *and* aspects (right side of Fig. 5.2)[1]. Benefit of this integration is that we have well encapsulated large-scale feature modules that refine one another incrementally and that dispose of powerful mechanisms for dealing with crosscutting phenomena.

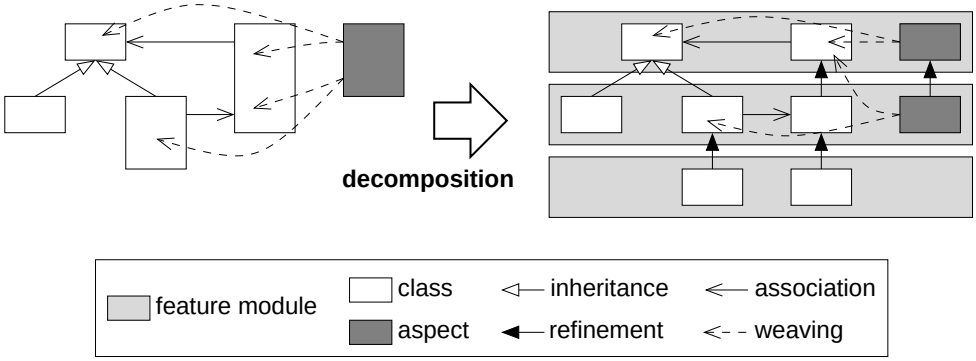


Figure 5.2: Feature-driven decomposition of an aspect-oriented design.

[1] Note that the original aspect has been split into two pieces (a base and a subsequent refinement). In Chapter 6, we address this issue in more depth.

aspects and feature modules do not compete

In summary, aspects and feature modules are not competing implementation techniques but decompose a program in different ways. That is, from our perspective a program design is decomposed along three dimensions: classes, aspects, and features. An object-oriented design is the basis; aspects modularize certain kinds of concerns that crosscut the underlying object-oriented design; feature modules decompose the design to impose a structure that is of interest to stakeholders, i.e., the features of the program. In this symbiosis, FOP and AOP profit from each other and overcome their individual limitations, as we will illustrate in this dissertation.

5.3 Aspectual Feature Modules

Aspectual feature modules (AFMs) is a concrete approach to implement the integration of AOP and FOP. AFMs extend the notion of a traditional feature module known from Jak by encapsulating, beside classes and refinements of classes, also aspects. That is, an AFM encapsulates the roles of collaborating classes *and* aspects that contribute to a feature. Hence, a feature is implemented by a collection of artifacts, among them classes, refinements, and aspects. We argue that this is close to the ideal of what a feature should be. Thus, a feature is implemented by different kinds of artifacts, each artifact appropriate for a specific design or implementation problem.

Figure 5.3 shows a base program (light gray box above) refined by an AFM (light gray box below). The AFM refines the base program in two ways: (1) it contains a class refinement and (2) an aspect (dark gray box) to implement the changes to be made to the base program. Our implementation of AFMs relies on *mixin layers* [SB02] and *AHEAD refinements* [BSR04]. Other mechanisms such as *virtual classes* [MMP89, EOC06, OZ05], *nested inheritance* [NCM04, NQM06], and *traits* [DNS+06] would be possible (see Sec. 5.6).

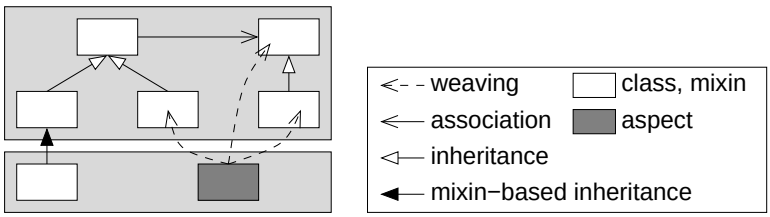


Figure 5.3: Aspectual feature modules.

two ways of refinement

An AFM can refine a base program in two ways: (1) by using mixin-composition or (2) by using aspect-oriented mechanisms, i.e., advice and inter-type declarations. Probably the most important contribution of AFMs is that programmers may choose the appropriate technique – refinements or aspects – that fits a given problem best. Moreover, they can

apply a combination of both and decide to what extent either technique is used. The questions that arise consequentially are (1) when to use AOP and FOP mechanisms and (2) how often their application really occurs in real-world software projects. We elaborate on this in more depth in Chapter 7 and Chapter 8.

Figure 5.4 depicts the feature-oriented design of our graph implementation, consisting of the features *BasicGraph*, *Weight*, and *Color*. *Color* is implemented by using an aspect and a class; it is encapsulated by an AFM. As we discussed before, advising executions of the methods `print` in `Node` and `Edge` is a homogeneous crosscut – the same is true for injecting the field `color` and the methods `setColor` and `getColor` to `Node` and `Edge` (cf. Fig. 4.4). In this situation, it is beneficial to use an aspect because it is able to avoid replicated code. Encapsulating the aspect `AddColor` and the class `Color` improves feature cohesion, compared to a pure AOP variant.

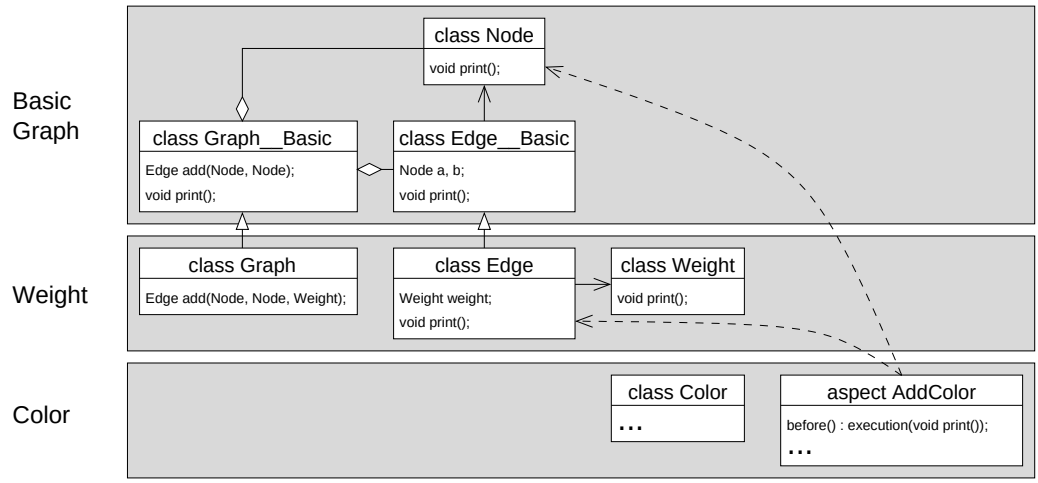


Figure 5.4: Implementing the feature *Color* as an aspectual feature module.

superimposing containment hierarchies

As with standard feature modules, an AFM is represented as a containment hierarchy. Besides Java or C++ artifacts an AFM contains also aspect files. Figure 5.5 depicts the simplified containment hierarchies of our graph features *BasicGraph*, *Weight*, and *Color*. The containment hierarchy synthesized finally is generated by superimposing the three feature hierarchies. The composition order is specified via a feature expression. During the composition the programmer needs not to be aware of which kinds of software artifacts actually are inside the features to be composed. This helps to concentrate on the composition process at the feature level and facilitates compositional reasoning because implementation details are hidden. That is, the programmer needs not to know which files and types of artifacts contribute to a feature implementation.

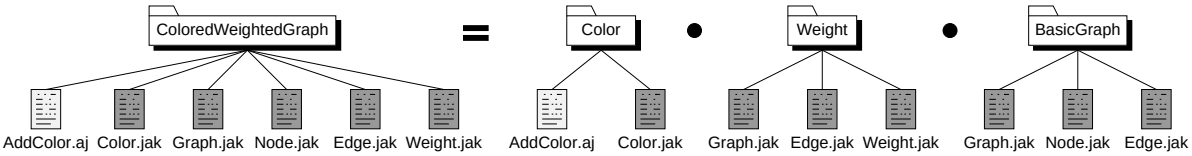


Figure 5.5: Superimposing containment hierarchies including aspects.

mixin and jampack composition

The result of superimposing containment hierarchies is a program, i.e., a set of collaborating software artifacts. Batory et al. propose two principal ways of implementing the actual composition [BSR04]. Figure 5.6 depicts a possible hierarchy of classes and aspects synthesized by the above feature selection. Classes and their refinements are merged into composite classes, which is called *jampack composition*; Figure 5.7 depicts the same program synthesized by mixin composition, which translates refinement to subclassing, i.e., a base class and a series of refinements is translated to a base class and a series of subclasses. Note that, beside these two solutions, also alternative mechanisms such as virtual classes, nested inheritance, and traits could implement (emulate) refinements of classes.

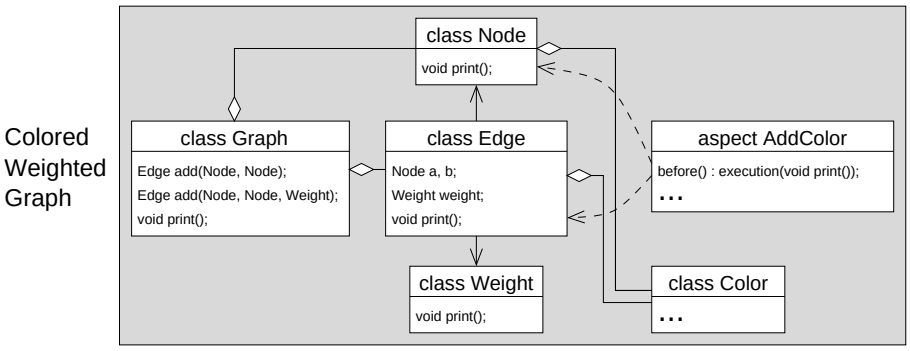


Figure 5.6: Jampack-composed graph implementation.

two-staged composition of AFMs

Either way, after the composition process we have in case of AFMs a traditional aspect-oriented program (and in case of traditional feature modules an object-oriented program). Now it becomes clear that it is necessary to weave the aspects and the object-oriented base program in a subsequent step – after the base classes and refinements have been composed. These two steps can be accomplished by different compiler passes or by different tools.

AFMs are language independent

AFMs integrate feature modules and aspects. The AHEAD architectural model is the basis for the integration. Thus, AFMs are independent of a specific host language. They can be implemented in any pair of object-oriented and aspect-oriented language which

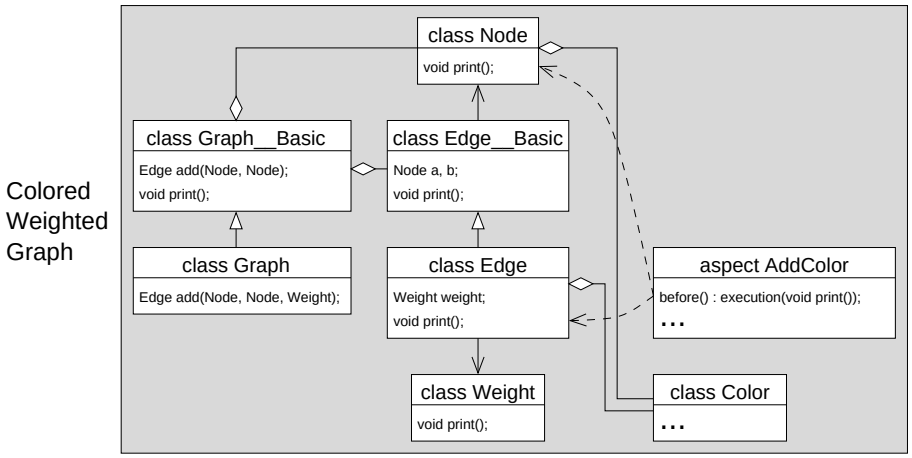


Figure 5.7: Mixin-composed graph implementation.

can be woven, e.g., Java and AspectJ, C++ and *AspectC++*[2], or C# and *AspectC#*[3], etc. This circumstance makes the concept of AFMs invariant to the specifics of the host languages. When the host languages improve, especially the AOP languages, then AFMs improve as well. Thus, AFMs can profit from research in AOP and FOP. With an in-language approach that would not be possible or only with a major effort.

5.4 A Conceptual Evaluation of Aspectual Feature Modules

To evaluate AFMs and to compare them to traditional FOP and AOP we apply our evaluation criteria.

5.4.1 Abstraction

AFMs support the combined abstraction mechanisms of feature modules and aspects. On the one hand, AFMs are treated as regular feature modules; a programmer composes AFMs by enumerating the feature names without needing to know the internal implementation details. The keyword `refines` abstracts from the composition semantics, i.e., mixin composition, jampack composition, and others[4]. On the other hand, AFMs contain aspects and thus build upon the advanced capabilities of AOP to implement a program refinement in dependence of the runtime control flow. Features can be

[2] http://www.aspectc.org/
[3] http://www.dsg.cs.tcd.ie/dynamic/?category_id=169
[4] Chapter 6 explains how this maps to aspects in order to compose and refine them as well.

implemented on top of the event model of AOP. The wildcard and pattern matching mechanisms of AOP avoid code replication in case of homogeneous crosscuts.

The integration of feature modules and aspects leads to a broader arsenal of abstraction mechanisms available when implementing and composing features – it unifies the strengths of FOP and AOP.

5.4.2 Crosscutting Modularity

Homogeneous and Heterogeneous Crosscuts

The integration of aspects and the traditional constituents of feature modules enables the programmer to choose the right technique for solving the right problem: the programmer uses aspects to implement homogeneous crosscuts and a set of classes and refinements to implement heterogeneous crosscuts, which are in fact collaborations. As mentioned, this is independent of whether a crosscut is static or dynamic. Aspects, classes, and refinements can be combined at will.

Static and Dynamic Crosscuts

The integration of FOP and AOP allows us to express static crosscuts in two ways, using refinements of classes and using inter-type declarations in aspects. This introduces a semantic redundancy. As mentioned in the previous paragraph, we propose to use aspects to implement static crosscuts that are homogeneous and to use traditional feature modules to implement static crosscuts that are heterogeneous.

By using aspects, a programmer can implement features depending on the runtime control flow. As with static crosscuts, method extensions can be implemented by aspects (using advice) and by refinements (using method overriding). We handle this analogously to static crosscuts: use aspects for dynamic crosscuts that are homogeneous and use feature modules and method extensions for basic dynamic crosscuts that are heterogeneous. Advanced dynamic crosscuts are always implemented using advice because FOP does not supports adequate language mechanisms.

An observance of these guidelines improves the crosscutting modularity of AFMs compared to traditional feature modules without destroying the object-oriented structure per se.

5.4.3 Feature Cohesion

Since we encapsulate aspects in feature modules, we achieve a high degree of feature cohesion. Aspects as well as collaborating classes (e.g., aspect `AddColor` and class `Color`) are located in one feature module along with other software artifacts that contribute to the implementation of the feature (e.g., documentation, makefiles, test cases). Despite their encapsulation in feature modules, aspects still crosscut module boundaries, but this is intended to be able to modularize certain kinds of crosscuts. We are aware that this property is controversial [Ste06, Ale03], but our approach does not rely on a specific AOP mechanism and will profit by improvements to AOP, e.g., *open modules* [Ald05, OAT+06], *information hiding interfaces* [SGS+05], *stratified aspects* [BFS06], or *harmless advice* [DW06]. What is novel is that the programmer is able to recognize explicitly which artifacts belong to a feature, not only at the file system or tool level but also at the model level.

5.4.4 Feature Integration

The structures of feature modules are superimposed during composition. While this is appropriate for many integration problems [BSR04, SB02, Ern03, Bos99, OH92, OZ05], superimposition is not always sufficient [MO02, MO03, TOHSMS99]. Using traditional feature modules in the form of collaborations the integration of non-related, structural differing features results in workarounds, code scattering, and code tangling [MO03, LLO03, Her02]. This is because of their manifold dependencies and interactions [Nov00]. But it has been shown that aspects in concert with wrappers can modularize such integration code [MO03]. AFMs support superimposition and can employ aspects for crosscutting integration, thus outperforming FOP and AOP used in isolation.

5.4.5 Feature Composition

AFMs can be aggregated to form new AFMs. At implementation level this is accomplished by nesting containment hierarchies. At model level it is described by function composition. Thus, aspects as software artifacts become nested in feature hierarchies. With traditional AOP such mechanisms have to be implemented by hand, which is non-trivial, e.g., as done in the FACET project [HC02]. However, the problem of composing aspects to form new aspects is not solved.

5.5 Tool Support

We provide tool support for AFMs on top of two host programming languages, C++ and Java.

5.5.1 FeatureC++

FeatureC++[5] was developed by the author within the scope of this dissertation. It is a language extension of C++ that supports FOP. It consists of a tool for composing feature modules and an FOP compiler for C++ artifacts. Specifically, it introduces class refinement to the C++ language in the form of the syntax presented here, i.e., the keywords `refines` and `Super` – with some minor adaptations to the C++ standard. FeatureC++ supports AFMs by integrating *AspectC++* [SLU05] aspects into feature modules. Furthermore, it supports the AHEAD algebraic expressions and design rule checks for compositional reasoning [BSR04].

FeatureC++ feature modules

Figure 5.8 depicts a template class `List` (Lines 1-6) and a refinement (Lines 7-13). The class `List` receives the type of the items being stored. The refinement adds a new variable `size` (Line 11) and extends the method `put` (Line 12) to increment the size. Note that the refinement extends also the type argument list. Given this refinement, the programmer specifies the type of the items and the type of the size counter. This kind of refinement is called *generic refinement* and it is embedded in a *generic feature module*. Generic feature modules can be parameterized statically using the powerful template mechanism of C++. A deeper explanation of generic feature modules is out of scope of this dissertation and described elsewhere [AKL06].

```
1  template <typename _ItemT>
2  class List {
3    typedef _ItemT ItemT;
4    ItemT *head;
5    void put(ItemT *i) { i->next = head; head = i; }
6  };
7  template <typename _ItemT, typename _SizeT>
8  refines class List {
9    typedef _ItemT ItemT;
10   typedef _SizeT SizeT;
11   SizeT size;
12   void put(ItemT *i) { super::put(i); size++; }
13 };
```

Figure 5.8: A FeatureC++ code example.

[5] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

compilation process

Figure 5.9 depicts the process of compiling AFMs using FeatureC++. The compiler receives FeatureC++ code as input and transforms it to native C++ code and to AspectC++ code. The transformation is done on top of abstract syntax trees. The FeatureC++ parser uses the *PUMA*[6] framework. In a second step PUMA is used to weave the AspectC++ aspects and the native C++ code. Finally, the woven C++ code is compiled to produce binaries. A deeper description of the FeatureC++ compiler is out of scope of this dissertation and is published elsewhere [ALRS05, AKL06].

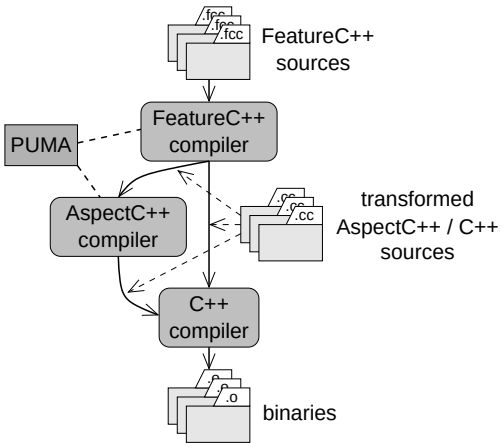


Figure 5.9: FeatureC++ compilation process.

5.5.2 AHEAD Tool Suite & AspectJ

A further way to implement AFMs is to combine the AHEAD Tool Suite (ATS) and AspectJ. While Jak is used to compose traditional feature modules, AspectJ weaves the aspects of the individual feature modules to the synthesized class hierarchies. The examples given in this dissertation are written in this way. This necessitates some minor tool support and modifications to the aspect code. For example, a build script needs to keep track of the aspects included in the selected feature modules and to weave them in a subsequent step. Also the programmer has to be aware of the fact that the target classes of an aspect are renamed during the compilation process, e.g., class `List` of feature *BasicList* is renamed to class `List__BasicList`. In FeatureC++ this is handled automatically. While a further explanation of the technical problems is out of scope of this dissertation, we refer to the successful application of this approach to a non-trivial software project (see Chapter 7).

The process of compiling AFMs using the ATS and AspectJ is similar to the one of FeatureC++ (cf. Fig. 5.9). It is worth to note that we were able to integrate two tools

[6] http://ivs.cs.uni-magdeburg.de/~puma/

(ATS and AspectJ) to achieve an appropriate support for AFMs. This demonstrates that AFMs indeed are a language-independent approach that is realizable with current tools. Of course, in contrast to the integrated solution of FeatureC++, some workaround is necessary (build scripts, minor code adjustments).

5.5.3 FeatureIDE

Supporting feature-oriented software development across the entire software life cycle is the aim of a parallel dissertation project. It provides a tool *FeatureIDE* that is an integrated development environment for *feature-oriented domain analysis (FODA)* [KCH+90], FOP, as well as the subsequent configuration. *FeatureIDE* was developed (and is still under development) in cooperation with the author of this dissertation [LAMS05]. It supports AFMs based on FeatureC++ as well as AHEAD & AspectJ. Figure 5.10 depicts a FeatureIDE screen snapshot that shows the FODA features model of our graph example. It contains the features *BasicGraph*, *Weight*, *Color*, and *PrintHeader*, all visualized as boxes. The feature model is created in a drag-and-drop way like in a drawing program.

Figure 5.11 depicts a stack of feature modules generated automatically from the feature diagram. It consists of the feature modules *BasicGraph*, *Weight*, and *PrintHeader*. The generation process creates the underlying file system structure for the containment hierarchies of the feature modules. Feature modules are visualized as gray boxes and artifacts within a module as white boxes. In our example, *PrintHeader* is an AFM and contains an aspect.

5.6 Related Work

Implementation of Refinements

Our approach of implementing class refinement is based on mixins and mixin-based inheritance [BC90, VN96b]. We chose mixins because of their success in several domains [CBML02, BZM01, BJMvH02, BCGS95, VN96c, AB04, LAS05]. However, we are aware of several alternative mechanisms that might achieve similar results (we discuss only a representative selection).

Traits aim at structuring object-oriented programs [DNS+06]. Traits are stateless units of code reuse that group multiple methods, but not state-holding members. Multiple traits can be combined using *glues* in order to synthesize a final class. Traits offer customizability at a more fine-grained level than mixins. Traits could be used to implement

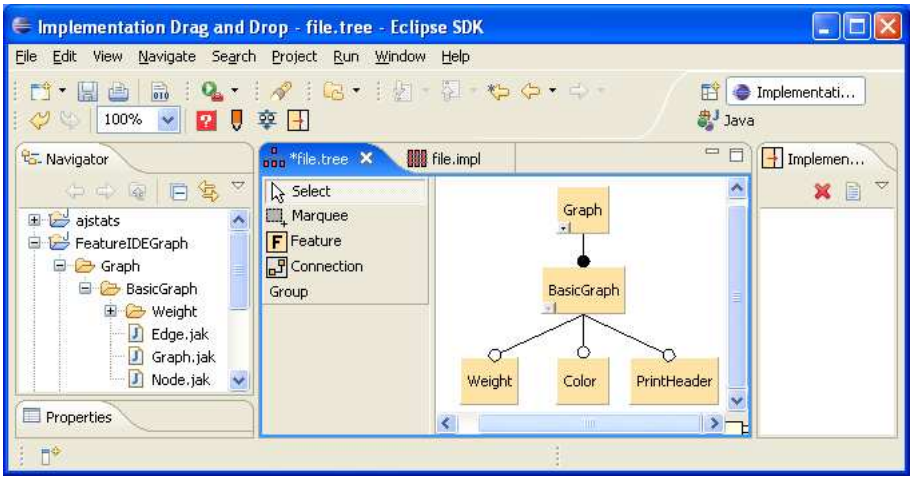


Figure 5.10: Feature modeling in FeatureIDE.

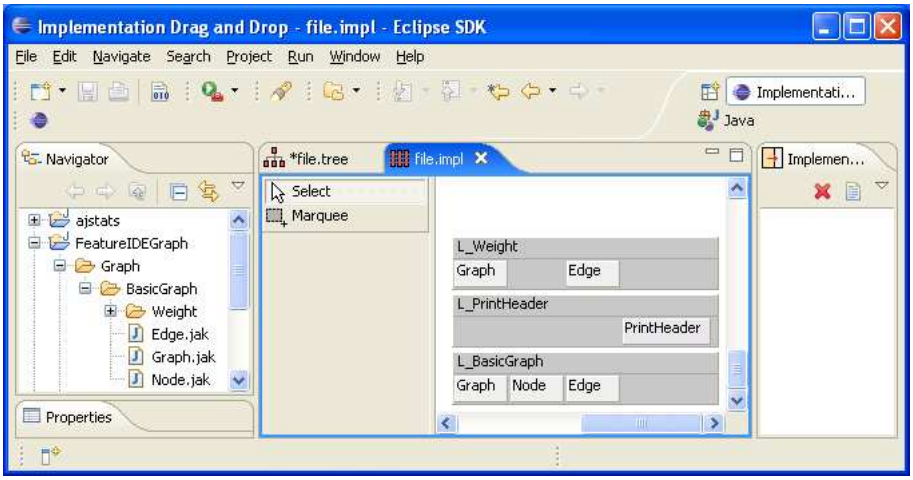


Figure 5.11: A stack of feature modules in FeatureIDE.

refinements of classes that contain methods only. However, in our experience refinements of classes often requires to add state variables, i.e., fields.

Virtual classes are a sophisticated mechanism to combine mixin composition with polymorphism [MMP89, EOC06]. Since virtual classes depend on the dynamic type of an enclosing object (class-valued attributes of the object), their semantics varies depending on the dynamic object identity. Virtual classes have been shown useful for the implementation of collaboration-based designs [AGMO06], but they require runtime instances of collaborations as a whole. It is not obvious how to align that with the AHEAD principle of uniformity where features contain beside code artifacts also non-code artifacts.

Nested inheritance [NCM04] and *classboxes* [BDN05] are related to virtual classes. The difference is that the types of the inner classes (the participants of a collaboration) do not depend on the runtime type of the enclosing object but on the static type of the enclosing class. Therefore, these both approaches are closer to the static composition semantics of AHEAD and AFMs than virtual classes are. Though nested inheritance and classboxes are in-language approaches they might be adopted to implement AFMs.

Delegation is a mechanism for implementing *object-based inheritance* [Lie86]. This enables the runtime reconfiguration of inheritance hierarchies and could be used to implement refinements of classes. As with virtual classes, this is only meaningful for collaborations that are instantiated and composed at runtime, e.g., as with *delegation layers* [Ost02]. This is difficult to align with the AHEAD architectural model and demands further investigation.

Implementation of Feature Modules

Several languages and tools support collaboration-based design. Potentially all of them could be used to implement feature modules and AFMs, however, each with some limitations.

Several languages support the abstraction and static composition of mixin layers, e.g., *C++ mixin layers* [SB02], *P++* [Sin96], and *Java layers* [CL01]. Other approaches exploit related ideas of composing and nesting class hierarchies [Coo89, Ern03], e.g., *Scala* [OZ05], *Jx* [NCM04], *J&* [NQM06], *Classbox/J* [BDN05], *CaesarJ* [AGMO06], *ContextJ* [CHdM06], to name a few; all these are in-language approaches.

A main advantage of AFMs is that they have AHEAD as an architectural model – the approaches mentioned above do not refer to any model. Hence, AFMs build upon the strengths of AHEAD: beside classes and aspects also other kinds of software artifacts may be included in a feature; feature modules are composed declaratively by means of a separate language (feature expressions) and checked against domain-specific design rules [Bat05]. This opens the door to automatic algebra-based optimization and compositional reasoning [BSR04]. It is not obvious how to carry this over to in-language approaches because the definition of features is done in the same language as their composition. That is, without a separated composition mechanism/language it is not trivial to implement mechanisms for optimizing and reasoning about composition specifications.

Jiazzi is a component system that supports the composition of binary collaborations via external rules [MFH01]. Since collaborations are abstracted outside of the language Jiazzi fits the AHEAD architectural model. However, while aspects could possibly be integrated, it is not obvious how to compile them independently.

Aspects and Collaborations

Several studies suggest to exploit the synergetic potential of mechanisms for aspects and collaborations, e.g., *Aspectual Components* [LLM99], *Adaptive Plug-and-Play Components* [ML98], *Pluggable Composite Adapters* [MSL00], *Caesar* [AGMO06], *Aspectual Collaborations* [LLO03], and *Object Teams* [Her02]. Since these approaches are highly influenced by one another, we compare our approach to their general concepts.

All the approaches mentioned abstract collaborations explicitly at the languages level and integrate different kinds of mechanisms associated to AOP, e.g., pointcuts and advice, *aspectual methods*, *traversals*, *adapters*, and *bindings*. These AOP mechanisms are intended mainly for the modularization of crosscutting concerns that arise from integrating two collaborations, which we call crosscutting integration in AFMs (cf. Sec. 4.1.4).

According to the design space of integrating AOP and FOP, the approaches above fall into the first category: they integrate AOP and FOP mechanisms at language level. This is advantageous when exploring issues like typing and polymorphism. Consequently, these approaches address issues such as *on-demand remodularization* (a.k.a. *a-posteriori integration*) of collaborations, *aspectual polymorphism*, dynamic aspect deployment, and distributed aspect components, which all are not supported by AFMs.

Aspects and Roles

Pulvermüller et al. [PSR00] and Sihmam et al. [SK03] propose to implement collaborations as single aspects that inject the participating roles into the base program by using introductions and advice. In our experience, explicitly representing collaborations by traditional object-oriented techniques and refinements facilitates program comprehensibility, which is in line with prior work [VN96c, MO04, LLO03, Ern01, OZ05, Ost02, TVJ+01, EOC06, TOHSMS99, BSR04, SB02, Ste00, Ste05, Bos99]. Favoring the approach of Pulvermüller et al. and and Sihmam et al. would lead in the end to a base program with empty classes that are extended by a series of aspects that inject structure and behavior. This would destroy the object-oriented design of the program and prevent the programmer from understanding the structure and behavior of the overall program as well as its individual features.

Hanenberg et al. [HU02], Kendall [Ken99], and Sihmam et al. [SK03] suggest to use aspects for implementing individual roles. In our context this would mean to replace each refinement of a class within a feature by one or more aspects. We and others [Ste05, MO04] argue that replacing traditional object-oriented techniques that suffice (e.g., inheritance) is questionable. Instead, we favor to use aspects only when traditional techniques fail.

Multi-Dimensional Separation of Concerns

Multi-dimensional separation of concerns (MDSoC) is a concept and method that aims at the clean separation of multiple, potentially overlapping and interacting concerns simultaneously, with support for on-demand remodularization to encapsulate new concerns at any time [TOHSMS99]. *Hyper/J* supports MDSoC for Java [OT00]; it introduces the concept of *hyperslices*, which maps roughly to an encapsulated collaboration of classes. It has been observed that features in AHEAD and hyperslices have many commonalities, especially regarding their composition semantics based on superimposition and their mechanisms for composing hyperslices/features [BLS03]. What differs in FOP is that integrating two features that are of a different structure demands a manual integration of the artifacts inside the features, e.g., by using wrappers or multiple inheritance [MO02, Her02]. Hyper/J supports declarative composition rules to establish a (possibly complex) mapping between different hyperslices. AHEAD supports only recursive merging of containment hierarchies by type and name.

AFMs, as extension to traditional feature modules, use aspects to establish the mapping between two unrelated features, as suggested first by Mezini et al. [MO03]. This is related to the Hyper/J composition rules, but at a lower level (language level). In this respect, AFMs follow more the approach of Caesar than of Hyper/J. It remains an open issue which variant of on-demand remodularization and crosscutting integration is preferable.

Aspect Quantification and Composition

Traditionally, aspects are quantified globally. Conceptually, they may affect potentially all elements of a program. Unfortunately, this approach ignores the principle of SWD that refinements are permitted to affect only refinements that have been applied in previous development steps [Wir71, Dij76, Par76, Par79]. Several studies show that this circumstance is directly responsible for several problems and penalties, e.g., unpredictable program behavior [MA05, DFS04, DFS02, LHBL06], weak modularity [GSF+05, GSC+03] and decreased evolvability [Lie04, LHBL06, GB03].

In order to address this issue, Lopez-Herrejon et al. propose an approach to aspect composition [LHBL06]. They model aspects as *functions* that operate on programs. Applying several aspects to a program maps to function composition. For example, $A_2(A_1(P))$ denotes a program P refined by aspect A_1, and the result refined by A_2. In this way the scope of aspects is restricted to a particular step in a program's development, e.g., A_1 can advise P but not A_2. This is called *bounded quantification* of aspects as opposed to *unbounded quantification* used in traditional AOP.

The idea of bounding aspect quantification can be integrated seamlessly into AFMs: Since a compiler, e.g., FeatureC++, knows to which development step (feature module) each aspect and each refinement belongs, it can determine which program parts the

aspects are permitted to affect. In [ALS05, KAS06, AL06] we discuss an approach for implementing *functional aspects* (aspects with bounded quantification) by restructuring their pointcut expressions. In a nutshell, pointcuts are modified such that the connected advice affects only join points associated with previous development steps, i.e., feature modules that have been applied before (Fig 5.12). A more detailed explanation is out of scope of this dissertation and can be found elsewhere [ALS05, KAS06, AL06].

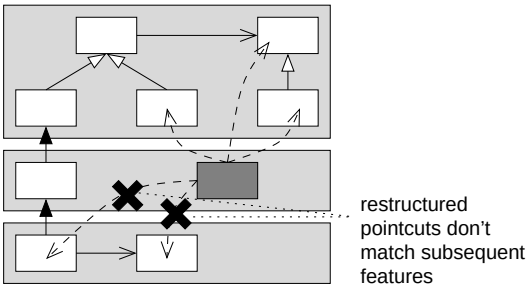


Figure 5.12: Implementing functional aspects via pointcut restructuring.

What is important is that the notion of AFMs enables, for the first time, to *implement and experiment with* bounded aspect quantification. Even if there is no agreement on the benefits of bounded aspect quantification [LHBL06], our approach may help to prove corresponding arguments and deliver empirical evidence.

Aspects and Information Hiding

One issue of AFMs is that current AOP languages do not respect the principle of information hiding [Ste06, SGS+05, Ald05, OAT+06]. However, there are several efforts to solve this problem, e.g., *open modules* [Ald05, OAT+06] and *information hiding interfaces* [SGS+05, GSS+06] propose module interfaces that specify explicitly which join points may be advised – the others are hidden. *Harmless advice* is a restricted form of advice that is designed to obey a weak non-interference property [DW06]. It may change the termination behavior of computations and use I/O, but it does not otherwise influence the final result of the mainline code. *Stratified aspects* adjust the quantification mechanism of aspects to avoid infinite recursion caused by advice that unintentionally advise itself [BFS06].

The point here is that AFMs can profit from these developments. Since AFMs do not depend on a specific host language, new languages can easily be integrated. This is a major advantage of AFMs compared to in-language approaches.

	heterogeneous	**homogeneous**
static	a set of refinements that add elements	one piece of inter-type declaration
basic dynamic	a set of refinements that override methods	one piece of basic advice
advanced dynamic	a set of advanced advice	one piece of advanced advice

Table 5.1: What implementation technique for which kind of crosscutting concern?

5.7 Summary

The notion of AFMs defines a feature as a collection of artifacts, among them classes, refinements, and aspects that collaborate. We argue that this is close to the ideal of what a feature should be. They are implemented by different kinds of artifacts, each artifact appropriate for a specific design or implementation problem.

The conceptual evaluation has shown that regarding almost any criterion AFMs perform better than aspects or Jak-like feature modules in isolation. However, mixin composition and aspect weaving overlap with regard to the implementation of refinements: (1) inter-type declarations and refinements of classes inject new members (static crosscuts); (2) advice and method overriding refine methods calls/executions (dynamic crosscuts). Hence, a crucial question arises: when to use what mechanism without interspersing both? As explained, our evaluation gives the answer: on the one hand, the programmer uses collaborations of classes and refinements in the situations in which they suffice, i.e., in implementing heterogeneous and basic dynamic crosscuts. On the other hand, the programmer uses aspects to implement certain kinds of crosscutting concerns, i.e., homogeneous and advanced dynamic crosscuts, where traditional feature modules fail. Table 5.1 summarizes what implementation technique should be used for which kind of crosscut.

We conclude that AFMs perform better than FOP and AOP by themselves because they combine the strengths of both – presuming programmers apply the right technique for implementing the right problem. While the guidelines in Table 5.1 are reasonable, they provide no certainty that the resulting implementation is structured appropriately nor that the combination of AOP and FOP mechanisms does not lead to hidden conflicts or inconsistencies. In Chapter 7, we present our experiences of applying AFMs to a non-trivial case study, thus evaluating our programming guidelines.

In the next chapter we address several interesting issues that arise from the integration of aspects into the incremental development model of FOP.

CHAPTER 6

Aligning Aspects and Stepwise Development

This chapter shares material with the APSEC'05 paper 'Aspect Refinement and Bounded Quantification in Incremental Designs' [ALS05].

AFMs integrate aspects into the incremental development style of FOP and AHEAD. Consequently, the following issues arise, which we address in this chapter: (1) does AOP fit with the principles of SWD and, if not, (2) how can AOP be aligned with SWD?

6.1 Aspects and Stepwise Software Development

aspects are just another kind of software artifact

AHEAD, the architectural model of AFMs, defines that a feature is implemented by a collection of collaborating software artifacts of varying types. In this sense aspects are just another kind of software artifact. The AHEAD principle of uniformity has an interesting consequence: since aspects are artifacts as any others, it is natural to refine them in a SWD manner as well. That is, AFMs may not only extend and modify classes via subsequent refinement but also aspects, which we call *aspect refinement (AR)*. Hence, AR is the consequential application of SWD principles to the world of AOP.

three use cases for AR

With AR, aspects evolve over time, as do all other software artifacts. In each development step aspects may be refined, i.e., extended and modified. In this dissertation we focus on three use cases for refining aspects, which may overlap in parts:

1. Adapting aspects to the changes made to a base program, e.g., join points have changed or new join points occur.

2. Tailoring aspects to changing user requirements, e.g., the user needs an aspect to implement a new design decision.

3. Decomposing aspects to decouple them from a specific configuration of the base program, e.g., a base program in different configurations demands aspects in different variants.

Applying AR to deal with the above situations means a decomposition and subsequent composition of an aspect out of a base aspect and a series of refinements. Refinements should be freely combinable – of course, in the limits of desired program behavior. This flexibility facilitates reuse of aspect code. The user-driven composition of AFMs and thus of aspects and their refinements customizes aspect-specific functionality. AR enables a similar improvement in reusability and customizability of aspect code as the analogous object-oriented mechanisms do for classes, e.g., mixins [BC90, SB02], refinements [BSR04], and virtual classes [MMP89, EOC06, OZ05].

unification of classes and aspects

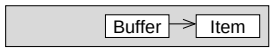
AR bears the potential to unify classes and aspects with respect to subsequent refinement. An advantage of this view is that several ideas of class refinement can be mapped directly to aspects, as we will show. But more interesting is the fact that it becomes possible to refine also aspect-specific structural elements, in particular pointcuts and advice, which opens new possibilities of aspect reuse and customization.

6.1.1 An Example of Aspect Refinement

Figure 6.1 illustrates the evolution of a program developed using AFMs. The program contains classes for buffers and sockets as well as aspects for synchronizing concurrent access to the data structures. The evolution spans four steps shown in four subfigures (I-IV). Each development step is explained in terms of its Java/Jak/AspectJ code and in diagram form; refinements of aspects are implemented as subaspects for the time being; aspect weaving is denoted by dashed arrows.

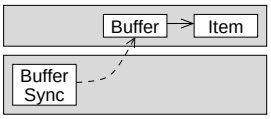
I. `Buffer` objects store sets of data items; class `Buffer` provides the methods `put` and `get` for accessing the stored items.
II. The aspect `BufferSync` synchronizes the access to the methods `put` and `get` of `Buffer` by invoking the methods `lock` and `unlock`.
III. The class `Stack` is introduced; in order to synchronize the access to `Stack` objects, the aspect `StackSync` refines the aspect `BufferSync`. Specifically, `StackSync` extends the set of intercepted method executions by `push` and `pop`; for that it overrides and reuses the pointcut `syncPC` of aspect `BufferSync`.
IV. The class `Socket` is introduced; a `Socket` object uses several `Buffer` and `Stack` objects. The aspect `SocketSync` limits the set of synchronized methods to those that are inside the control flow of `Socket`, i.e., method executions are synchronized only when they are initiated directly or indirectly by a `Socket` object. This is achieved by overriding the pointcut `syncPC` and restricting the set of captured join points via `cflow`.

```
1  class Buffer {
2    Vector<Item> buf = new Vector<Item>();
3    void put(Item e) { buf.add(e); }
4    Item get(int i) { return buf.get(i); }
5  }
```



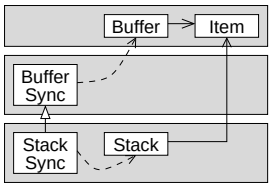
(I)

```
1  abstract aspect BufferSync {
2    pointcut syncPC() :
3      execution(Item Buffer.get(int)) ||
4      execution(void Buffer.put(Item));
5    Object around() : syncPC() {
6      lock();
7      Object res = proceed();
8      unlock();
9      return res;
10   }
11 }
```



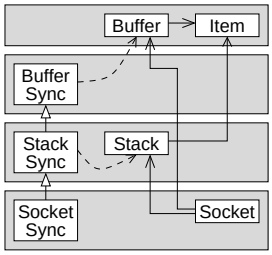
(II)

```
1  class Stack {
2    LinkedList<Item> list = new LinkedList<Item>();
3    void push(Item i) { list.addFirst(i); }
4    Item pop() { return list.getFirst(); }
5  }
6  abstract aspect StackSync extends BufferSync {
7    pointcut syncPC() :
8      BufferSync.syncPC() ||
9      execution(Item Stack.pop()) ||
10     execution(void Stack.push(Item));
11 }
```



(III)

```
1  class Socket {
2    void receive() {
3      Buffer buf = new Buffer();
4      Stack stack = new Stack(); /* ... */
5    }
6  }
7  aspect SocketSync extends StackSync {
8    pointcut syncPC() :
9      StackSync.syncPC() &&
10     cflow(execution(* Socket.*(..)));
11 }
```



(IV)

Figure 6.1: Four steps in the evolution of a program using AFMs.

AR is the application of SWD to AOP

This example illustrates the usefulness of refining aspects in a step-wise manner over several development steps. Aspect refinement is a logical consequence of applying SWD principles to AOP. The incremental development process makes the evolution of the program explicit. Design decisions are encapsulated and can be modified in separation as well as combined and reused in different variants. A reasonable desire is to derive different customized program variants that share common features and reuse invariant code. For example, one variant contains only a synchronized buffer:

$$BasicBuffer = BufferSync \bullet Buffer$$

another contains a buffer that is synchronized only with respect to calls from `Socket`:

$$SocketBuffer = SocketSync \bullet BufferSync \bullet Buffer$$

and a third contains a buffer that combines the entire functionality:

$$SocketStackBuffer = SocketSync \bullet StackSync \bullet BufferSync \bullet Buffer$$

6.1.2 Limited Language-Level Support for Aspect Refinement

Beside the advantages of AR, our example also demonstrates the shortcomings of AspectJ in supporting SWD:

Aspect inheritance: Inheritance is known as a concept for reusing and non-invasively refining software artifacts [Tai96]. Therefore, most AOP languages support aspect inheritance. Although this enables to refine aspects to some degree, it lacks flexibility to interchange and reuse refinements. Using aspect inheritance, a refinement (subaspect) is fixed to a specific base aspect. Hence, refinements cannot be combined flexibly in different orderings for customization and adaptation purposes. For example, we are not able to derive different variants of our buffer example without changing code invasively.

Constrained aspect extension: Using traditional aspect inheritance in AspectJ, an aspect has to be declared as *abstract* to be able to be refined. This means that adding a subaspect requires the programmer to modify the parent aspect. This and similar requirements[1] cause a fundamental problem regarding SWD: implementing an aspect in a particular development step forces the programmer to decide whether the aspect will be refined in a later step. Unfortunately, this cannot always be anticipated by the programmer. Hence, the programmer has a serious dilemma. Declaring the aspect as abstract makes it necessary to add later at least one concrete subaspect. But this may not happen and then the aspect does not work. If the programmer decides to declare an aspect as concrete (without modifier) he prevents the later refinement of this aspect.

[1] For example, refining a pointcut in AspectC++ requires to declare the parent pointcut as *virtual*.

Advice is not first-class: Advice is one of the main mechanisms of AOP [MK03b]. A piece of advice is invoked implicitly, i.e., it executes code when an associated pointcut matches. This prevents other advice or methods from invoking it explicitly. Since advice has no name it cannot be overridden and extended by another piece of advice, inside a refinement. This prevents reusing and customizing advice code.

Aspects are not functions: A refinement in SWD is modeled as a function [HFC76, Wir71, Par79, Bax92, BSR04, LHBL06, AL06]. It expects a program as input and returns a modified program as output. Applying a series of aspects to a program – which is in fact a series of refinements [LHBL06, LHBC05, AL06] – differs from this scheme. Potentially, each aspect may affect every artifact of a program no matter whether it is applied before or subsequently in the evolution of a program (*unbounded quantification*). This behavior does not follow a functional approach and bears some potential errors and misbehavior, as explained in Section 5.6. In this dissertation we do not address this issue since it has already been explored and solved in parts by introducing *bounded quantification* of aspects (cf. Sec. 5.6). Nevertheless, in Section 6.4 we discuss an interesting consequence of AR with regard to modeling aspects as functions.

The problems sketched above show that current AOP languages as exemplified by AspectJ do not support SWD appropriately at the language level. Consequently, we propose an alternative approach implementing AR and a set of accompanying language mechanisms.

6.2 Mixin-Based Aspect Inheritance

mixin composition of aspects

In order to support AR at the language level, we introduce the notion of *mixin-based inheritance* [BC90] to AOP: *mixin-based aspect inheritance* explicitly supports SWD at the language level by introducing mixin capabilities to aspects. Though most aspect languages, such as AspectJ, support a limited form of aspect inheritance, they do not flexibly enough to express refinements of aspects and their structural elements. Mixin-based aspect inheritance overcomes this limitation by decoupling refinements from base aspects and providing a set of accompanying language mechanisms for refining independently all the kinds of structural elements of aspects. Specifically, we provide mechanisms for refining pointcuts (*pointcut refinement*) and advice (*named advice*, *advice refinement*), which are tailored to AspectJ-like languages.

We use Jak as archetype for expressing AR at the language level. This emphasizes the uniformity of classes and aspects with respect to refinement. As with class refinement, aspect refinement could also be implemented using alternative mechanisms, such as virtual classes or traits. With respect to this issue, we refer to the discussion in Section 5.6.

Figure 6.2 shows a synchronization aspect (Lines 1-4) and a refinement (Lines 5-21) extending the aspect. Refinements may introduce new structural elements as well as extend existing ones, as we will explain soon. They can be applied to abstract and concrete aspects as well as to other refinements. This eliminates the dilemma of anticipating subsequently applied refinements by declaring base aspects as abstract. Moreover, it allows a series of refinements to be applied to an aspect in different permutations.

```
1  aspect Sync {
2    void lock() { /* locking access */ }
3    void unlock() { /* unlocking access */ }
4  }
5  refines aspect Sync {
6    int threads;
7    void lock() {
8      threads++; Super.lock();
9    }
10   void unlock() {
11     threads--; Super.unlock();
12   }
13   pointcut syncPC() : execution(Item Buffer.get(int)) ||
14                       execution(void Buffer.put(Item));
15   Object around() : syncPC() {
16     lock();
17     Object res = proceed();
18     unlock();
19     return res;
20   }
21 }
```

Figure 6.2: Adding members and extending methods via AR.

AR weaving semantics

Notably, refining aspects is conceptually different from applying aspects. Applying two aspects modifies the base program in two independent steps. In our example this would lead to two different instances of the synchronization aspect. Instead, AR results in two aspect fragments that are merged via mixin composition. That is, an aspect together with all of its refinements constitutes the final aspect that is woven *once* to the base program. Figure 6.3 illustrates this semantics of AR: on the left side there is an aspect and a set of compatible refinements. Subsequently, the base aspect is composed with a series of user-selected refinements, which results in a final aspect. This one is then woven to the base program (right side).

6.2.1 Adding Members and Extending Methods.

A refinement may extend an aspect by adding new members. As shown in Figure 6.2, the refinement adds a field (Line 6), a pointcut (Lines 13-14), and an advice (Lines 15-20). Refinements may also extend methods to reuse existing functionality (Lines 7-9 and 10-12). A method extension usually overrides and calls the parent method (Lines 8,11).

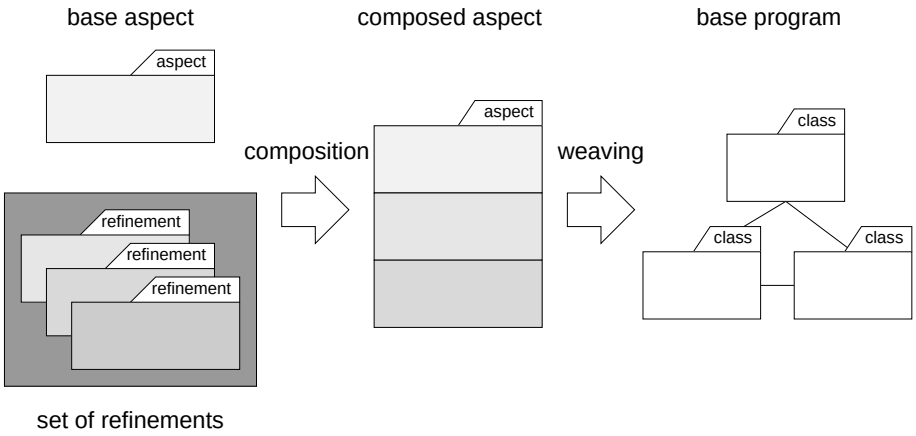


Figure 6.3: AR composition and weaving semantics.

6.2.2 Pointcut Refinement

A refinement may extend the pointcuts of an aspect. Recall our example aspect that synchronizes the access to the methods of `Buffer` (cf. Fig. 6.1). For this aspect we defined two refinements, an aspect that extends the set of advised join points by all executions of `Stack` methods (III), and an aspect that constrains this set to executions that occur in the control flow of `Socket` methods (IV). Both aspects were derived using traditional aspect inheritance. They override the pointcut `syncPC`, reuse its expression, and add new pointcut expressions that extend or constrain the set of matched join points.

decoupling refinements from base pointcuts

In AspectJ, pointcuts have to be accessed by their full-qualified name, in our example, `BufferSync.syncPC`. Thus, the programmer is forced to hard-wire the aspect to be refined and the subaspect. This tight coupling decreases reusability. Figure 6.4 depicts the synchronization aspect for `Buffer` and our refinements regarding `Stack` and `Socket`, but now implemented using mixin-based aspect inheritance. Using `Super` the programmer refers to the parent's pointcut (*base pointcut*) without being aware of what actual sequence of refinements is applied to the base aspect. For example, with traditional inheritance each refinement would change the final type of the aspect and thus fix the pointcut refinement to a specific base aspect. With mixin-based inheritance the order is variable.

semantics of pointcut refinement

The semantics of pointcut refinement is as follows: the most refined (specialized) pointcut in a series of pointcut refinements specifies when connected advice is executed. Which pieces of advice are executed can be specified all along the refinement chain, i.e., in every refinement of an aspect advice may be connected to a pointcut, although the base pointcut was declared before and refinements to that pointcut subsequently. Figure 6.5 shows a pointcut that matches a set of join points (dotted arrow), that triggers

```
1  aspect Sync {          // synchronize Buffer
2    pointcut syncPC() : execution(Item Buffer.get(int)) ||
3                        execution(void Buffer.put(Item));
4    Object around() : syncPC() {/* synchronization */}
5  }
6  refines aspect Sync { // synchronize Stack
7    pointcut syncPC() : Super.syncPC() || execution(* Stack.*(..));
8  }
9  refines aspect Sync { // only within cflow of Socket
10   pointcut syncPC() : Super.syncPC() && cflow(execution(* Socket.*(..)));
11 }
```

Figure 6.4: Altering the set of locked methods via pointcut refinement.

a connected advice (dashed arrow), and an advice that advises the selected join points (dot-dashed arrow). Figure 6.6 demonstrates that refining this pointcut (solid arrow) alters the triggering mechanism: the most refined pointcut extends the set of matched join points (dotted arrows) and triggers the advice (dashed arrow), albeit the advice was defined and connected in the base aspect. After the refinement the advice advised the extended set of join points (dot-dashed arrows).

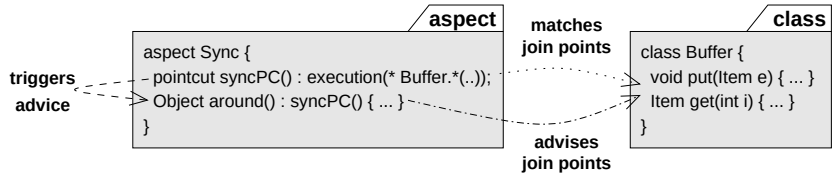


Figure 6.5: Pointcut-advice-binding.

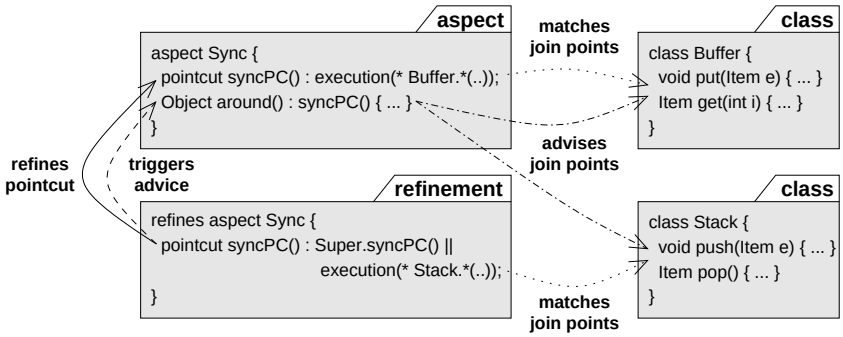


Figure 6.6: The most refined pointcut triggers connected advice.

6.2.3 Advice Refinement

Before explaining advice refinement it is necessary to introduce the notion of *named advice*.

Named Advice

Named advice is a named element of an aspect. It can be overridden and referred to from advice inside subsequent refinements.

named advice = unnamed advice + advice method

Figure 6.7 depicts an aspect for synchronization that contains a named advice (Lines 3-8). Named advice is defined by a result type (`Object`), an advice type (`around`), a name (`syncMethod`), an argument list (empty), an exception list (empty), a binding to a pointcut (`syncPC`), and an advice body. One can think of named advice as a pair of unnamed advice and a separate method, which we call *advice method*. The advice method contains the whole advice functionality; unnamed advice simply invokes this method and passes all arguments (Fig. 6.8). The difference is that named advice has full access to the dynamic context (`proceed` and join point API). Though named advice can be implemented differently, this view is helpful for understanding the semantics of advice refinement.

```
1 aspect Sync {
2   pointcut syncPC() : execution(* Buffer.*(..));
3   Object around syncMethod() : syncPC() {
4     lock();
5     Object res = proceed();
6     unlock();
7     return res;
8   }
9 }
```

Figure 6.7: An aspect with named advice.

Refining Named Advice

advice refinement = method refinement

As opposed to traditional advice, named advice can be refined in subsequent development steps. The key idea is to treat named advice in subsequent refinements similarly to a method. This is possible since named advice has at least a name, a result type, and an argument list. As mentioned, named advice can be understood roughly as a pair of unnamed advice and corresponding advice method. Hence, an advice refinement simply refines the advice method by method overriding, i.e., by defining a method with the same name and signature as the piece of named advice to be refined.

```
1  aspect Sync {
2    pointcut syncPC() : execution(* Buffer.*(..));
3    Object around() : syncPC() {
4      return syncMethod();
5    }
6    Object syncMethod() {
7      lock();
8      Object res = proceed();
9      unlock();
10     return res;
11   }
12   Object proceed() { /* invoking the advised method */}
13 }
```

Figure 6.8: Implementing named advice as pair of unnamed advice and advice method.

Figure 6.9 depicts an aspect that refines our synchronization aspect by extending its named advice. The refinement contains an advice method `syncMethod` (Lines 3-8) that overrides the parent named advice by counting the number of threads. Since we exploit method overriding, the refining method must have the same name and the same signature as the parent advice. The keyword `Super` is used to refer to the parent advice (Line 4).

```
1  refines aspect Sync {
2    int count = 0;
3    Object syncMethod() {
4      count++;
5      Object res = Super.syncMethod();
6      count--;
7      return res;
8    }
9  }
```

Figure 6.9: Refining named advice.

named advice with arguments

Figure 6.10 depicts a more complex example of advice refinement, in which the named advice has multiple arguments: a logging aspect advises all executions of `Item.toString` (Lines 2-3). A reference to the `Item` object that is called is passed to a named advice (Lines 4-6) that prints out some logging text (Line 5). Additionally, the named advice has a second argument, a reference to the resulting `String` object, which is expressed by the keyword `returning` (Line 4)[2]. Refining named advice subsequently (Lines 10-13), we introduce an advice method with the same name *and* the same signature. In our example the signature is composed of the two advice arguments[3].

[2] The keyword `returning` means that the advice is executed only when the method execution terminates without throwing an exception.

[3] Advice declares arguments at two positions: (1) behind its name and (2) behind the keywords `returning` or `throwing`.

```
1  aspect Logging {
2    pointcut ItemToString(Item i) :
3      execution(String Item.toString()) && this(i);
4    after LogToString(Item i) returning(String s) : ItemToString(i) {
5      System.out.println("item:" + i + "=" + s);
6    }
7  }
8  refines aspect Logging {
9    FileBuffer buf = new FileBuffer("foo");
10   void LogToString(Item i, String s) {
11     Super.LogToString(i, s);
12     buf.write("item:" + i + "=" + s);
13   }
14 }
```

Figure 6.10: Refining named advice with arguments.

named advice behaves like virtual methods

The semantics of named advice is similar to a virtual method, which passes the control flow to the most specialized descendant method of the inheritance chain. Mapped to advice refinement this means that, when the associated pointcut matches, the most specialized advice method is invoked. Figure 6.11 shows an advice method that refines a named advice (solid arrow). It is is executed (dashed arrows) when the pointcut `syncPC` matches (dotted line). Programmers use `Super` to navigate the refinement chain upwards. The root of the refinement chain defines to which pointcut the piece of advice is bound.

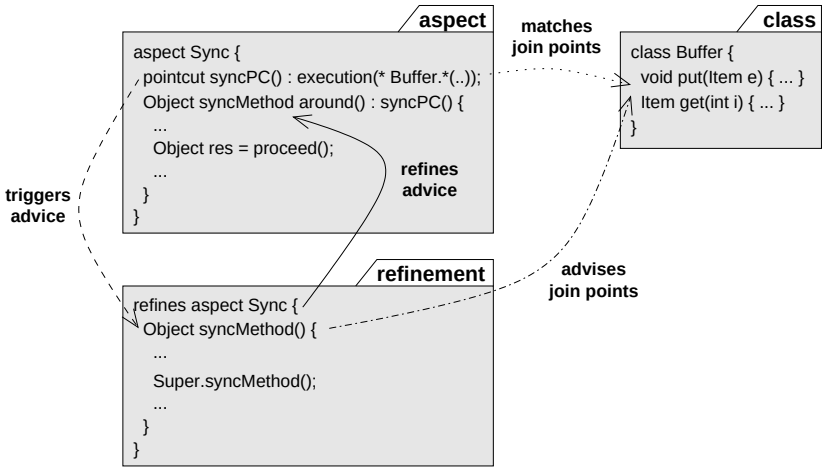


Figure 6.11: Semantics of advice refinement.

accessing the join point context in named advice

An issue that we left out is how to use and access `proceed` and contextual information within named advice and its refinements. We made no statement as to which information of the exposed context of a join point should be visible to descendant advice methods. This issue arises because programmers may access the context using `proceed` or runtime

variables as `thisJoinPoint`. Thus, one may use information that is not passed explicitly via the advice interface. The question that arises is: should refinements have unlimited access to context information and `proceed`?

access rules

We argue that an advice refinement should only be permitted to access those pieces of context information that are passed via the advice interface, and thus are part of the advice method signature. This would preclude invoking `proceed` or accessing `thisJoinPoint` from within an advice method. For example, in our logging example the refinement of the advice accesses only those objects that were passed via the advice interface. To preserve simplicity and safety the usage of the reflective support for accessing context information (e.g., `thisJoinPoint`) is forbidden in advice refinements. Furthermore, we do not allow named advice to be invoked directly by other advice and methods – such a mechanism is out of scope of this dissertation and addressed elsewhere (cf. Sec. 6.4).

6.2.4 Discussion

AR and its implementation via mixin-based aspect inheritance offer the following benefits: they allow a base aspect to be composed with a series of refinements, thus enabling to customize and reuse aspect code flexibly. Pointcut refinement decouples refinements from their immediate base aspects, thus enhancing the composability and customization of the aspect weaving behavior. Advice refinement promotes reuse in the same way as method extension between classes. Named advice can be reused in different variants of an aspect, thus supporting the customization of advice code.

At the beginning of this chapter we identified three beneficial use cases for AR, which we now want to revisit:

1. A programmer applies a refinement to adapt an aspect to the changes made to a base program. For example, Figure 6.12 shows an aspect that counts the updates of `Buffer` objects (Lines 1-7) and a refinement that adapts the aspect to count also executions of `clear` (Lines 8-10) that updates the `Buffer` object state as well; this is achieved by pointcut refinement (Line 9).

2. A programmer can customize an aspect to react to a changed user requirement. Suppose a new design decision that requires our `UpdateCounter` aspect to inform a listener when an update operation was performed. Figure 6.13 shows a refinement that implements this design decision using named advice refinement.

3. A programmer can decompose an aspect to decouple it from a specific configuration of the base program. For example, Figure 6.14 shows an aspect that introduces a new interface `Serializable` to a set of target classes (`Buffer`, `Stack`). Figure 6.15

```
1  aspect UpdateCounter {
2    int count = 0;
3    pointcut updatePC() : execution(void Buffer.put(Item));
4    after updateCounter() returning: updatePC() {
5      count++;
6    }
7  }
8  refines aspect UpdateCounter {
9    pointcut updatePC() : Super.updatePC() || execution(void Buffer.clear());
10 }
```

Figure 6.12: Counting the updates of `Buffer` objects.

```
1  refines aspect UpdateCounter {
2    UpdateListener listener = null;
3    void setListener(UpdateListener l) { listener = l; }
4    void updateCounter() {
5      Super.updateCounter();
6      listener.notify();
7    }
8  }
```

Figure 6.13: Notify a listener when `Buffer` objects are updated.

shows the result of decomposing this aspect into a base and two refinements, where each refinement introduces the interface to one target class. Before composing and compiling the final program, a programmer or a tool select only those refinements that target classes that are actually present in the program configuration, e.g., when `Stack` is present then also the according refinement is present (Lines 7-9).

```
1  aspect Serialization {
2    /* ... */
3    declare parents : (Buffer || Stack) implements Serializable;
4  }
```

Figure 6.14: Introducing the interface `Serializable` to `Buffer` and `Stack`.

AR improves reuse and customization

The use cases discussed have one thing in common: aspect code (i.e., base aspect, refinements) can be reused in different variants of a program; aspects can be customized to the specific needs of a programmer or to fit the structure of the base program.

AFMs and AR

It is worth to note that without the notion of AFMs it would be difficult to realize AR. The layered structure of AHEAD designs assigns to each aspect an enclosing feature module, which is associated to a development step. This information helps to organize and compose refinements and their base aspects, which is explained elsewhere [KAS06].

In the context of AFMs, decomposing an aspect into a base aspect and several refinements means decomposing the enclosing AFM into several pieces that are themselves

```
1 aspect Serialization {
2   /* ... */
3 }
4 refines aspect Serialization {
5   declare parents : Buffer implements Serializable;
6 }
7 refines aspect Serialization {
8   declare parents : Stack implements Serializable;
9 }
```

Figure 6.15: Decomposed `Serialization` aspect.

AFMs (see Fig. 6.16). Thus the number of AFMs increases but this provides the necessary flexibility to compose different sets of features.

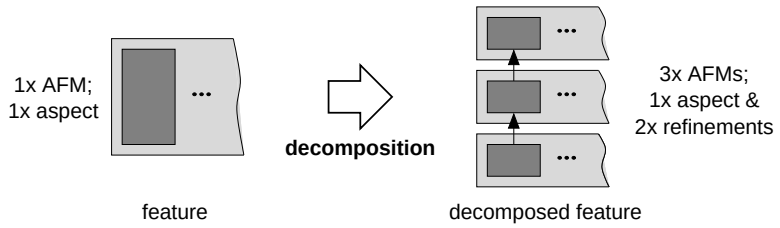


Figure 6.16: Decomposing aspects by decomposing AFMs.

AR as AHEAD operator

According to AHEAD's algebraic model, mixin-based aspect inheritance is simply a composition operator that is invoked when aspects (and their refinements) of different development steps are composed. Hence, this aspect composition operator corresponds to the class composition operator, which composes classes using mixin-based inheritance.

6.3 Tool Support

6.3.1 ARJ

ARJ is a language extension of AspectJ that supports aspect refinement. It has been implemented during this dissertation project as a modular extension to the *abc* compiler framework [ACH+05]. It extends the abc parser enabling it to recognize our new syntactical elements and it adds several frontend and backend passes for implementing a syntax tree transformation. ARJ is implemented to work in concert with the AHEAD Tool Suite and Jak to integrate AR into AFMs: ARJ expects a feature expression in form of an AHEAD equation file. AFMs are represented by containment hierarchies that contain the associated aspects, classes, and refinement files (class and aspect refinements). Further details about ARJ are explained elsewhere [KAS06]. The current

version of ARJ supports all language constructs proposed here. The compiler as well as several documents and examples can be downloaded from the ARJ Web site[4].

6.3.2 FeatureC++

Our FeatureC++ compiler (cf. Sec. 5.5.1) supports a limited form of AR. AspectC++ aspects can be refined ('`refines aspect ...`') by adding members, extending methods, and refining pointcuts. In the current version of FeatureC++ there is no support for named advice or advice refinement.

6.4 Related Work

Higher-Order Functions, Pointcuts, and Advice

Aspects are refinements and can be modeled as functions [LHBL06, LHBC05, AL06]. As already explained in Section 5.6, treating aspects as functions helps to avoid several problems arising from the unbounded quantification of aspects, which are not repeated here. What is interesting is that in the light of the function model, AR is related to *higher-order functions*. A higher-order function expects a function as input and returns another function as output. Since aspects can be modeled as functions a refinement of an aspect can be understood as a function that applies to a function, which is a higher-order function, e.g., $R(A)(P)$, where P is a program, A is an aspect, and R is a refinement of A. It remains open how high-order functions fit with current algebraic models of aspects and features [LHBL06, LHBC05, AL06].

Our notion of aspect refinement is related further to higher-order pointcuts and advice, discussed by Tucker and Krishnamurthi [TK03]. They integrate advice and pointcuts into languages with higher-order functions and model them as first-class entities. Pointcuts can be passed to other pointcuts as arguments. Thus, they can be modified, combined, and extended. In this respect, our approach of aspect and pointcut refinement is similar. We can combine, modify, and extend pointcuts by applying subsequent refinements.

Due to the opportunity to refine named advice, we can also modify and extend advice using subsequent advice. This corresponds to higher-order advice that expects a piece of advice as input and returns a modified piece of advice. Named advice can be passed to other advice – usually to advice that refines other (input) advice. Thus, refining advice is similar to passing a piece of advice to higher-order advice.

[4] http://wwwiti.cs.uni-magdeburg.de/iti_db/arj/

Implementation of Aspect Refinement

As discussed in Section 5.6, refinement can be implemented in different ways, such as mixins, virtual classes, nested inheritance, or traits. In the context of aspects and AR, a further possibility arises: an aspect could by refined itself via advice and inter-type declarations of another aspect. In this case aspects themselves are part of a base program and the programmer has the choice to refine them via mixins, etc. or aspect weaving. The answer to the question when to use which refinement mechanism is the same as for refining classes: in the case of homogeneous and advanced dynamic crosscuts, aspects are used to refine base aspects; in all other cases, our notion of AR in the form of mixins or virtual classes is used to refine base aspects.

Unifying Advice and Methods

Using the annotation-based programming style of AspectJ, aspects are implemented as classes and advice is implemented as method and declared as such via annotation. In this programming style advice is already named and can be refined by method overriding. However, it is not obvious how this relates to other mechanisms for refinement, e.g., pointcut refinement.

Rajan and Sullivan propose the notion of *classpects* that combine capabilities of aspects and classes [RS05]. A classpect associates to each piece of advice a method that is executed for advising a particular join point. Moreover, classpects unify aspects and classes with respect to instantiation. Since advice is implemented via methods, it could be refined. However, the authors of classpects do not make a statement about this nor about the consequences.

Aspects and Genericity

Several recent approaches enhance aspects with genericity, e.g., *Sally* [HU03], *Generic Advice* [LBS04], *LogicAJ* [KR06], *Framed Aspects* [LR04]. This improves reusability of aspects in different application contexts. Aspect refinement and mixin-based aspect inheritance provide an alternative way to customize aspects, i.e., by composing a base aspect and a series of desired refinements. However, ideas on generic aspects can be combined with our compositional approach, just as *generic feature modules* combine AFMs with generics [AKL06].

AspectJ Design Patterns

Hanenberg and Unland discuss the benefits of inheritance in the context of AOP [HU01, HS03]. They argue that aspect inheritance improves aspect reuse and propose design patterns that exploit structural elements specific to AspectJ. Their patterns *pointcut method*, *composite pointcut*, and *chained advice* suggest to refine pointcuts in subsequent development steps to improve customizability, reusability and extensibility. Due to its flexibility, AR can enhance these patterns by simplifying the composition of aspects. The pattern *template advice* can be enhanced by the notion of named advice because it becomes possible to refine advice directly.

Feature-Optionality Problem

In FOP, features may depend on (or interact with) other features that are optional [Pre97, LBL06]. In order to be reliable with regard to putting in and removing optional features, Prehofer proposes to split features into slices, i.e., into a base feature and several so called *lifters* [Pre97]. Lifters encapsulate those pieces of code that depend on (and interact with) other features. When composing a program from features, a programmer or a tool selects for each feature the base feature and those lifters that refer to features that actually participate in the current configuration. Liu et al. lay an algebraic foundation for this methodology [LBL06].

Our method of splitting aspects into pieces to resolve dependencies between aspects and classes of a base program is similar to their approach: our refinements correspond to lifters, but in the context of AOP.

6.5 Summary

Aspect refinement is the incarnation of SWD in AOP. It follows directly from the integration of aspects and feature modules. AR unifies classes and aspects with respect to subsequent refinement. We have illustrated three use cases where AR improves reuse and customization of aspect code. To introduce the principles of SWD at the programming language level, we proposed mixin-based aspect inheritance and a set of accompanying language constructs that facilitate SWD: pointcut refinement, named advice, and advice refinement.

Though we feel certain that AR is an improvement in reuse and customization capabilities of AOP, as mixins, refinements, and virtual classes are to OOP, in Chapter 7 we evaluate the notion of AR by means of a non-trivial case study.

CHAPTER 7

Case Study: A Product Line for Peer-to-Peer Overlay Networks

This chapter shares material with the GPCE'06 paper 'When to Use Features and Aspects? A Case Study' [AB06].

This chapter demonstrates the practical applicability of AFMs and AR to a medium-sized case study. Furthermore, we address the interesting and fundamental issue, which arises from the previous two chapters: when should a programmer use feature-oriented mechanisms (i.e., classes, virtual classes, and mixins) and when should aspect-oriented mechanisms (i.e., introductions, pointcuts, and advice) be used to implement features of a product line? That is, how do our programming guidelines perform in a non-trivial software project? Our case study gives answers, provides a set of supporting statistics, and reveals open issues.

7.1 Overview of P2P-PL

We use a product line for *peer-to-peer overlay networks (P2P-PL)*, which was implemented by the author [BAS05, AB05b, AB05a]. Beside the basic functionality as routing and data management in a P2P network [ATS04], P2P-PL supports several advanced features, e.g., query optimization based on flexible routing path selection [AB05b], metadata propagation for the continuous exchange of control information among peers [BAS05], incentive mechanisms to counter peers that misbehave (*free riders*) [BB06]. Numerous experiments concerning these features demanded many different configurations to make statements about their specific effects, their variants, and combinations [BAS05]. Hence, P2P-PL seemed to be a good test case for AFMs and AR.

fine-grained design

P2P-PL has a fine-grained design. It follows the principle of evolving a design by starting from a minimal base and applying incrementally minimal refinements to implement design decisions [Par79]. In its current state, it consists of 113 end-user visible features, categorized into several subdomains. An *end-user visible* feature is an increment in program functionality that users (in case of P2P-PL the author is the user) feel is important in describing and distinguishing programs within a product line.

Figure 7.1 depicts the first two levels of the organizational structure of P2P-PL. The set of features is divided into features of plain P2P systems (*P2P*), of distributed hash tables (*DHT*) – a special kind of P2P system [ATS04], of content-addressable networks (*CAN*) – a special kind of DHT [RFH+01], and features for experimental purposes (*Exp*). The subdomains are subdivided as well. The number behind each subdomain refers to the number of features contained in the subdomain, e.g., subdomain *Peers* contains four features. The actual features are not shown because of their large number.

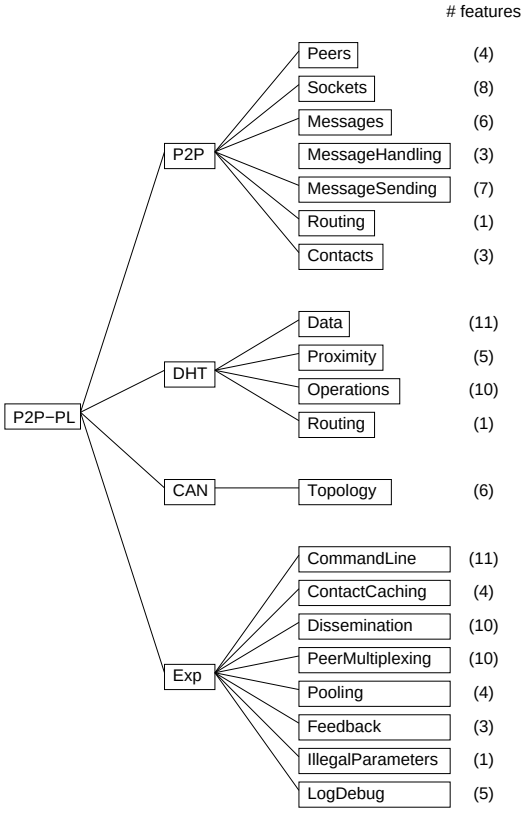


Figure 7.1: The organizational structure of P2P-PL.

implementation

P2P-PL was implemented using the AHEAD Tool Suite (ATS) and ARJ. As explained in Sections 5.5.2 and 6.3.1, the ATS served for implementing feature modules and ARJ

for composing and weaving aspects within feature modules. The code base of P2P-PL is approximately 6.4 thousand lines of source code, distributed over 113 features.

7.1.1 Aspectual Feature Modules in P2P-PL

14 of the 113 end-user visible features of P2P-PL (12%) use aspects (see Tab. 7.1); the remaining 99 features were implemented as traditional feature modules – without aspects. To give the reader an impression of how aspects and mixins have been combined in P2P-PL, we explain two simplified examples of AFMs.

aspect	**description**
responding	sends message replies automatically
forwarding	forwards messages to adjacent peers
message handler	base aspect for message handling
pooling	stores and reuses open connections
serialization	prepares objects for serialization
illegal parameters	discovers illegal system states
toString	introduces `toString` methods to several classes
log/debug	a mix of logging and debugging
dissemination	piggyback meta-data propagation
feedback	generates feedback by observing peers
query listener	waits for query response messages
command line	provides command line access
caching	caches peer contact data
statistics	collects and calculates runtime statistics

Table 7.1: Aspectual Mixin Layers used in P2P-PL.

Feedback Generator

feedback counters free riders

The feedback generator feature is part of an incentive mechanism for penalizing *free riders* – peers that profit by the P2P network but do not contribute adequately [BB06]. A feedback generator feature, on top of a peer implementation, identifies free riders by keeping track of whether other peers respond adequately to messages. If this is not the case, an observed peer is considered a free rider. Specifically, the generator observes the traffic of outgoing and incoming messages and traces which peers have responded in time to posted messages. The generator creates positive feedback to reward cooperative peers and negative feedback to penalize free riders. Feedback information is represented by objects of class `Feedback` and stored in a repository (`FeedbackRepository`); it is passed to other (trusted) peers attached to outgoing messages in order to inform them about free riding. Based on the collected information, a peer judges the cooperativeness

of other peers. Messages from peers considered free riders are ignored – only cooperative peers profit by the overall P2P network [BB06].

feedback generation is crosscutting

The implementation of the feedback generator crosscuts the message sending and receiving features. As Figure 7.2 shows, the feedback generator AFM contains an aspect (dark gray) and introduces four new classes for feedback management. Additionally, it refines the peer abstraction (by mixin composition) so that each peer owns a log for outgoing queries and a repository for feedback information.

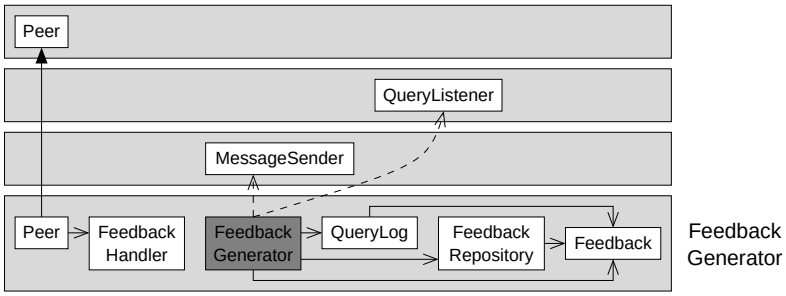


Figure 7.2: Feedback generator AFM.

While the feedback generator feature implements a heterogeneous crosscut, it relies on dynamic context information, i.e., it is an advanced dynamic crosscut. Figure 7.3 lists an excerpt of the aspect `FeedbackGenerator`. The first advice refines the message sending mechanism by registering outgoing messages in a query log (Lines 2-7). It is executed only if the method `send` was called in the dynamic control flow of the method `forward`. This is expressed using the `cflow` pointcut (Line 5) and avoids advising unintended calls to `send`, which are not triggered by the message forwarding mechanism[1]. The second advice intercepts the execution of a query listener task for creating feedback (Lines 8-10).

Figure 7.4 lists the refinement of the class `Peer` implemented as a mixin[2]. It adds a feedback repository (Line 2) and a query log (Line 3). Moreover, it refines the constructor by registering a feedback handler in the peer's message handling mechanism (Lines 4-7).

AFM encapsulates multiple artifacts

In summary, the feedback generator AFM encapsulates four classes that implement the basic feedback management, an aspect that intercepts the message transfer, and a mixin that refines the peer abstraction. Omitting AOP mechanisms would result in code tangling and scattering since the retrieval of dynamic context information crosscuts

[1] The background of using `cflow` it that the method `send` is called many times inside a peer, but we wanted to advise only those executions of `send` that occur when forwarding a message to another peer.

[2] The actual syntax for constructor refinement in Jak differs slightly [BSR04].

```
1  aspect FeedbackGenerator { ...
2    after(MessageSender sender, Message msg, PeerId id) :
3      target(sender) && args(msg, id) &&
4      call(boolean MessageSender.send(Message, PeerId)) &&
5      cflow(execution(boolean Forwarding.forward(..))) &&
6      if(msg instanceof QueryRequestMessage)
7      { /* ... */ }
8    after(QueryListener listener) : target(listener) &&
9      execution(void QueryListener.run())
10     { /* ... */ }
11 }
```

Figure 7.3: Feedback generator aspect (excerpt).

```
1 refines class Peer {
2   FeedbackRepository fr = new FeedbackRepository();
3   QueryLog ql = new QueryLog();
4   Peer() {
5     Super();
6     FeedbackHandler fh = new FeedbackHandler(this);
7     this.getMessageHandler().subscribe(fh);
8   }
9 }
```

Figure 7.4: Feedback management refinement of the class `Peer`.

other features, e.g., clients of the message forwarding mechanism. On the other hand, implementing this feature as one standalone aspect would not reflect the structure of the P2P-PL framework that includes feedback management. All would be merged in one or more aspect(s) that would decrease program comprehension. Our AFM encapsulates all contributing elements coherently as a collaboration that reflects the intuitive structure of the P2P-PL framework we had in mind during its design.

Connection Pooling

reusing open connections

The connection pooling feature is a mechanism to save time and resources for frequently establishing and shutting down connections. To integrate connection pooling into P2P-PL, we implemented a corresponding AFM. Figure 7.5 shows this AFM consisting of the aspect `Pooling` and the class `Pool`. The aspect intercepts all method calls that create and close connections[3]. The pool stores open connections.

Figure 7.6 lists the pooling aspect; it uses a pool for storing references to connections (Line 2). The pointcuts `close` (Lines 3-4) and `open` (Lines 5-6) match the join points that are associated to shutting down and opening connections. Named advice `putPool` (Lines 7-9) intercepts the shutdown process of connections and instead stores the associated `ClientConnection` objects in a `Pool` object. Named advice `getPool` (Lines 10-13)

[3] Note that this is not ideally visualized because the calls are intercepted at the client side.

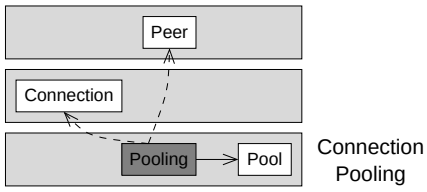


Figure 7.5: Connection pooling AFM.

recovers open connections (if available) and passes them to clients that request a new connection. This aspect makes use of the built-in pointcut `this` to limit the advised calls to those that originate from `MessageSender` objects.

```
1  aspect Pooling {
2    static Pool pool = new Pool();
3    pointcut close(ClientConnection con) :
4      call (void ClientConnection.close()) && target (con) && this(MessageSender);
5    pointcut open(ClientSocket socket) :
6      call (ClientConnection.new(ClientSocket)) && args(socket) && this(MessageSender);
7    Object around putPool (ClientConnection con) : close(con) {
8      pool.put(con); return null ;
9    }
10   ClientConnection around getPool (ClientSocket socket) : open(socket) {
11     if (pool.empty(socket)) return proceed(socket);
12       return (ClientConnection)pool.get(socket);
13   }
14 }
```

Figure 7.6: Connection pooling aspect (excerpt).

Why not using a feature module?

Implementing this feature using FOP exclusively would lead to code tangling and scattering. We would have to modify `MessageSender` at every place at which the method `close` and the constructor of `ClientConnection` is called. Simply extending both is not possible since this would affect *all* calls, not only those that originate from `MessageSender`. We solve this problem elegantly using advice that advises calls conditionally, i.e., dependently on the type of the caller, which is an advanced dynamic crosscut.

Furthermore, we did not implement `Pool` as a nested class within the aspect `Pooling` to emphasize that it is regular part of the P2P-PL. We consider it part of the collaboration of artifacts that implement the feature. Subsequent refinements may extend and modify the class `Pool`.

7.1.2 Aspect Refinement in P2P-PL

We used AR to refine 7 of the 14 aspects used in P2P-PL. That is, we decomposed each of the 7 AFMs with aspects into a base AFM and multiple refinements, where

each refinement is an AFM itself (cf. Fig. 6.16 in Sec. 6.2.4). We explain two simplified examples below.

Serialization

feature dependencies in P2P-PL

The feature *Serialization* consists only of one aspect. Figure 7.7 depicts the aspect `Serialization` tailored for a fully-configured P2P system. It enumerates a list of `declare parent` statements that add the interface `Serializable` to a set of target classes[4]. The key thing to note here is that the *list of declared parents depends on the set of features that are in a P2P system*. This means that, if the feedback generator feature is not present in a target P2P system, the statement `declare parents: Feedback` in Figure 7.7 would need to be removed from the `Serialization` aspect, otherwise a warning would be reported (because there would be no `Feedback` class)[5]. The same holds for `PeerId`, `Contact`, `Key`, and `DataItem`. Thus, the definition of the `Serialization` aspect depends on other features that are present in a target system (according to Sec. 6.2.4, it is an instance of use case 3).

We model the synthesis of a customized `Serialization` aspect by refining a base aspect. That is, we apply AR to break apart the `Serialization` aspect into smaller pieces – a base aspect + a series of refinements – to synthesize a system-specific `Serialization` aspect.

```
1 aspect Serialization {
2   declare parents : Message implements Serializable;
3   declare parents : PeerId implements Serializable;
4   declare parents : Contact implements Serializable;
5   declare parents : Key implements Serializable;
6   declare parents : DataItem implements Serializable;
7   declare parents : Feedback implements Serializable;
8   ...
9 }
```

Figure 7.7: Serialization aspect (excerpt).

Figure 7.8 lists the decomposed `Serialization` aspect, i.e., a base `Serialization` aspect and a set of refinements (merged in one listing). Each refinement introduces the `Serializable` interface to only one target class. This enables programmers to choose only those pieces (refinements) that are required for a particular configuration of P2P-PL. For example, the refinement that targets the class `Feedback` (Lines 10-12) is included

[4] This particular aspect could also be implemented by enumerating all target classes in a logical expression, e.g., '`declare parents : (Message || PeerId || ... ) implements Serializable`'.

[5] Not all aspect compilers will issue warnings; some may issue errors when non-existent classes are referenced. Our use of AR avoids compiler warnings/errors at the expense of imposing more structure on synthesized P2P-PL programs.

only in a program if the feedback generator feature is added as well. How fine-grained this decomposition should be depends on the flexibility of composing end-user visible features. In P2P-PL, we split the compound *Serialization* feature into 12 pieces (1 base aspect and 11 refinements).

```
1  aspect Serialization {
2    declare parents : Message implements Serializable;
3  }
4  refines aspect Serialization {
5    declare parents : PeerId implements Serializable;
6  }
7  refines aspect Serialization {
8    declare parents : Contact implements Serializable;
9  }
10 refines aspect Serialization {
11   declare parents : Feedback implements Serializable;
12 } ...
```

Figure 7.8: Decomposed serialization aspect (excerpt).

Connection Pooling

implementing design decisions

Figure 7.6 depicts the `Pooling` aspect for a basic P2P system. By implementing further design decisions, the definition of the `Pooling` aspect changes – use case 2 (cf. Sec. 6.2.4). Using AR we implemented these design decisions as refinements.

```
1  refines aspect Pooling {
2    pointcut open(ClientSocket sock) : Super.open(sock) &&
3      cflow(execution(void Peer.main(..)));
4  }
5  refines aspect Pooling {
6    Object putPool(ClientConnection con) {
7      synchronized(pool) { return Super.putPool(con); }
8    }
9    ClientConnection getPool(ClientSocket sock) {
10     synchronized(pool) { return Super.getPool(sock); }
11   }
12 }
```

Figure 7.9: Encapsulating design decisions using AR.

Figure 7.9 depicts two refinements (merged in one listing). The first (Lines 1-4) refines the pointcut `open` to limit the matched joint points to those that occurs in the control flow of `Peer`. This excludes join points associated to helper and experimentation classes that use `ClientConnection` objects as well. Pointcut refinement decouples the aspect refinement from a fixed base aspect and thus increases the flexibility to combine this refinement with other refinements.

The second refinement is more sophisticated (Lines 5-12). It refines both advice (`putPool`, `getPool`) with synchronization code to guarantee thread safety. Since the pooling activities are implemented via named advice, this refinement adds synchronization code.

7.2 Statistics

In this section, we present statistics on how and when FOP and AOP mechanisms were used in implementing our P2P product line. These statistics provide insight into the programming guidelines on mechanism usage, which we discuss in detail in Section 7.3.

7.2.1 Statistics on Used AOP and FOP Mechanisms

We collected the following statistics: (1) the number of implementation mechanisms used, (2) the *lines of code (LOC)* associated with these mechanisms, and (3) the LOC associated with static crosscuts (introductions) and dynamic crosscuts (extending methods).

Number of Classes, Mixins, and Aspects

number of aspects sums to 5%

The base P2P framework contains only 2 classes. A fully-configured P2P system consists of 127 classes. Thus, refining the base framework into a fully-configured system required the incremental introduction of 125 classes. In addition to class introductions, there were 120 class refinements implemented as mixins, and 14 aspects were used to modularize crosscutting concerns. The main point is that we used classes and mixins primarily for implementing features rather than aspects, which were used only to a minor degree – about 5% of the overall number of mechanisms for constructing features (Fig. 7.10).

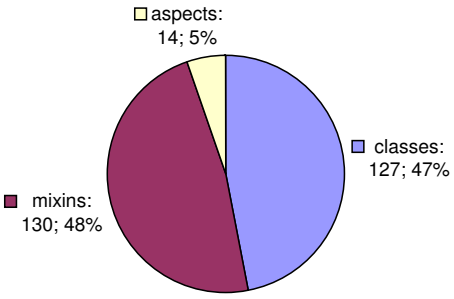


Figure 7.10: Number of classes, mixins, and aspects in P2P-PL.

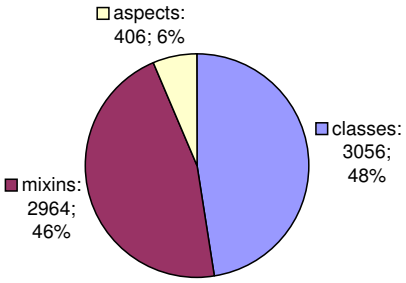


Figure 7.11: LOC of classes, mixins, and aspects in P2P-PL.

LOC Associated With Classes, Mixins, and Aspects

aspect code sums up to 6% of the code base

The overall code base of P2P-PL consists of 6,426 LOC. Of these, 3,056 LOC are associated with classes, 2,964 LOC with mixins, and 406 LOC with aspects and refinements of aspects. These statistics are in line with the numbers given above on the ratio of implementation mechanism usage. Aspect code sums up to 6% and mixin code to 46% of the overall code base (Fig. 7.11).

LOC Associated With Refinements and Introductions

dominant activity of features is introduction

1,488 LOC of all mixins and aspects extend existing methods (dynamic crosscuts). Of these, 374 LOC are associated with AspectJ advice and 1,114 with method extensions via mixins and overriding. The remaining 4,938 LOC are associated with introductions of new functionality (static crosscuts). This suggests that the dominant role of features is to introduce new structures in P2P-PL (77%), rather than extending existing methods (17%) or advising join points (6%) (Fig. 7.12).

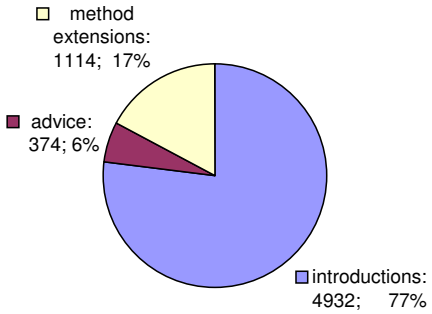


Figure 7.12: LOC of static and dynamic crosscutting in P2P-PL.

7.2.2 Statistics on AFMs with Aspects

Number and Properties of Aspects

most aspects exploit advanced AOP

Of the 14 aspects that were used, 6 modularized homogeneous crosscuts (that refined a set of targets coherently with a single piece of code), 7 aspects implemented advanced dynamic crosscuts (that access dynamic context information, e.g., `cflow`), 2 aspects altered inheritance relationships (that introduce interfaces), and 3 aspects implemented purely heterogeneous crosscuts (Fig. 7.13)[6].

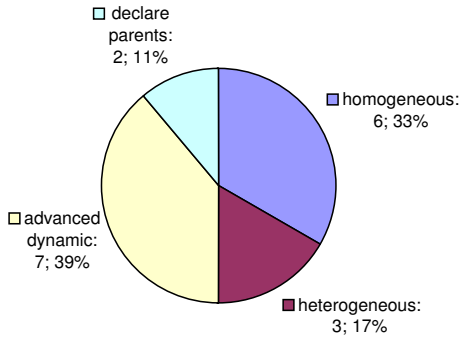


Figure 7.13: Number of crosscuts implemented by aspects.

In summary, 11 of 14 aspects (79%) exploit the advanced capabilities of AOP. Using mixins exclusively would result in replicated, scattered, and tangled workarounds, as explained before. Only 3 aspects implement collaborations that could also be implemented by traditional feature modules. Section 7.3 explains why in these particular cases using aspects was appropriate.

Number of Feature-Related Classes and Mixins

With respect to the question of if aspects are used standalone or with other classes and mixins in concert, we observed that an AFM with one aspect also has 1 to 2 (up to 6) additional classes and mixins. This demonstrates that AFMs in P2P-PL encapsulate collaborations of aspects, classes, and mixins, rather than aspects in isolation.

7.2.3 Statistics on Aspect Refinement

As explained in Section 7.1.2, AR is useful for decomposing and refining aspects. Table 7.2 lists the decomposed aspects and the number of their refinements. On average,

[6] Note that some aspects were counted for more than one category, e.g., homogeneous *and* dynamic.

there were 7 refinements per base aspect and 1/2 of all aspects were candidates for decomposition via AR. While the predominant role of aspect refinement was to add new structural elements, i.e., advice, pointcuts, methods, fields, we refined only 3 named advice and 1 pointcut.

decomposed aspect	**number of refinements**
serialization	11
responding	4
toString	12
log/debug	13
pooling	3
dissemination	3
feedback	2

Table 7.2: Aspects decomposed by AR.

7.3 Lessons Learned

7.3.1 Refinements and Aspects – When to Use What?

many problems could be solved by FOP

A central question for programmers is when to use refinements à la FOP and when to use aspects? What we have learned from our case study is that a wide range of problems can be solved by using object-oriented mechanisms and FOP. Specifically, we used FOP for expressing and refining collaborations of classes. Collaborations are typically heterogeneous crosscuts with respect to a base program. Each added feature module reflects a subset of the structure of the base program (i.e., a sparse version of the class hierarchy of the base program [OH92]) and adds new and refines existing structural elements. As we explained in Chapter 4, a significant body of prior work advocates this view [VN96c, MO04, LLO03, Ern01, OZ05, Ost02, TVJ+01, EOC06, TOHSMS99, BSR04, SB02, Ste00, Ste05, Bos99].

using aspects standalone was not appropriate

Using aspects in isolation for implementing collaboration-based designs, as proposed in [PSR00, HU02, SK03], would not reflect the object-oriented structure we had in mind during the design of P2P-PL. For example, the peer abstraction of P2P-PL plays different roles in different collaborations, e.g., with the network driver and with the data management. Encapsulating these different roles and their collaborations in single aspects would hinder us and others to recognize and understand the inherent object-oriented structure and the meaning of these features. In particular, if a collaboration embraces many roles and they are merged into one (or more) standalone aspect(s), the resulting code would be hard to read and to understand.

beneficial use cases for AOP

Nevertheless, aspects proved to be a useful modularization mechanism. In our study we learned that they help in those situations where traditional OOP and FOP failed:

1. By using aspects and their pattern-matching and wildcard mechanisms for homogeneous crosscuts we could avoid code replication. The aspect-oriented implementation achieves a 5% code reduction compared to an equivalent object-oriented or feature-oriented variant.

2. Aspects helped to express advanced dynamic crosscuts in the implementation of 7 features in P2P-PL. Aspects perform better in this respect than FOP because they provide sophisticated language-level constructs that capture the programmers intension more precisely and intuitively (e.g., `cflow`).

statistical support

Our case study provides statistics on how often AOP and FOP mechanisms are used. AOP mechanisms were used in 12% of all end-user visible features, because they allowed us to avoid code replication, scattering, and tangling. However, aspects occupied only 6% of the code base. This is because standard object-oriented mechanisms were sufficient to implement most features (i.e., 94% of the P2P-PL code base). Using AOP for homogeneous crosscuts we coould achieve a code reduction of 5%.

7.3.2 Borderline Cases

While we understand the above considerations as guidelines for programmers that help in most situations to decide between aspects and refinement mechanisms like mixins and virtual classes, we also discovered a few situations where a decision is not obvious.

alternative implementation of homogeneous crosscuts

We realized that some homogeneous crosscuts could be modularized alternatively by introducing an abstract base class that encapsulates this common behavior. While this works, for example, for introducing an integer field for assigning IDs to different types of messages, it does not work for classes that are completely unrelated, as in the case of a logging feature. It is up to the programmer to decide if the target classes are syntactically and semantically close enough to be grouped via an abstract base class.

alternative implementation of heterogeneous crosscuts

Though our study has shown that a traditional collaboration-based design à la FOP works well for most features, we found at least one heterogeneous feature for which it is not clear if an aspect would not be more intuitive. This feature introduces `toString` methods to a set of classes. Naturally, each of these methods is implemented differently. Thus, the feature is a heterogeneous crosscut. However, in this particular case it seems more intuitive to group all `toString` methods in one aspect. We believe that this is caused by the partly homogeneous nature of this crosscut, i.e., introducing a set of methods for the same purpose to different classes.

7.3.3 Benefits of Aspect Refinement

AR improved customizability

While the application of AR increased the total number of AFMs in P2P-PL considerably, the fine-grained decomposition of aspects (which results in 48 refinements applied to 7 aspects) did not only structure the design and implementation of P2P-PL, but it also increased the configuration space, i.e., the tailored variants that can be derived by the configuration process. For example, the aspect `Serialization` has as many variants as different sets of target classes are possible in P2P-PL (theoretically 2^{10}). The aspect `Pooling` comes in fewer variants (8) because it has only 3 optional refinements, which can be combined freely (2^3).

AR improved reusability

Beside an improvement in customizability we achieved a better reusability of aspect code amongst different variants of P2P-PL. In our study, all derivable variants of aspects share common functionality, thus reusing aspect code. In case of the aspect `Pooling`, each of the 8 variants reuses code of the base aspect and of 0 to 2 refinements. On average, each variant of each of the 7 decomposed aspects reuses code of 1 1/2 aspects and refinements. This is because, for most aspects, all variants can be freely combined, i.e., they are optional and can be applied standalone to their base aspects, in combination with *some* other refinements, or in combination with *all* other refinements.

Finally, it remains to note that we did not find many use cases for advice and pointcut refinement (3 named advice and 1 pointcut refinement). We believe that this small number originates from the refactoring approach we chose, i.e., we decomposed each considered aspect retroactively into a base aspect and several refinements.

7.4 Open Issues

Granularity and Scalability

On average, in P2P-PL each feature is implemented by 56 LOC. Thus, the features of P2P-PL are very fine-grained. Although, we are not aware of guidelines that tell programmers what feature granularity is appropriate, this fine-grained approach might not scale to larger software projects because programmers might get lost in the myriads of features. One way to address this issue would be to implement coarse-grained features, e.g., as in [TBD06]. While this solves the problem of limited scalability, it decreases the potential scenarios a feature can be reused with [Big98]. Remarkably, not aware of this fact when implementing P2P-PL, we chose intuitively an approach in between. As explained in Section 7.1, we organized the set of 113 features into a logical tree structure of subdomains. All these subdomains have counterparts in the domain model of P2P systems. Those subdomains can be understood as large-scale compound features. Such

a hierarchical approach might be a trade-off between fine-grained customizability and scalability.

Code Tangling

During our study a fundamental question emerged: when is an interaction between to feature modules (e.g., class `A` calls a method of class `B`) considered undesirable code tangling? For example, Figure 7.14 depicts a simplified excerpt of the class `Peer` that uses several times the message subsystem for sending messages. We implemented this interaction via direct method calls from the class `Peer` to the class `MessageSender` (Lines 7,14). Moreover, `Peer` uses a logging subsystem to log its current state. This is implemented also via method calls from `Peer` to `Log` (Lines 5,8,12,15).

```
1  class Peer {
2    int id;
3    /* ... */
4    void run() {
5      Log.log("running peer: " + id);
6      /* ... */
7      MessageSender.send(new RequestMessage(this));
8      Log.log("send request: " + id);
9      /* ... */
10   }
11   void startup() {
12     Log.log("startup peer: " + id);
13     /* ... */
14     MessageSender.send(new StartupMessage(this));
15     Log.log("send startup notification: " + id);
16     /* ... */
17   }
18 }
```

Figure 7.14: `Peer` invokes methods of `Log` and `MessageSender`.

What is undesirable tangling?

Most programmers would probably agree that the method calls from the class `Peer` to the class `MessageSender` are not undesirable code tangling, but invoking the method `log` in the class `Log` is considered code tangling. That is, the calls to `Log` should be moved to an aspect, whereas the calls to `MessageSender` should remain in `Peer`. In this particular case it might be easy to decide, but in other situations it might be unclear when to factor a collaboration in an aspect and when not. So what is the general rule for considering a uses-relationship as tangling or as meaningful collaboration?

We believe that the *law of demeter of concerns (LoDC)* may help in this matter [Lie04]. Informally, it states that a concern should only know about concerns that contribute to its functionality. Mapped to our problem it is evident that calling methods of `MessageSender` is necessary for the operation of `Peer`, whereas logging is not required.

Hence, programmers may use the LoDC for deciding when to use aspects and when collaborations of refinements and classes.

7.5 Related Work

Recent studies have applied and evaluated AOP and FOP by their application to larger software projects. We review a representative subset.

AOP Case Studies

Colyer and Clement refactored an application server using aspects [CC04]. 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 did not explore) a strong relationship to FOP. This dissertation demonstrates the useful integration of both worlds.

Zhang and Jacobsen refactored several CORBA ORBs [ZJ04]. Using code metrics, they demonstrate that program complexity could be reduced. They propose an incremental process of refactoring which they call *horizontal decomposition*. Liu et al. point to the close relationship to FOP [LBL06]. Our study confirms that with respect to the implementation of program features, aspects are too small units of modularization [MO04, LLO03].

Coady and Kiczales undertook a retroactive study of aspect evolution in the code of the FreeBSD operating system (200-400 KLOC) [CK03]. They factored 4 concerns and evolved them in three steps; inherent properties of concerns were not explained in detail.

Lohmann et al. examine the applicability of AOP to embedded infrastructure software [LST+06]. They show that AOP mechanisms, if carefully used, do not impose a significant overhead. In their study they factored 3 concerns of a commercial embedded operating system; 2 concerns were homogeneous and 1 heterogeneous. They show that aspects are useful for encapsulating design decisions, which is also confirmed by our study.

FOP Case Studies

A significant body of research supports the success of FOP in the implementation of large-scale applications, e.g., for the domain of network software [BO92], databases [BT97, LAS05, BO92], avionics [BCGS95], and command-and-control simulators [BJMvH02], to mention a few. The AHEAD tool suite is the largest example with about 80-200 KLOC [BSR04, TBD06]. However, none of these studies make quantitative statements

about the properties of the implemented features, nor do they evaluate the implementation mechanisms used with respect to the structures of the concerns. The features they consider are traditional collaborations with heterogeneous crosscuts, which is in line with our findings in P2P-PL.

Lopez-Herrejon et al. explore the ability of AOP to implement product lines in a FOP and SWD fashion [LH06, LHB06]. They illustrate how collaborations are translated automatically to aspects. They do not address in what situations which implementation technique is most appropriate nor how the aspects generated affect program comprehensibility.

Xin et al. evaluate *Jiazzi* and AspectJ for feature-oriented decomposition [XMEH04]. They reimplemented an AspectJ-based CORBA event service [HC02] by replacing aspects with *Jiazzi units*, which are a form of feature modules. They conclude that Jiazzi provides better support for structuring software and manipulating features, while AspectJ is more suitable for manipulating existing Java code in unanticipated ways. However, they do not examine the structure of the implemented features. Their success in implementing all features of their case study using Jiazzi feature modules hints that most of them (if not all) come in form of object-oriented collaborations.

We are not aware of further published studies that take both, AOP and FOP into account.

7.6 Summary

17% method extensions; 6% aspect code

Our conducted study demonstrated the practical applicability of the integration of AOP and FOP. We observed that the dominant role of features is the introduction of new structural elements – adding new classes and new members to existing classes. Refinement of existing methods involved a small fraction of features in our case study (17%). This is in line with prior studies [LH06, LHB06]. Further, while aspects were used infrequently (6% of the code base), they enhanced the crosscutting modularity of features and reduced code replication. That is, using aspects or refinements in isolation would not have achieved an elegant design or implementation.

Are features predominantly collaborations?

The result of our case study is a first data point. Although we cannot generalize of a single study, we believe this work supports the hypothesis that object-oriented collaborations (expressed by classes and mixins) define the predominant way in which concerns (features) are implemented, where aspects are useful in expressing homogeneous and advanced dynamic crosscuts. In the next chapter we address this issue in more depth.

Regarding AR we observed that 1/2 of all aspects in our study could be decomposed to separate design decisions or to decouple aspects from the details of the base program (i.e., to synthesize tailored aspects). The composability of aspects and their refinements

increased the configuration space and facilitated aspect code reuse between aspect variants, which are tailored to different program contexts.

In summary, our case study provides supporting evidence that our programming guidelines can assist programmers in choosing and using the right implementation mechanism for the right problem. In the next chapter we provide further support for our guidelines by means of analyzing further studies implemented by others.

CHAPTER 8

Aspects Versus Collaborations

This chapter shares material with the AOPLE'06 paper 'On the Structure of Crosscutting Concerns: Using Aspects or Collaborations?' [ABR06].

We have shown how integrating AOP and FOP can overcome their individual limitations. Our case study supports our claims for a particular software project. We observed that our programming guidelines are reasonable for this example, but also that aspects and collaborations have been applied to different extents. However, this is only one data point and furthermore it may be biased – even against our best will.

In this chapter we revisit the question of when to use what mechanism and how implementation techniques are used today. We formulate a problem statement that serves as a starting point for our investigations. Subsequently, we present a set of code metrics for analyzing programs based on AOP. Finally, we apply our metrics to a set of case studies implemented by others. Based on this, we can make stronger claims about the issues regarding the current practice of programming with collaborations and aspects.

8.1 Problem Statement: Aspects vs. Collaborations

Aspects and collaborations overlap in their capabilities to solve certain design and implementation problems. Our derived programming guidelines reflect this fact and assist the programmer in choosing the right programming technique for the right problem:

1. Collaborations are heterogeneous crosscuts and should be abstracted explicitly, e.g., by feature modules [BSR04] or related mechanisms [OZ05, AGMO06, LLO03, Her02, TVJ+01, Ost02].

2. Aspects should be used in the remaining cases, i.e., for homogeneous crosscuts and advanced dynamic crosscuts.

These guidelines are not arbitrary at all, but were inferred from the individual strengths of aspects and collaboration abstraction mechanisms. They build on a long line of work on OOP and collaboration-based designs [RAB+92, GHJV95, VN96c, LLO03, MO04, SB02, BSR04, MMP89, Ern01, EOC06, OZ05, Ste00, NQM06, NCM04, BDN05, CL01] and follow the initial idea of AOP, namely to implement only those concerns as aspects for which the underlying modularization mechanisms fail [KLM+97, EFB01].

AOP filled a vacuum

However, we are aware that although the concept of collaborations predates AOP by quite some time, mainstream OOP languages have been slow in supporting these constructs. AOP has filled the vacuum and offered some programming mechanisms that remain controversial [Lie04, Ste06, Ale03] and that may lead to serious penalties, e.g., unpredictable program behavior [MA05, DFS04, LHBL06], weak modularity [GSF+05, GSC+03] and decreased evolvability [Lie04, LHBL06, GB03]. Furthermore, the weak support of collaborations and related mechanisms (e.g., *virtual classes* [MMP89, EOC06, OZ05], *mixins* [BC90, SB02], *nested inheritance* [NCM04, NQM06], *classboxes* [BDN05]) has contributed to a confusion regarding their relationship to crosscutting concerns, which we have addressed in Chapter 3.

Do aspects implement collaborations?

The aim of this chapter is to explore whether aspects used today really implement AOP-specific problems or implement in fact collaborations, which could have been implemented by languages that support collaborations. With the advent of languages that support collaborations (e.g., *Scala* [OZ05], *Jiazzi* [MFH01], *Jx* [NCM04], *J&* [NQM06], *Classbox/J* [BDN05], *Jak* [BSR04], *ContextJ* [CHdM06], *Lasagne/J* [TVJ+01], *CaesarJ* [AGMO06], *Aspectual Collaborations* [LLO03], *Object Teams* [Her02], *Aspectual Feature Modules*) the question of whether and how aspects should be replaced by collaboration abstraction mechanisms arises. Furthermore, we are interested in which design and implementation problems remain for aspects – beyond collaborations.

To answer these questions we have analyzed a set of AspectJ programs. In order to quantify the application of aspects we propose a set of code metrics for aspect-oriented programs.

8.2 Metrics

A software metric is a measure of some property of a piece of software or its specification. The metrics we propose target exclusively the issues discussed above, namely the question in which of our two categories (collaboration or not) a given aspect falls. Specifically, we are interested in the following metrics:

- What fraction of a program's code base is occupied by classes, interfaces, and aspects?
- What fraction of inter-type declarations and advice is heterogeneous and homogeneous?
- What fraction of advice is basic and advanced?

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)

This metric determines the NOO of classes, interfaces, and aspects, as well as the LOC associated with them. It tells us whether the number of aspects (as opposed to classes and interfaces) is a small or a large fraction of the modularization mechanisms used in a program, and whether aspects implement a significant or only a small part of the code base. However, the CIA metric does not tell us how aspects are used, e.g., how often advanced advice is used in an aspect as opposed to basic advice and methods. This is where further metrics come into play.

Heterogeneous and Homogeneous Crosscuts (HHC)

The HHC metric explores to what extent aspects implement homogeneous and heterogeneous crosscuts. Specifically, we determine the fraction of advice and inter-type declarations that implement heterogeneous and homogeneous crosscuts (NOO) and the fraction of the code base that is associated with them (LOC). The HHC metric tells us whether the implemented aspects take advantage of the wildcard and pattern-matching mechanisms of AOP (homogeneous crosscuts) or merely emulate OOP mechanisms (heterogeneous crosscuts).

Basic and Advanced Dynamic Crosscuts (BAC)

The BAC metric determines the NOO of pieces of basic and advanced advice and the overall LOC associated with them. This metric tells us to what extent the aspects of a program take advantage of the advanced capabilities of AOP for implementing dynamic crosscuts. Basic advice can be implemented as method extensions via overriding.

8.2.1 Analyzing AspectJ Programs

As our metrics are language-independent, we now explain how to collect statistics for AspectJ programs.

CIA Metric

Collecting data for the CIA metric is straightforward: we simply traverse all source files included in a given project and count the NOO and LOC of aspects, classes, and interfaces. Upfront we eliminate blank lines and comments.

HHC Metric

Homogeneous crosscuts affect multiple join points by applying identical code. Typically this can be recognized syntactically by advice and inter-type declarations that have wildcards (i.e., `*`, `+`, and `..`) and disjunctions (e.g., '`execution(/*...*/) || execution(/*...*/)`' or '`declare parents : (Line || Point) implements Shape`'). Furthermore, advice is considered homogeneous that does not qualify a target method or field completely, e.g., by omitting the declaring type: '`execution(void print())`'. All remaining pieces of advice and inter-type declarations are considered heterogeneous.

BAC Metric

We consider all pieces of advice as advanced except those associated with `call`[1] and `execution` and that are not combined with any other pointcuts, except with `target` and `args`[2]. This is an overestimation: it might consider some pieces of advice advanced that is not, but not vice versa. However, our studies show that this does not affect the results, since we found very few pieces of advanced advice, even with this overestimation. The remaining advice is considered basic.

8.2.2 AJStats: A Statistics Collector for AspectJ Programs

For collecting statistics of AspectJ programs we developed a tool, called *AJStats*[3]. The core of AJStats is an AspectJ parser that is generated by means of a JavaCC grammar,

[1] Although the semantics of `call` is to advise the client side invocations of a method, it can be implemented as method extension – provided that *all* calls to the target method are advised.

[2] The pointcut `execution` can be combined also with the pointcut `this`.

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

borrowed from [FLG06]. AJStats analyzes AspectJ programs and collects the following data (measured in NOO and LOC):

- classes, aspects, interfaces (distinguishes between top-level and nested)
- methods, constructors, fields (distinguishes between classes, interfaces, and aspects)
- pointcuts, advice
- inter-type declarations (field, methods, constructors, others)

AJStats does not identify homogeneous and advanced dynamic crosscuts. In order to do so one has to examine the code by hand.

Figure 8.1 shows a screen snapshot of the output of AJStats after analysis of an arbitrary AspectJ program.

8.3 Case Studies

There are not many published, non-trivial studies on AspectJ in the open literature. We analyze a diverse selection of small-sized, medium-sized, and large-sized programs that we were able to locate. We did not include P2P-PL and other programs of our own because we did not want to bias the results.

8.3.1 Overview of the Analyzed AspectJ Programs

The first 5 case studies are small and medium-sized AspectJ programs (< 20 KLOC); the last 3 are large-sized AspectJ programs (> 20 KLOC).

Tetris: The Game

Tetris is the implementation of the popular game in AspectJ. It was developed at the Blekinge Institute of Technology in Sweden. The source code is available publicly at the project Web site[4]. The code base of Tetris is 1,030 LOC. It implements features such as a GUI, various game levels, or block management.

OAS: An Online Auction System

OAS (Online Auction System) is a system that allows people to negotiate the purchase and sale of goods in the form of English-style auctions (over the Internet). OAS was

[4] http://www.guzzzt.com/coding/aspecttetris.shtml

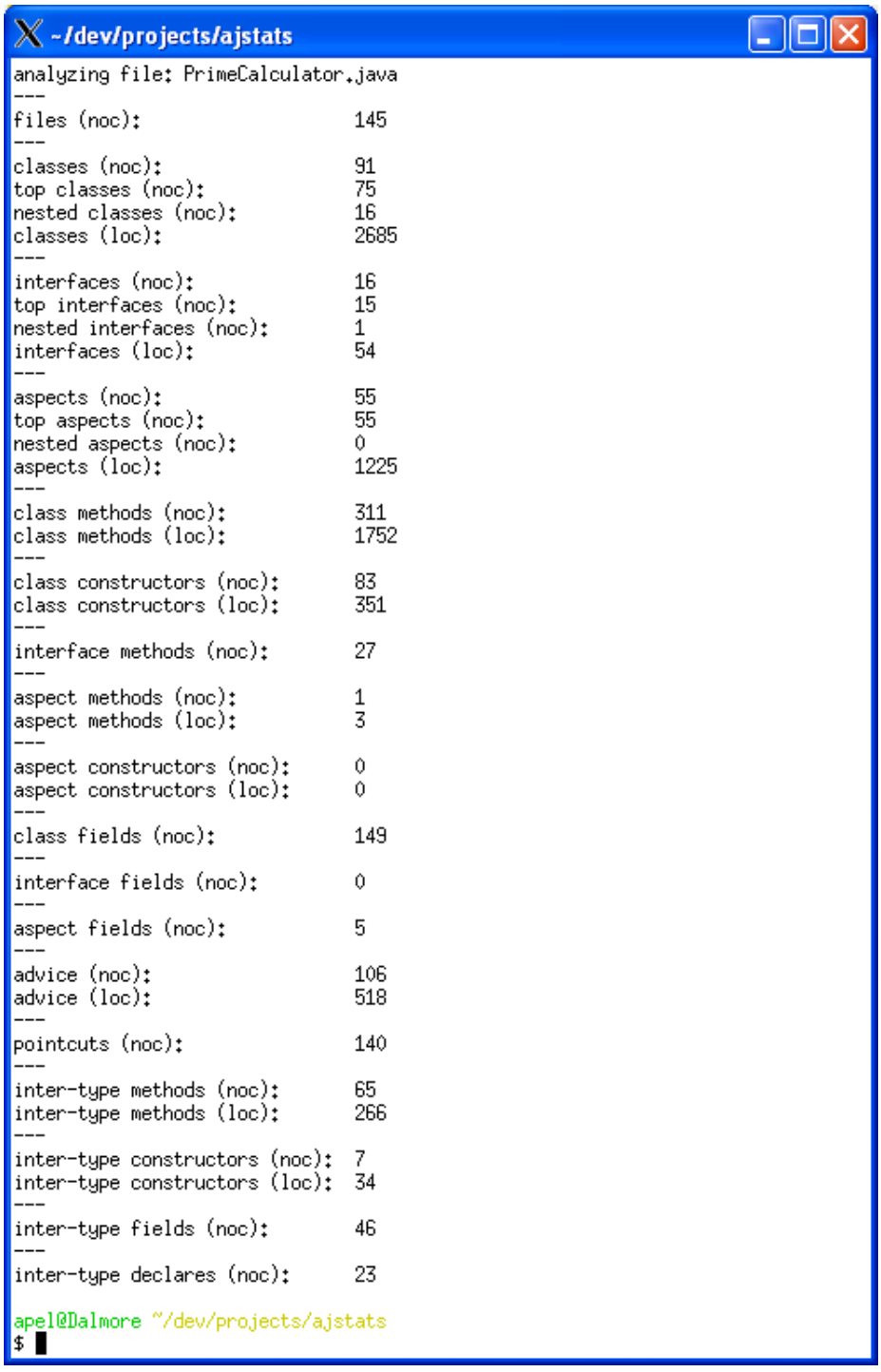


Figure 8.1: AJStats Screen Snapshot.

developed from scratch using AspectJ at the Lancaster University. The source code was released kindly by Awais Rashid. The code base of OAS is 1,623 LOC. OAS does not employ a special notion of features. Nevertheless it factors functionality such as a GUI, serialization, as well as auction, user, and bidding management.

Prevayler: Transparent Persistence for Java

Prevayler is a Java application that implements transparent persistence for Java objects. It is a fully functional main memory database system in which business objects may persist. Prevayler was refactored by the University of Toronto using AspectJ and *horizontal decomposition* [GJ05, ZJ04]. Successively, a series of features has been detached and encapsulated into aspects. Example features are persistence, transaction, query, and replication management. The refactored AspectJ source code is available at the project Web site[5]. The code base of Prevayler is 3,964 LOC subdivided into 18 features.

AODP: Aspect-Oriented Implementation of the GoF Design Patterns

AODP (Aspect-Oriented Design Patterns) is an AspectJ implementation of the GoF (Gang-of-Four) design patterns [GHJV95], implemented at the University of British Columbia [HK02]. The programmers of AODP restructured several design patterns using AspectJ and separated the reusable parts of aspects and classes. The AspectJ implementation can be obtained at the project Web site[6]. The overall code base consists of 3,995 LOC subdivided into 23 features, which are the different design pattern instances.

FACET: An Aspect-Based CORBA Event Channel

FACET (Framework for Aspect Composition for an EvenT channel) is an AspectJ implementation of a CORBA event channel, developed at the Washington University [HC02]. The source code is available publicly at the project Web site[7]. The goal of the FACET project is to investigate the development of customizable middleware using AOP. FACET implements a real-time event channel in Java and AspectJ, modeled after the *TAO Real-time Event Channel* [SLM98]. The code base of FACET is 6,364 LOC subdivided into 34 features. Features in FACET are for example different event types, synchronization, a CORBA core, or tracing.

[5] http://www.msrg.utoronto.ca/code/RefactoredPrevaylerSystem/
[6] http://www.cs.ubc.ca/~jan/AODPs/
[7] http://www.cs.wustl.edu/~doc/RandD/PCES/facet/

AJHotDraw: A 2D Graphics Framework

AJHotDraw is an aspect-oriented refactoring of the JHotDraw two-dimensional graphics framework. It is an open source software project hosted by the *SourceForge.net* open source development Web portal. The code is publicly available at the AJHotDraw project Web site[8]. The code base of AJHotDraw is 22,104 LOC. It provides numerous features for drawing and manipulating graphical, planar objects.

Hypercast: A Multicast Overlay Network Protocol

Hypercast is an implementation of a protocol for multicast overlay network communication. It was developed at the University of Virginia in cooperation with the Microsoft Corporation [LB99]. The original object-oriented implementation was refactored using AspectJ and *crosscutting interfaces* [GSS+06]. The source code was released kindly by Yuanyuan Song and Kevin Sullivan. The code base of the aspect-oriented implementation of Hypercast is 67,260 LOC. Example features of Hypercast are different base protocols (UDP, TCP, HTTP), encryption, or message handling.

Orbacus: A CORBA Middleware Framework

Orbacus is a mature CORBA-compliant middleware product that has been deployed by IONA Technologies[9]. It has been used successfully in mission-critical systems in the telecommunications, finance, government, defense, aerospace and transportation industries. We consider the AspectJ-based version of Orbacus (a.k.a. *Abacus*) developed by refactoring at the University of Toronto [ZJ04, ZGJ05]. The source code was released kindly by Charles Zhang and Hans-Arno Jacobsen. The code base of the AspectJ version of Orbacus is 129,897 LOC. Orbacus is a complex software with numerous features, e.g., dynamic invocation interface, event handling, encoding conversation.

8.4 Statistics

We used AJStats for collecting the statistics. We identified homogeneous advice and inter-type declarations as well as advanced advice by hand, i.e., we examined the code manually. This method revealed an interesting issue: we identified advice and inter-type declarations that have patterns and wildcards in their pointcut expressions but that do not affect multiple join points. For example, advice '`after() : call(* foo(..))`' is

[8] http://sourceforge.net/projects/ajhotdraw/
[9] http://www.orbacus.com/

	Tetris		OAS		Prevayler		AODP	
	NOO	*LOC*	*NOO*	*LOC*	*NOO*	*LOC*	*NOO*	*LOC*
features / code base	6	1030	1	1623	18	3964	23	3995
classes + interfaces	9	818	21	1283	107	2739	244	3241
aspects	8	212	9	340	55	1225	41	754
java fields	81	81	64	64	149	149	149	149
java methods	47	583	149	1042	338	1779	432	2010
java constructors	7	116	37	137	83	351	76	334
aspect fields	17	17	20	20	24	24	16	16
aspect methods	1	11	6	78	1	3	39	205
aspect constructors	0	0	0	0	0	0	0	0
itd fields	0	0	0	0	27	27	2	2
itd methods	0	0	2	15	65	266	41	182
itd constructors	0	0	0	0	7	34	0	0
itd declare	2	2	8	8	23	23	37	37
advice	21	145	20	141	106	518	15	94
hom. advice	0	0	15	61	10	52	5	50
hom. itds	2	2	8	8	3	7	7	7
advanced advice	2	12	4	25	30	136	3	30
basic advice	19	133	16	116	76	382	12	64
het. crosscuts	21	145	7	95	215	809	83	258

	FACET		AJHotDraw		Hypercast		Orbacus	
	NOO	*LOC*	*NOO*	*LOC*	*NOO*	*LOC*	*NOO*	*LOC*
features / code base	34	6364	13	22104	10	67260	30	129897
classes + interfaces	181	5143	351	21909	328	67142	1894	118938
aspects	113	1221	10	195	12	118	125	10959
java fields	198	198	712	712	2691	2691	3180	3180
java methods	340	2936	2850	15937	3122	52130	7659	89642
java constructors	88	375	356	1461	383	6879	1447	7219
aspect fields	3	3	0	0	8	8	33	33
aspect methods	57	187	0	0	0	0	19	139
aspect constructors	0	0	0	0	0	0	2	16
itd fields	22	22	1	1	0	0	63	63
itd methods	52	229	20	121	0	0	460	4036
itd constructors	3	12	0	0	0	0	0	0
itd declare	34	34	10	10	0	0	4	4
advice	49	297	5	19	8	27	289	4748
hom. advice	4	16	3	11	8	27	14	209
hom. itds	8	8	1	6	0	0	0	0
advanced advice	11	110	3	12	2	8	53	488
basic advice	38	187	2	7	6	19	236	4260
het. crosscuts	148	570	32	134	0	0	802	8642

Table 8.1: Collected data of the analyzed case studies.

formally a homogeneous advice but if there is only one method `foo` in the base program it is in fact heterogeneous. We address this issue in more depth in Section 8.5.

Table 8.1 lists the data we collected from the AspectJ programs. Especially, interesting for our analysis are the rows *classes + interfaces*, *aspects*, *hom. advice*, *hom. itds*, and *advanced advice*. In the following paragraphs we discuss the data in depth.

CIA Metric

Since the projects analyzed are of different size (1 KLOC – 130 KLOC) the number of classes, interfaces, and aspects varies as well. The spectrum of the number of classes and interfaces ranges from 9 to 1,894 and the number of aspects from 9 to 125. The LOC of classes and interfaces ranges from 818 to 118,938 LOC and the LOC of aspects from 118 to 10,959 LOC. Figure 8.2 illustrates that AOP has been used to different extents (0.2% to 31% of the code bases). Especially in the small-sized and medium-sized projects (< 20 KLOC) aspects occupy a significant part of the code base (19% – 31%); in the larger projects (> 20 KLOC) aspects occupy a smaller fraction (0.2% – 8%).

	classes + interfaces	**aspects**
Tetris	79.4 %	20.6 %
OAS	79.1 %	20.9 %
Prevayler	69.1 %	30.9 %
AODP	81.1 %	18.9 %
FACET	80.8 %	19.2 %
AJHotDraw	99.1 %	0.9 %
Hypercast	99.8 %	0.2 %
Orbacus	91.6 %	8.4 %

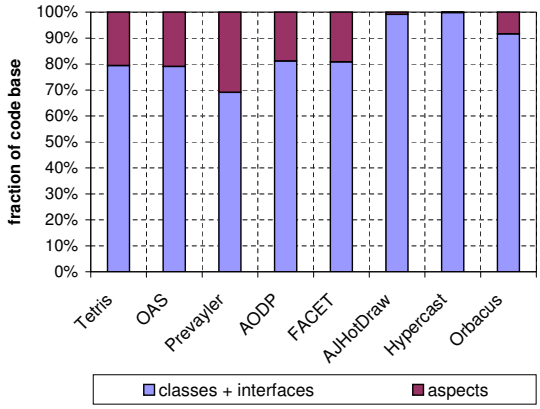


Figure 8.2: NOO and LOC of classes, interfaces, and aspects.

HHC Metric

Homogeneous crosscuts have been used in the analyzed programs to different extents: the spectrum ranges from 2 to 209 LOC associated with homogeneous advice and inter-type declarations. That is, we found 0.04% to 4.3% of the code base implementing homogeneous crosscuts (Fig. 8.3). Note that the 4.3% comes from the second smallest program (OAS). We revisit this issue in Section 8.5.

In contrast to homogeneous crosscuts, we found 0 to 8,642 LOC implement heterogeneous advice and inter-type declarations, which are 0% to 20% of the code base.

	heterogeneous	homogeneous
Tetris	14.1 %	0.2 %
OAS	5.9 %	4.3 %
Prevayler	20.4 %	1.5 %
AODP	6.5 %	1.4 %
FACET	9.0 %	0.4 %
AJHotDraw	0.6 %	0.1 %
Hypercast	0.0 %	0.04 %
Orbacus	6.7 %	0.2 %

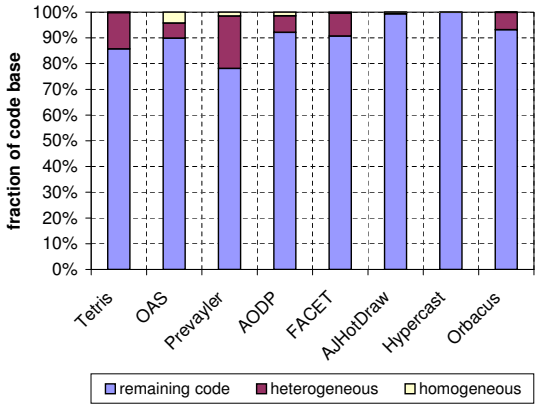


Figure 8.3: NOO and LOC of heterogeneous and homogeneous crosscuts.

BAC Metric

Advanced dynamic crosscuts have been used to different extents in the analyzed AspectJ programs. The spectrum ranges from 8 to 488 LOC, which sums up to 0.01% to 3.4% (Fig. 8.4). The highest percentage comes from Prevayler, a mediums-sized program.

In contrast to advanced advice, we found 7 to 4,260 LOC implement basic advice, which sums up to 0.03% to 13% of the code base.

	basic advice	advanced advice
Tetris	12.9 %	1.2 %
OAS	7.1 %	1.5 %
Prevayler	9.6 %	3.4 %
AODP	1.6 %	0.8 %
FACET	2.9 %	1.7 %
AJHotDraw	0.03 %	0.1 %
Hypercast	0.03 %	0.01 %
Orbacus	3.3 %	0.4 %

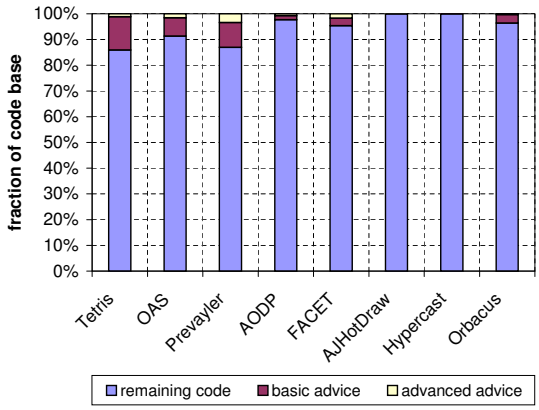


Figure 8.4: NOO and LOC of basic and advanced advice.

8.5 Discussion

advanced AOP vs. collaborations

Figure 8.5 depicts the fractions of the code base of the AspectJ programs analyzed by us that demand advanced AOP mechanisms and that require only OOP and collaboration abstraction mechanisms. Note that the fractions that require AOP are not calculated by simply adding the code associated with homogeneous advice & inter-type declarations and advanced advice together. This is because sometimes advanced advice is also homogeneous (e.g., '`after() returning: call(* foo(..)) && cflow(execution(* bar(..))`').

	collaborations	advanced AOP
Tetris	98.6 %	1.4 %
OAS	94.5 %	5.5 %
Prevayler	95.1 %	4.9 %
AODP	98.4 %	1.6 %
FACET	97.9 %	2.1 %
AJHotDraw	99.9 %	0.1 %
Hypercast	99.96 %	0.04 %
Orbacus	99.5 %	0.5 %

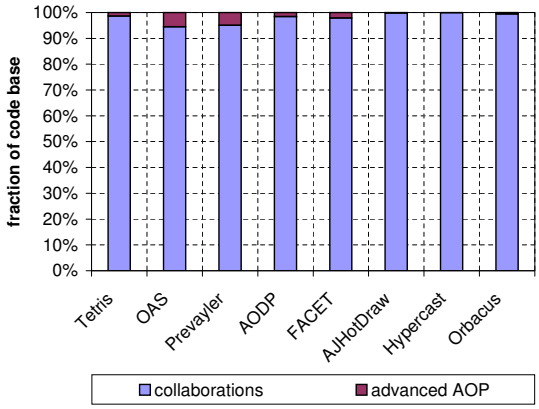


Figure 8.5: Fractions of code that require (1) advanced AOP and (2) OOP and collaboration abstraction mechanisms.

In summary, the spectrum of the fractions of the code base that exploits advanced AOP mechanisms ranges from 0.04% to 5.5%, where the small-sized and medium-sized programs have the largest fractions (1.4% – 5.5%) and the large-sized programs have the smallest fractions (0.04% – 0.5%).

Interpretation of the Data

2% of the code exploits advanced AOP

A major insight gained from the statistics is that only *a minor fraction of the code base (on average 2%) of the analyzed AspectJ programs exploits the advanced capabilities of AOP*, i.e., homogeneous and advanced dynamic crosscuts. This also means that on average *98% of the code base implements collaborations.*

the larger the code base, the lesser AOP is been used

A further interesting outcome is that there seems to be a correlation between the extent of advanced AOP in a program and the size of its code base. In our analyzed programs, we observed that the larger the code base, the smaller the fraction of advanced AOP. While the small-sized and medium-sized programs (< 20 KLOC) use some AOP mechanisms

(on average 3%), the large-sized programs (> 20 KLOC) virtually do not make any use of advanced AOP (on average 0.2%).

three suspicions

These statistics suggest that *the impact of advanced AOP mechanisms decreases as the program size increases*. Large programs use virtually no advanced AOP but exclusively OOP and collaborations. Though we have no definitive answer to the question of why there is an inversely proportional correlation between program size and impact of AOP, we have three suspicions:

1. The impact of AOP in large-sized programs is negligible because it is certainly harder to understand a large-sized program than a small-sized program. This circumstance may be responsible for why the programmers were not able to discover homogeneous and advanced dynamic crosscuts in large-sized programs. Tool support for discovering aspects automatically could help to assist the programmer, e.g., aspect mining tools [BK04, MvDM04, TC04] and clone detection tools [BvDvET05, LLM06, BYM+98, FR99, LPM+97, Bak95].

2. The impact of AOP in large-sized programs is negligible because these programs have not been developed with AOP in mind. All of them have been constructed via a refactorization of object-oriented code into aspect-oriented code. It may be that the programmers simply stopped using AOP after having detached a reasonable number of aspects. Thus, the ratio of aspect code and object-oriented code differs in small-sized and large-sized programs. The development of aspect-oriented large-sized programs from scratch might confirm this conjecture.

3. The impact of AOP in large-sized programs is negligible because the design and implementation problems that occur in large-sized programs are predominantly collaborations. This could be explained by the sheer complexity of these problems that is incompatible with the generic character of homogeneous crosscuts. That is, it is really hard to find problems in large-sized programs that affect many join points and that do the same thing at all points. The same might be true for advanced dynamic crosscuts.

Code reduction

AOP reduces code replication

We have argued that aspects are useful for reducing code replication in a program. Imagine an aspect that advises 100 join points and executes at each join point 10 lines of code encapsulated in one piece of advice. Compared to an OOP equivalent, this aspect would reduce the code size by approximately 990 lines of code. This benefit is not reflected in our metrics and statistics. An aspect-oriented program may have only a few pieces of advice and reduce code replication significantly.

In order to explore this issue, we analyzed for all the considered AspectJ programs, the reduction of code replication achieved by using aspects for modularizing homogeneous crosscutting concerns.

	code reduction
Tetris	0.0 %
OAS	22.0 %
Prevayler	2.7 %
AODP	1.7 %
FACET	7.7 %
AJHotDraw	0.2 %
Hypercast	0.1 %
Orbacus	0.5 %

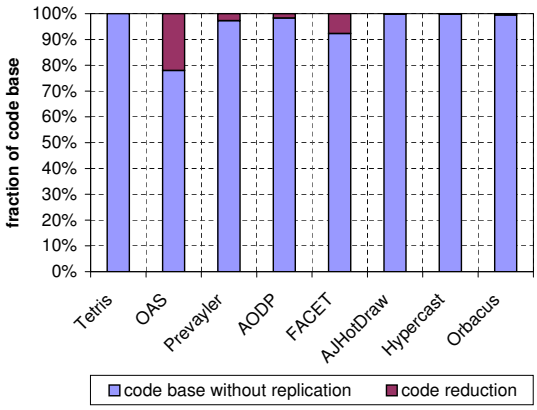


Figure 8.6: Code reduction achieved by using AOP.

small-sized vs. large-sized programs

Figure 8.6 shows different degrees of code reduction achieved by using AspectJ instead of Java in the analyzed programs. In OAS the code size is reduced by 22% compared to an OOP equivalent; in FACET a reduction of 7.7% has been achieved; all other code reductions are 3% and below. It is interesting that the second smallest program achieves the highest degree of code reduction (OAS; 22%). However, *the ability of AOP to the reduce code decreases as the program size increases.* While we observed a significant code reduction (on average 7%) in the small-sized and medium-sized programs, we observed almost no reduction (on average 0.3%) in the large-sized programs.

impact on large-sized programs

The reason why the benefit of using AOP in large-sized programs is so marginal might be that aspects have not been used to the same extent as in small-sized programs. When the impact of aspects in large-sized programs increases then it is reasonable to expect a reduction of code replication – similar to the one in small-sized programs (7%).

4% code reduction through AOP

Nevertheless, the observed code reduction of on average 4% confirms our programming guidelines: use AOP for homogeneous crosscuts because you can avoid code replication. Though 4% may seem to be a marginal benefit, it has been observed that any kind of code replication may lead to serious maintenance problems [Bak95, LPM+97, BYM+98, FR99, LLM06]. Furthermore, this result is in line with prior work on clone detection that conjectures that 5% to 15% of large software projects are clones, i.e., replicated code fragments [Bak95, LPM+97, BYM+98].

Misuse of Wildcards

wildcards are used for single join points

A further observation of our study is that the programmers of the analyzed AspectJ programs used wildcards to match sets of join points (homogeneous crosscuts). We discovered that sometimes these wildcard-based pointcuts do not match multiple join points, but each pointcut matches exactly one join point (heterogeneous crosscuts). It follows that identifying wildcards in pointcut expressions of an AspectJ program does not indicate how many homogeneous crosscuts were implemented because they may match single join points only. We suspect two possible reasons for this: (1) the programmers anticipated features to be added subsequently to the programs, or (2) they used wildcard because they are a 'convenient' way to save time and coding effort.

Regarding the second alternative it remains to note that this programming style may come at a high price [Ale03]. Programmers may get lost easily when adding new features because it may be hard to figure out whether all the pointcuts of the program affect the *correct* sets of join points after this change [LHBL06].

8.6 Related Work

There is some related work on a quantification of the use of AOP via code metrics.

Zhang and Jacobson use a set of object-oriented metrics to quantify the program complexity reduction when applying AOP to middleware systems [ZJ03, ZJ04]. They show that refactoring a middleware system (23 KLOC code base) into aspects reduces the complexity (quantified by McCabe's cyclomatic complexity) and results in a code reduction of 2% – 3%, which is in line with our results.

Garcia et al. analyzed and compared several aspect-oriented programs (4 KLOC – 7 KLOC code bases) and their object-oriented counterparts [GSF+05, KSG+06]. They observe that the aspect-oriented variants exhibited superior stability and reusability through the changes, as it has resulted in fewer lines of code (12% code reduction), etc.

Benn et al. apply the metrics of Garcia et al. to a distributed computing application (0.7 KLOC code base) [BCP+05]. They observe a code reduction of 11% of the aspect-refactored variant compared to an OOP equivalent.

Zhao and Xu propose several metrics for aspect cohesion based on aspect dependency graphs [ZX04]. Ceccato and Tonella propose metrics for measuring the coupling degree between program elements [CT04]. To our knowledge, they did not evaluate their metrics by a case study. Gelinas et al. discuss previous work on cohesion metrics and propose a novel approach based on dependencies between aspect members [GBB06]. They evaluate different metrics by three small-sized and medium-sized case studies (< 7 KLOC).

All of the above proposals and case studies take neither the structure of crosscutting concerns nor the difference between collaborations and other concerns into account.

Lopez-Herrejon and the author propose a set of code metrics for analyzing the crosscutting structure of aspect-based product line features [LHA07]. However, this work focuses exclusively on homogeneous and heterogeneous crosscutting concerns. It does not consider elementary crosscuts but analyzes crosscutting properties of entire features, which may have a substantial size. This way, the crosscutting structure of a feature can be homogeneous, heterogeneous, or any value in between the spectrum of both. We applied these metrics to a large-scale case study (200 KLOC) and observed that virtually every feature was predominantly heterogeneous.

8.7 Summary and Perspective

What fraction of aspects implements collaborations?

The motivation for our study was to determine the fraction of aspects that have been used to implement collaborations. The background is that there are two classes of modularization mechanisms for crosscutting concerns: (1) collaboration abstraction mechanisms and (2) aspect-oriented mechanisms. Due to the missing support for collaborations in contemporary mainstream programming languages, aspects are frequently used to implement collaborations, which we identified as one category of crosscutting concerns (cf. Chapter 3). However, with the advent of collaboration abstraction mechanisms (e.g., classboxes, nested inheritance, virtual classes, delegation layers, AFMs) it stands to question how many of these aspects implement collaborations, and how many are used for alternative use cases beyond collaborations, i.e., homogeneous and advanced dynamic crosscuts.

2% of the code bases is associated to advanced AOP

To address this issue we analyzed a set of AspectJ programs available publicly, which range from small-sized and medium-sized (1 KLOC – 6 KLOC) to large-sized AspectJ programs (20 KLOC – 120 KLOC). We found that in these programs on average 2% of the code bases is associated with advanced AOP; 98% is associated with collaborations and OOP. This result is in line with our experience and the experience of others, who distiguish between aspects and collaborations [LH06, LHB06].

AOP vs. advanced AOP

It is worth noting that the fraction of 2% is in contrast to the real use of AOP mechanisms in the analyzed programs, which is on average 15%. This result leads us to conclude that, given an appropriate support of implementing and composing collaborations, collaboration abstraction mechanisms can replace traditional aspects to a significant extent in contemporary aspect-oriented programs. That is, 13% of the code base of the analyzed AspectJ programs is associated with aspects that implement collaborations and that should be implemented using languages that support collaborations.

impact of program size

Furthermore, we revealed an inversely proportional correlation between program size and the impact of advanced AOP. This is remarkable since we expected a constant percentage of advanced AOP code without any dependence on the program size. Though we have no definitive answer for why this is the case, we have three possible suggestions, which largely build on the fact that the increasing complexity in large-sized program prevents programmers to discover or to implement aspects that are not collaborations. In any case, we conjecture that a fraction of around 5% is a typical upper limit for the use of advanced AOP.

use cases for AOP

Nevertheless, AOP should not be avoided completely. In this dissertation, we condensed two reasons why one should use AOP: (1) when modularizing homogeneous crosscutting concerns a code reduction can be achieved (on average 4% in our analysis) and (2) advanced dynamic crosscuts can be expressed more intuitively, at a higher level of abstraction (on average 1% of the code bases in our analysis). Our analysis of AspectJ programs supports our belief that AFM are an appropriate approach to implement such software projects because they integrate collaboration-based design and AOP, which are both necessary for certain design and implementation problems.

misuse of wildcards

Finally, our study revealed that sometimes the powerful AOP mechanisms, i.e., wildcards in pointcut expressions were used without any benefit. It has been argued that this may lead to serious problems regarding reliability and evolvability [LHBL06, Ale03]. We argue that our programming guidelines help avoiding such misuse of AOP since they point programmers to this problem and assist them to choose the right technique for the right problem.

CHAPTER 9

Concluding Remarks and Further Work

problem area

The principles of separation of concerns and modularity aim at solving problems associated with the software crisis, i.e., canceled projects, projects running over-time, projects running over-budget, etc. Though in the recent years significant progress has been made, the current situation in software development is far from adequate. According to the most recent Standish Group report, only 34% of all software projects are successful.

aim of the dissertation

This dissertation aspires to contribute to this line of research by analyzing, explaining, combining, and devising conceptual, methodical, practical, and tool-related means to improve separation of concerns and modularity in software. Specifically, we focus on two programming paradigms, FOP and AOP that have been discussed intensively in the literature. This dissertation can be understood as a historical survey of the author's work on FOP and AOP, their evaluation, comparison, combination, analysis, and discussion. The structure of the dissertation reflects, beside the chronology of work on this topic, also the evolution of the author's understanding of FOP, AOP, and their relationship.

9.1 Summary of the Dissertation

Chapter 3

We presented in Chapter 3 a classification of crosscutting concerns, which are the main design and implementation problems addressed by FOP and AOP. This classification is crucial to a systematic discussion about separation and modularization of crosscutting concerns. It is a prerequisite for an evaluation and comparison of FOP and AOP.

Chapter 4

The evaluation in Chapter 4 revealed that FOP and AOP are not competing approaches and that their combination can overcome their individual limitations. The strengths and weaknesses of FOP and AOP are expressed in programming guidelines that assist programmers to choose the right implementation technique for the right problem.

Chapter 5 In Chapter 5 we presented an approach for the combination of FOP and AOP. The symbiosis of FOP and AOP incorporates the strengths of FOP and AOP into one uniform approach, which we call *aspectual mixin layers (AFMs)*. AFMs provide a way of designing and implementing programs incrementally; they combine aspect-oriented and feature-oriented programming mechanisms; and they are language-independent. Additionally, we provide tool support for Java/AspectJ and C++/AspectC++. The assessment of AFMs is driven by our evaluation criteria and programming guidelines that demonstrate the successful symbiosis of FOP and AOP, i.e., AFMs profit largely from the strengths of FOP and AOP.

Chapter 6 Given the integration of feature modules and aspects, we addressed in Chapter 6 the issue of whether and how aspect-oriented mechanisms fit the stepwise development style of FOP. We observed that current AOP language mechanisms are not adequate and proposed the integration of aspects and a set of accompanying language mechanisms, which we call *aspect refinement (AR)*. AR unifies classes and aspects with respect to stepwise development. According to this view, aspects are just another software artifact that can be subject of subsequent refinement, which satisfies the principle of uniformity [BSR04].

Chapter 7 In Chapter 7 we presented and discussed the results of applying the notions of AFMs and AR to a non-trivial, medium-sized software project. In this study we implemented 14 of 113 features as AFMs; 8 aspects were refined using AR. This demonstrates the practical applicability of AFMs and AR. An interesting insight gained in this study is that aspect-oriented (advice and inter-type declarations) and feature-oriented (collaborations) mechanisms are not used to the same extent. We found that the dominant role of features is the introduction of new functionality (77% of the code base) and the extension of methods (17% of the code base). Only 6% of the code base represents aspect-oriented mechanisms.

Chapter 8 In Chapter 8 we examined the disproportion of code related to FOP and AOP noted in our case study. We derived from our experience a problem statement: What is the current practice of using AOP and FOP-related mechanisms? The background is that we noticed a confusion about the relationship of crosscutting concerns and collaborations, which was revealed and resolved by this dissertation. Due to the long-standing missing support of collaborations in main stream programming languages, AOP filled a vacuum, i.e., aspects were used for implementing collaborations. But, with the advent of languages, tools, methods, and formalisms that support collaborations, aspects should be avoided in these situations.

The questions that arise are: How many aspects implement collaborations and how many solve problems beyond collaborations, i.e., homogeneous and advanced dynamic crosscuts. To answer these questions, we defined in Chapter 8 a set of code metrics and applied them with tools we provide to 8 AspectJ programs of different size. We found that on average 2% of the code base of the analyzed programs represents ad-

vanced AOP and 98% represents collaborations. We noted that the impact of AOP decreases as the program size increases, i.e., in small-sized and medium-sized programs we found 3% of the code base associated with advanced AOP and in large-sized program only 0.2%. Furthermore, we observed that, despite the marginal use, advanced AOP mechanisms reduced code replication by on average 4%. Also here we found that the benefit of code reduction decreases as the program size increases, i.e., 7% in small-sized and medium-sized programs and 0.3% in large-sized programs. We summarize our suspicions regarding this phenomenon in the following sections.

9.2 Contributions and Perspective

impact of AFMs and AR

The contribution of this dissertation is twofold. First, we evaluated, compared, and combined FOP and AOP to overcome their individual limitations. This resulted in the notions of AFM and AR. However, our work on AFMs and AR contributes not only a design method, language and tool support but also helps in understanding the relationship of aspects and feature modules. FOP and AOP are not competing programming paradigms, but merely decompose software in different ways so that their combination leads to a better program design. Our programming guidelines assist not only programmers but sensitize them to the issues discussed in the dissertation. The tools we provide enable other researchers to make their own investigations in AFMs and AR. Finally, our case study demonstrated the practical applicability of AFMs and AR and it pointed to a further fundamental question: What is the current practice of AOP and how many aspects implement collaborations?

advanced AOP is rarely used

Answering this question is the second contribution of this dissertation. Especially, that advanced AOP is rarely used and that its impact even decreases at larger scales are interesting observations. While we expected the fraction of advanced AOP to be around 5%, we did not expect that the fraction decreases as the program size increases. We have several intuitive explanations that roughly boil down to the sheer complexity that either prevents the programmer to discover aspects or makes implementation problems so complicated that they cannot be modularized well using advanced AOP.

clone detection

An interesting, related branch of research might provide more satisfying answers. Work on clone detection suggests that 5% – 15% of the code base of a program is associated with code clones [Bak95, LPM+97, BYM+98], which are in fact a kind of homogeneous crosscutting concerns. It is known that clones are hard to discover and to avoid and that tool support is necessary. Using clone detection tools we could explore whether there are use cases of aspects additionally to the ones found by hand. So the upper limit for the percentage of code clones could be similar to the upper limit for the percentage of advanced AOP (5% – 15%).

advanced clones

However, the clone detection community considers also parameterized clones and clones that are equal in parts, which we call *advanced clones*. Possibly, advanced clones cannot be modularized using AOP mechanisms, i.e., homogeneous advice and inter-type declarations. This may be because a part or even only a pattern is equal in all clones and AOP does not provide appropriate language mechanisms to express the commonalities and variabilities of the clones. Thus, the 5% – 15% estimation might be too optimistic. Personally, the author estimates that approximately 5% of a program code base is associated with advanced AOP.

9.3 Suggestions for Further Work

According to the two main clusters of contributions of this dissertation, we see two clusters of suggestions for further work: (1) FOP and AOP, (2) aspects vs. collaborations.

Further Work on FOP and AOP

quantification and functional aspects

Regarding the symbiosis of FOP and AOP, we suggest to explore further their relationship at the design and the language level. It is interesting to know how the global quantification of aspects affects or even hinders the incremental development style of FOP. Although touched in this dissertation (cf. Sec. 5.6 and 6.4), we omitted an in-depth investigation. In a ongoing branch of work we address this issue [ALS05, AL06, KAS06].

What is essential?

Another interesting issue is how to strip down the integrated approach of FOP and AOP to provide a minimal set of abstractions and language mechanisms. The question is: What is essential and how can we develop a consistent design method, language, and tool suite? Several researchers made already first steps into this direction [LHBC05, LHBL06, Hut06]

features and genericity

A further interesting line of research arises from the implementation of AFMs with FeatureC++. Similar to C++, FeatureC++ provides a template mechanism for generic programming. This poses the question of when to use generics and when to use feature modules to make a program customizable. The background is that both techniques support the implementation of customizable and reusable code. We observed that the combination of generics and feature modules improves customizability and reusability in SPLs since they act at different scales [AKL06]. While feature modules are the building blocks of an SPL, generics enable feature modules to be adapted to specific needs. We call the combination *generic feature modules* and it is implemented in FeatureC++ [AKL06]. It would be interesting to explore the impact of genericity on non-standard FOP/AOP mechanisms like AFMs and AR.

refactoring

While this dissertation targets the principal differences and commonalities of FOP and AOP in software development, others explored their benefit on refactoring. It will be interesting to revisit work on *aspect-oriented refactoring (AOR)* [HMK05, CB05, MF05, CC04, ZJ04, GJ05, CK03, LST+06, BCP+05] and *feature-oriented refactoring (FOR)* [LHBL06, TBD06, LH06, LHB06, XMEH04] by taking the results and experiences of this dissertation into account.

Further Work on Aspects vs. Collaborations

automatic clone detection

The most remarkable result of this dissertation is probably that, in the analyzed AspectJ programs, only 2% of the code base is associated with advanced AOP and 98% with collaborations. Moreover, the impact of AOP decreases as the program size increases. We suggest that clone detection tools may help to find out whether this proportion should be expected generally or whether either programmers or AOP languages today are simply not capable of exploiting the advantages of AOP. Consequently, it is promising to evaluate several clone detection methods and tools and their use for quantifying the impact of AOP compared to collaborations and OOP. Taking the existence of advanced clones into account, we conjecture that approximately 5% of the code base may be associated with advanced AOP and 95% with collaborations. Due to the multiplicity and diversity of clone detection approaches, this attempt is a non-trivial endeavor and part of further work.

empirical and comparative studies

Finally, it is interesting to compare different collaboration abstraction mechanisms and programming languages and to reimplement aspect-oriented programs by replacing aspects that implement collaborations. The AspectJ programs analyzed here qualify as a starting point. Empirical studies on the aspect-oriented and collaboration-based variants can quantify their performance with respect to understandability, maintainability, reusability, and customizability, etc. A point that is not stressed in this dissertation is the impact of the cognitive distance between programmer and program that depends clearly on the used programming paradigm and its mechanisms. Further empirical studies will have to shed light on this issue.

Bibliography

[AAC+05] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364. ACM Press, 2005.

[AB04] S. Apel and K. Böhm. Towards the Development of Ubiquitous Middleware Product Lines. In *Proceedings of ASE Workshop on Software Engineering and Middleware (SEM)*, volume 3437 of *Lecture Notes in Computer Science*, pages 137–153. Springer, 2004.

[AB05a] S. Apel and K. Böhm. Self-Organization in Overlay Networks. In *Proceedings of CAISE'05 Workshops (Workshop on Adaptive and Self-Managing Enterprise Applications)*, volume 2, pages 139–153. FEUP Edicoes, 2005.

[AB05b] S. Apel and E. Buchmann. Biology-Inspired Optimizations of Peer-to-Peer Overlay Networks. *Practices in Information Processing and Communications (Praxis der Informationsverarbeitung und Kommunikation)*, 28(4):199–205, 2005.

[AB06] S. Apel and D. Batory. When to Use Features and Aspects? A Case Study. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 59–68. ACM Press, 2006.

[ABR06] S. Apel, D. Batory, and M. Rosenmüller. On the Structure of Crosscutting Concerns: Using Aspects or Collaborations? In *GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, 2006. Published at the workshop Web site: http://www.softeng.ox.ac.uk/aople/.

[ACH+05] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc:

An Extensible AspectJ Compiler. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 87–98. ACM Press, 2005.

[AGMO06] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I, Lecture Notes in Computer Science*, 3880:135–173, 2006.

[AKL06] S. Apel, M. Kuhlemann, and T. Leich. Generic Feature Modules: Two-Staged Program Customization. In *Proceedings of the International Conference on Software and Data Technologies (ICSOFT)*, pages 127–132. INSTICC Press, 2006.

[AL06] S. Apel and J. Liu. On the Notion of Functional Aspects in Aspect-Oriented Refactoring. In *Proceedings of the ECOOP Workshop on Aspects, Dependencies, and Interactions (ADI)*, pages 1–9. Computing Department, Lancaster University, 2006.

[Ald05] J. Aldrich. Open Modules: Modular Reasoning about Advice. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer, 2005.

[Ale03] R. Alexander. The Real Costs of Aspect-Oriented Programming. *IEEE Software*, 20(6):92–93, 2003.

[ALRS05] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2005.

[ALS05] S. Apel, T. Leich, and G. Saake. Aspect Refinement and Bounded Quantification in Incremental Designs. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 796–804. IEEE Computer Society, 2005.

[ALS06] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 122–131. ACM Press, 2006.

[ATS04] S. Androutsellis-Theotokis and D. Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys (CSUR)*, 36(4):335–371, 2004.

[Bak95] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 86–95. IEEE Computer Society, 1995.

[BAS05] E. Buchmann, S. Apel, and G. Saake. Piggyback Meta-Data Propagation in Distributed Hash Tables. In *Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST)*, pages 72–79. INSTICC Press, 2005.

[Bat88] D. Batory. Concepts for a Database System Synthesizer. In *Proceedings of the International Symposium on Principles of Database Systems*, pages 184–192. ACM Press, 1988.

[Bat05] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[Bax92] I. D. Baxter. Design Maintenance Systems. *Communications of the ACM (CACM)*, 35(4):73–89, 1992.

[BB06] K. Böhm and E. Buchmann. Free-Riding-Aware Forwarding in Content-Addressable Networks. *VLDB Journal, Online First*, 2006.

[BBG+88] D. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An Extensible Database Management System. *IEEE Transactions on Software Engineering (TSE)*, 14(11):1711–1730, 1988.

[BC90] G. Bracha and W. R. Cook. Mixin-Based Inheritance. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) and the European Conference on Object-Oriented Programming (ECOOP)*, pages 303–311. ACM Press, 1990.

[BCGS95] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer. Creating Reference Architectures: An Example from Avionics. In *Proceedings of the Symposium on Software Reusability (SSR)*, pages 27–37. ACM Press, 1995.

[BCP+05] J. Benn, C. Constantinides, H. K. Padda, K. H. Pedersen, F. Rioux, and X. Ye. Reasoning on Software Quality Improvement with Aspect-Oriented Refactoring: A Case Study. In *Proceedings of the International Conference on Software Engineering and Applications (SEA)*, pages 476–483. International Association of Science and Technology for Development, 2005.

[BDN05] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 177–189. ACM Press, 2005.

[BF88] L. Bouge; and N. Francez. A Compositional Approach to Superimposition. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, pages 240–249. ACM Press, 1988.

[BFS06] E. Bodden, F. Forster, and F. Steimann. Avoiding Infinite Recursion with Stratified Aspects. In *Proceedings of the International Net.ObjectDays Conference*, pages 49–64. Gesellschaft für Informatik, 2006.

[BG97] D. Batory and B. J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering (TSE)*, 23(2):67–82, 1997.

[BHMO04] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 83–92. ACM Press, 2004.

[Big98] T. J. Biggerstaff. A Perspective of Generative Reuse. *Annals of Software Engineering*, 5:169–226, 1998.

[BJMvH02] D. Batory, C. Johnson, B. MacDonald, and D. v. Heeder. Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):191–214, 2002.

[BK04] S. Breu and J. Krinke. Aspect Mining Using Event Traces. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 310–315. IEEE Computer Society, 2004.

[BK06] D. J. Barnes and M. Kölling. *Objects First with Java – A Practical Introduction using BlueJ*. Prentice Hall / Pearson Education, 3rd edition, 2006.

[BLS03] D. Batory, J. Liu, and J. N. Sarvela. Refinements and Multi-Dimensional Separation of Concerns. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 48–57. ACM Press, 2003.

[BO92] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

[Boe06] B. Boehm. A View of 20th and 21st Century Software Engineering. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 12–29. ACM Press, 2006.

[Boo93] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison Wesley Professional, 2nd edition, 1993.

[Bos99] J. Bosch. Super-Imposition: A Component Adaptation Technique. *Information and Software Technology*, 41(5):257–273, 1999.

[Bow96] J. Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. Thomson Publishing, 1996.

[BRJ05] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Professional, 2nd edition, 2005.

[BS01] M. Broy and K. Stoelen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[BSR04] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[BT97] D. Batory and J. Thomas. P2: A Lightweight DBMS Generator. *Journal of Intelligent Information Systems (JIIS)*, 9(2):107–123, 1997.

[BvDvET05] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. On the Use of Clone Detection for Identifying Crosscutting Concern Code. *IEEE Transactions on Software Engineering (TSE)*, 31(10):804–818, 2005.

[BYM+98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 368–377. IEEE Computer Society, 1998.

[BZM01] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing High-Performance Memory Allocators. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 114–124. ACM Press, 2001.

[CB05] L. Cole and P. Borba. Deriving Refactorings for AspectJ. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 123–134. ACM Press, 2005.

[CBML02] R. Cardone, A. Brown, S. McDirmid, and C. Lin. Using Mixins to Build Flexible Widgets. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 76–85. ACM Press, 2002.

[CC04] A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 56–65. ACM Press, 2004.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CHdM06] P. Costanza, R. Hirschfeld, and W. de Meuter. Efficient Layer Activation for Switching Context-Dependent Behavior. In *Proceedings of the Joint Modular Languages Conference (JMLC)*, volume 4228 of *Lecture Notes in Computer Science*, pages 84–103. Springer, 2006.

[CK03] Y. Coady and G. Kiczales. Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 50–59. ACM Press, 2003.

[CL01] R. Cardone and C. Lin. Comparing Frameworks and Layered Refinement. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 285–294. IEEE Computer Society, 2001.

[CM86] M. Chandy and J. Misra. An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):326–343, 1986.

[CN02] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[Con68] L. L. Constantine. Segmentation and Design Strategies for Modular Programming. In *Proceedings of the National Symposium on Modular Programming*. Information and Systems Press, 1968.

[Coo89] W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Department of Computer Science, Brown University, 1989.

[CRB04] A. Colyer, A. Rashid, and G. Blair. On the Separation of Concerns in Program Families. Technical Report COMP-001-2004, Computing Department, Lancaster University, 2004.

[CT04] M. Ceccato and P. Tonella. Measuring the Effects of Software Aspectization. In *WCRE Workshop on Aspect Reverse Engineering (WARE)*, 2004. Published at the workshop Web site: http://homepages.cwi.nl/~tourwe/ware/submissions.html.

[DDH72] O.-J. Dahl, E. W. Dijkstra, and C. Hoare. *Structured Programming*. Academic Press, 1972.

[DFS02] R. Douence, P. Fradet, and M. Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2002.

[DFS04] R. Douence, P. Fradet, and M. Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 141–150. ACM Press, 2004.

[Dij68] E. W. Dijkstra. The Structure of the "THE"-Multiprogramming System. *Communications of the ACM (CACM)*, 11(5):341–346, 1968.

[Dij72] E. W. Dijkstra. The Humble Programmer. *Communications of the ACM (CACM)*, 15(10):859–866, 1972.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[Dij82] E. W. Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982.

[DNS+06] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A Mechanism for Fine-Grained Reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.

[DW06] D. S. Dantas and D. Walker. Harmless Advice. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, pages 383–396. ACM Press, 2006.

[EFB01] T. Elrad, R. E. Filman, and A. Bader. Aspect-Oriented Programming: Introduction. *Communications of the ACM (CACM)*, 44(10):29–32, 2001.

[EOC06] E. Ernst, K. Ostermann, and W. R. Cook. A Virtual Class Calculus. In *Proceedings of the International Symposium on Principles of Programming Languages (POPL)*, pages 270–282. ACM Press, 2006.

[Ern01] E. Ernst. Family Polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, 2001.

[Ern03] E. Ernst. Higher-Order Hierarchies. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes in Computer Science*, pages 303–329. Springer, 2003.

[FGG+06] P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, L. Northrop, D. Schmidt, K. Sullivan, and K. Wallnau. *Ultra-Large-Scale Systems – The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon, 2006.

[FLG06] E. Figueiredo, C. Lucena, and A. Garcia. AJATO: An AspectJ Assessment Tool. In *Demo Session of the European Conference on Object-Oriented Programming (ECOOP)*, 2006.

[FR99] R. Fanta and V. Rajlich. Removing Clones from the Code. *Journal of Software Maintenance*, 11(4):223–243, 1999.

[GB03] K. Gybels and J. Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 60–69. ACM Press, 2003.

[GBB06] J. F. Gelinas, M. Badri, and L. Badri. A Cohesion Measure for Aspects. *Journal of Object Technology (JOT)*, 5(7):75–95, 2006.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GJ05] I. Godil and H.-A. Jacobsen. Horizontal Decomposition of Prevayler. In *Proceedings of the International Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 83–100. IBM Press, 2005.

[Gla05] R. L. Glass. IT Failure Rates – 70 Percent or 10–15 Percent? *IEEE Software*, 22(3):110–111, 2005.

[Gla06] R. L. Glass. The Standish Report: Does it Really Describe a Software Crisis? *Communications of the ACM (CACM)*, 49(8):15–16, 2006.

[Gro95] The Standish Group. Chaos Report. Technical report, Standish Group International, 1995.

[Gro03] The Standish Group. Chaos Report. Technical report, Standish Group International, 2003.

[GS04] J. Greenfield and K. Short. *Software Factories – Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[GSC+03] A. Garcia, C. Sant'Anna, C. Chavez, V. Silva, A. v. Staa, and C. Lucena. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In *Software Engineering for Multi-Agent Systems II, Research Issues and Practical Applications*, volume 2940 of *Lecture Notes in Computer Science*. Springer, 2003.

[GSF+05] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. v. Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 3–14. ACM Press, 2005.

[GSS+06] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60, 2006.

[HC02] 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 (LCTES/SCOPES)*, pages 38–45. ACM Press, 2002.

[Her02] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Proceedings of the International Net.ObjectDays Conference*, volume 2591 of *Lecture Notes in Computer Science*, pages 248–264. Springer, 2002.

[HFC76] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM (CACM)*, 19(5):266–272, 1976.

[HG06] B. Harbulot and J. R. Gurd. A Join Point for Loops in AspectJ. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 63–74. ACM Press, 2006.

[HK02] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173. ACM Press, 2002.

[HMK05] J. Hannemann, G. C. Murphy, and G. Kiczales. Role-Based Refactoring of Crosscutting Concerns. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 135–146. ACM Press, 2005.

[HS03] S. Hanenberg and A. Schmidmeier. Idioms for Building Software Frameworks in AspectJ. In *AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2003. Published at the workshop Web site: http://www.cs.ubc.ca/~ycoady/acp4is03/papers.html.

[HU01] S. Hanenberg and R. Unland. Using and Reusing Aspects in AspectJ. In *OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems (ASoC)*, 2001. Published at the workshop Web site: http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/ASoC.html.

[HU02] S. Hanenberg and R. Unland. Roles and Aspects: Similarities, Differences, and Synergetic Potential. In *Proceedings of International Conference on Object-Oriented Information Systems (OOIS)*, volume 2425 of *Lecture Notes in Computer Science*, pages 507–520. Springer, 2002.

[HU03] S. Hanenberg and R. Unland. Parametric Introductions. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 80–89. ACM Press, 2003.

[Hut06] D. Hutchins. Eliminating Distinctions of Class: Using Prototypes to Model Virtual Classes. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19. ACM Press, 2006.

[Jac02] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[Jac06] M. Jackson. The Structure of Software Development Thought. In *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, pages 228–253. Springer, 2006.

[JSHS96] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL: A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems (TOIS)*, 14(2):175–211, 1996.

[KAS06] C. Kästner, S. Apel, and G. Saake. Implementing Bounded Aspect Quantification in AspectJ. In *Proceedings of the ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, pages 111–122. School of Computer Science, University of Magdeburg, 2006.

[Kat93] S. Katz. A Superimposition Control Construct for Distributed Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):337–356, 1993.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[Ken99] E. A. Kendall. Role Model Designs and Implementations with Aspect-Oriented Programming. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 353–369. ACM Press, 1999.

[Kic06] G. Kiczales. Radical Research In Modularity: Aspect-Oriented Programming and Other Ideas. In *Keynote of the International Software Product Line Conference (SPLC)*. IEEE Computer Society, 2006. http://www.sei.cmu.edu/splc2006/splc_kiczales.pdf.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[KR06] G. Kniesel and T. Rho. A Definition, Overview and Taxonomy of Generic Aspect Languages. *L'Objet*, 11(3):9–39, 2006.

[Kru92] C. W. Krueger. Software Reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[KSG+06] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. v. Staa, and C. Lucena. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 223–233. IEEE Computer Society, 2006.

[Läm99] R. Lämmel. Declarative Aspect-Oriented Programming. In *Proceedings of the International Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 131–146. ACM Press, 1999.

[LAMS05] T. Leich, S. Apel, L. Marnitz, and G. Saake. Tool Support for Feature-Oriented Software Development – FeatureIDE: An Eclipse-Based Approach. In *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange (ETX)*, pages 55–59. ACM Press, 2005.

[LAS05] T. Leich, S. Apel, and G. Saake. Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *Proceedings of the East-European Conference on Advances in Databases and Information Systems (ADBIS)*, volume 3631 of *Lecture Notes in Computer Science*, pages 324–337. Springer, 2005.

[LB99] J. Liebeherr and T. K. Beam. HyperCast: A Protocol for Maintaining Multicast Group Members in a Logical Hypercube Topology. In *Proceedings of the International COST264 Workshop on Networked Group Communication (NGC)*, pages 72–89. Springer, 1999.

[LBL06] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 112–121. ACM Press, 2006.

[LBN05] J. Liu, D. Batory, and S. Nedunuri. Modeling Interactions in Feature-Oriented Designs. In *Proceedings of the International Conference on Feature Interactions (ICFI)*, pages 178–197. IOS Press, 2005.

[LBS04] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic Advice: On the Combination of AOP with Generative Programming in AspectC++. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3286 of *Lecture Notes in Computer Science*, pages 55–74. Springer, 2004.

[LH06] R. Lopez-Herrejon. *Understanding Feature Modularity*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2006.

[LHA07] R. Lopez-Herrejon and S. Apel. Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science. Springer, 2007. to appear.

[LHB06] R. Lopez-Herrejon and D. Batory. From Crosscutting Concerns to Product Lines: A Function Composition Approach. Technical Report TR-06-24, Department of Computer Sciences, The University of Texas at Austin, 2006.

[LHBC05] R. Lopez-Herrejon, D. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer, 2005.

[LHBL06] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proceedings of the International Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 68–77. ACM Press, 2006.

[Lie86] H. Liebermann. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 214–223. ACM Press, 1986.

[Lie04] K. Lieberherr. Controlling the Complexity of Software Designs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 2–11. IEEE Computer Society, 2004.

[Lip05] H. Lipson. Assembly, Integration, & Evolution Overview. In *Build Security In*. Software Engineering Institute and DHS National Cyber Security Division, 2005. https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/assembly/14.html.

[LLM99] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, 1999.

[LLM06] Z. Li, S. Lu, and S. Myagmar. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on Software Engineering (TSE)*, 32(3):176–192, 2006.

[LLO03] K. J. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual Collaborations – Combining Modules and Aspects. *The Computer Journal*, 46(5):542–565, 2003.

[Lod04] K. N. Lodding. The Hitchhiker's Guide to Biomorphic Software. *ACM Queue*, 2(4):66–75, 2004.

[LPM+97] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 314–321. IEEE Computer Society, 1997.

[LR04] N. Loughran and A. Rashid. Framed Aspects: Supporting Variability and Configurability for AOP. In *Proceedings of the International Conference on Software Reuse (ICSR)*, volume 3107 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 2004.

[LST+06] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proceedings of the International EuroSys Conference (EuroSys)*, pages 191–204. ACM Press, 2006.

[MA05] N. McEachen and R. T. Alexander. Distributing Classes with Woven Concerns: An Exploration of Potential Fault Scenarios. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 192–200. ACM Press, 2005.

[MF05] M. P. Monteiro and J. M. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 111–122. ACM Press, 2005.

[MFH01] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–222. ACM Press, 2001.

[MK03a] H. Masuhara and K. Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2003.

[MK03b] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes in Computer Science*, pages 2–28. Springer, 2003.

[ML98] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 97–116. ACM Press, 1998.

[MLWR01] G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard. Separating Features in Source Code: An Exploratory Study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 275–284. IEEE Computer Society, 2001.

[MMP89] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 397–406. ACM Press, 1989.

[MO02] M. Mezini and K. Ostermann. Integrating Independent Components with On-Demand Remodularization. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 52–67. ACM Press, 2002.

[MO03] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–100. ACM Press, 2003.

[MO04] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 127–136. ACM Press, 2004.

[Mog06] J. C. Mogul. Emergent (Mis)behavior vs. Complex Software Systems. In *Proceedings of the International EuroSys Conference (EuroSys)*, pages 293–304. ACM Press, 2006.

[MSL00] M. Mezini, L. Seiter, and K. Lieberherr. *Component Integration with Pluggable Composite Adapters*. Kluwer, 2000.

[MvDM04] M. Marin, A. van Deursen, and L. Moonen. Identifying Aspects Using Fan-In Analysis. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 132–141. IEEE Computer Society, 2004.

[NCM04] N. Nystrom, S. Chong, and A. C. Myers. Scalable Extensibility via Nested Inheritance. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 99–115. ACM Press, 2004.

[Nov00] G. S. Novak. Interactions of Abstractions in Programming. In *Proceedings of the International Symposium on Abstraction, Reformulation, and Approximation (SARA)*, volume 1864 of *Lecture Notes in Computer Science*, pages 185–201. Springer, 2000.

[NQM06] N. Nystrom, X. Qi, and A. C. Myers. J&: Nested Intersection for Scalable Software Composition. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 21–35. ACM Press, 2006.

[NR69] P. Naur and B. Randell, editors. *Software Engineering: Report of the Working Conference on Software Engineering, Garmisch, Germany, October 1968*. NATO Science Committee, 1969.

[OAT+06] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding Open Modules to AspectJ. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 39–50. ACM Press, 2006.

[OH92] H. Ossher and W. Harrison. Combination of Inheritance Hierarchies. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 25–40. ACM Press, 1992.

[OMB05] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer, 2005.

[Ost02] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *Lecture Notes in Computer Science*, pages 89–110. Springer, 2002.

[OT00] H. Ossher and P. Tarr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 734–737. IEEE Computer Society, 2000.

[OZ05] M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 41–57. ACM Press, 2005.

[Par72a] D. L. Parnas. A Technique for Software Module Specification with Examples. *Communications of the ACM (CACM)*, 15(5):330–336, 1972.

[Par72b] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM (CACM)*, 15(12):1053–1058, 1972.

[Par76] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering (TSE)*, SE-2(1):1–9, 1976.

[Par79] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering (TSE)*, SE-5(2):264–277, 1979.

[PGA02] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 141–147. ACM Press, 2002.

[PHOA89] L. Peterson, N. Hutchinson, S. O'Malley, and M. Abbott. RPC in the x-Kernel: Evaluating New Design Techniques. In *Proceedings of the International Symposium on Operating Systems Principles (SOSP)*, pages 91–101. ACM Press, 1989.

[Pre97] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997.

[PSR00] E. Pulvermüller, A. Speck, and A. Rashid. Implementing Collaboration-Based Design Using Aspect-Oriented Programming. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-USA)*, pages 95–104. IEEE Computer Society, 2000.

[RAB+92] T. Reenskaug, E. Andersen, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-Oriented Programming*, 5(6):27–41, 1992.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proceedings of the International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 161–172. ACM Press, 2001.

[RS05] H. Rajan and K. J. Sullivan. Classpects: Unifying Aspect- and Object-Oriented Language Design. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 59–68. ACM Press, 2005.

[SB02] 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 (TOSEM)*, 11(2):215–255, 2002.

[SCT03] Y. Sato, S. Chiba, and M. Tatsubori. A Selective, Just-in-Time Aspect Weaver. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 2830 of *Lecture Notes in Computer Science*, pages 189–208. Springer, 2003.

[SGS+05] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information Hiding Interfaces for Aspect-Oriented Design. In

Proceedings of the International Symposium on Foundations of Software Engineering (FSE), pages 166–175. ACM Press, 2005.

[Sha84] M. Shaw. Abstraction Techniques in Modern Programming Languages. *IEEE Software*, 1(4):10–26, 1984.

[Sin96] V. Singhal. *A Programming Language for Writing Domain-Specific Software System Generators*. PhD thesis, Department of Computer Sciences, University of Texas at Austin, 1996.

[SK03] M. Sihman and S. Katz. Superimpositions and Aspect-Oriented Programming. *The Computer Journal*, 46(5):529–541, 2003.

[SLM98] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, 1998.

[SLU05] O. Spinczyk, D. Lohmann, and M. Urban. AspectC++: An AOP Extension for C++. *Software Developer's Journal*, pages 68–74, 2005.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured Design. *IBM Systems Journal*, 13(2):115–139, 1974.

[Ste00] F. Steimann. On the Representation of Roles in Object-Oriented and Conceptual Modeling. *Data and Knowledge Engineering (DKE)*, 35(1):83–106, 2000.

[Ste05] F. Steimann. Domain Models are Aspect Free. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML)*, volume 3713 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2005.

[Ste06] F. Steimann. The Paradoxical Success of Aspect-Oriented Programming. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 481–497. ACM Press, 2006.

[Tai96] A. Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys (CSUR)*, 28(3):438–479, 1996.

[TBD06] S. Trujillo, D. Batory, and O. Diaz. Feature Refactoring a Multi-Representation Program into a Product Line. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 191–200. ACM Press, 2006.

[TC04] P. Tonella and M. Ceccato. Aspect Mining through the Formal Concept Analysis of Execution Traces. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 112–121. IEEE Computer Society, 2004.

[TK03] D. Tucker and S. Krishnamurthi. Pointcuts and Advice in Higher-Order Languages. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 158–167. ACM Press, 2003.

[TOHSMS99] P. Tarr, H. Ossher, W. Harrison, and Jr. S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 107–119. IEEE Computer Society, 1999.

[TVJ+01] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. Nørregaard Jørgensen. Dynamic and Selective Combination of Extensions in Component-Based Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 233–242. IEEE Computer Society, 2001.

[vGBS01] J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working Conference on Software Architecture (WICSA)*, pages 45–55. IEEE Computer Society, 2001.

[VN96a] M. VanHilst and D. Notkin. Decoupling Change from Design. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 58–69. ACM Press, 1996.

[VN96b] M. VanHilst and D. Notkin. Using C++ Templates to Implement Role-Based Designs. In *JSSST International Symposium on Object Technologies for Advanced Software (ISOTAS)*, volume 1049 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 1996.

[VN96c] M. VanHilst and D. Notkin. Using Role Components in Implement Collaboration-based Designs. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 359–369. ACM Press, 1996.

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM (CACM)*, 14(4):221–227, 1971.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[WKD04] 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 (TOPLAS)*, 26(5):890–910, 2004.

[XMEH04] B. Xin, S. McDirmid, E. Eide, and W. C. Hsieh. A Comparison of Jiazzi and AspectJ for Feature-Wise Decomposition. Technical Report UUCS-04-001, School of Computing, The University of Utah, 2004.

[YC79] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press, 1979. copyright 1979 by Prentice-Hall.

[ZGJ05] C. Zhang, D. Gao, and H.-A. Jacobsen. Towards Just-In-Time Middleware Architectures. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 63–74. ACM Press, 2005.

[ZJ03] C. Zhang and H.-A. Jacobsen. Quantifying Aspects in Middleware Platforms. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 130–139. ACM Press, 2003.

[ZJ04] C. Zhang and H.-A. Jacobsen. Resolving Feature Convolution in Middleware Systems. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 188–205. ACM Press, 2004.

[ZX04] J. Zhao and B. Xu. Measuring Aspect Cohesion. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2004.

Curriculum Vitae

1977, Apr. 21	Born in Osterburg, Germany
1983–1990	Karl-Marx-Oberschule Osterburg, Germany
1990–1995	Gymnasium Osterburg, Germany; graduated with the German 'Abitur'
1995–1996	Civilian service, Hospital "Johanniter Krankenhaus der Altmark", Stendal, Germany
1996–2002	Studies in Computer Science, Otto-von-Guericke-Universität Magdeburg, Germany; graduated with degree 'Diplom-Informatiker' (equivalent to MSc in Computer Science)
1998–2002	Student assistant, Department of Distributed and Operating Systems, Otto-von-Guericke-Universität Magdeburg, Germany
2002–2003	IT-consultant, METOP Private Research Institute, Magdeburg, Germany
2003–2007	Ph.D. student, Department of Technical and Business Information Systems, Otto-von-Guericke-Universität Magdeburg, Germany

Magdeburg, March 22, 2007