Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science

A System of Patterns for the Design of Reusable Aspect Libraries

Maarten BYNENS

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

Oktober 2011

A System of Patterns for the Design of Reusable Aspect Libraries

Maarten BYNENS

Jury:
Prof. dr. ir. A. Haegemans, chair
Prof. dr. ir. W. Joosen, promotor
dr. E. Truyen, co-promotor
Prof. dr. ir. E. Steegmans
Prof. dr. ir. Y. Berbers
Prof. dr. B. Jacobs

Prof. dr. D. Weyns
(Sweden)
Prof. dr. B. Folliot
(France)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering

Oktober 2011

© Katholieke Universiteit Leuven – Faculty of Engineering
Kasteelpark Arenberg 1 bus 2200, B-3001 Heverlee(Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

D/2011/7515/113
ISBN 978-94-6018-412-3

Abstract

Contemporary software systems are complex compositions of diverse kinds of functionality. A key driver in reducing the cost of software development is the availability of reusable libraries. Aspect-oriented software development (AOSD) enables the localization of functionality that traditionally crosscuts the decomposition of the system. However, it remains a challenge to develop reusable aspect libraries. The reason for this is the specific nature of aspect-oriented (AO) composition. It leads to a tight and fragile coupling between aspects and base and makes it hard to manage the implicit interactions that exist between different aspects.

The goal of this dissertation is to incept, describe, classify and integrate idioms and patterns for the architecture, design and implementation of reusable aspect libraries. These libraries should exhibit three reuse qualities: versatility, stability and ease-of-configuration. Versatile aspects are not coupled to any specific application, but are rather applicable to a wide range of applications. Stable aspects are both robust with respect to changes in the base code and extensible to realize new requirements. Easy configuration limits the effort and AO knowledge required to use the library in combination with a specific application.

Initial research on idioms and patterns for AOSD mainly present them as independent solutions to isolated problems. They are fragmentary contributions that are not targeted towards a common goal or problem context. They lack encompassing knowledge that binds and integrates them in a coherent framework. As a result, there is little guidance in selection of the appropriate pattern, especially in the context of reusable aspect libraries.

These insufficiencies form the basis of the main contributions for this dissertation. We present a structured catalog of patterns with a focus on one coherent problem context, namely, how to design reusable aspect libraries that can be easily deployed in various applications, and that are stable

in the presence of evolution. We extend this catalog with a sequence of patterns that guides the development of reusable aspect libraries with easy configuration. The sequence sets out a step-wise approach and presents a first step towards a pattern language. In subsequent steps it results in an extensible and configurable core design with library-controlled mediation of the internal aspect interactions and provides multiple alternative configuration modes. Additionally, we evaluate the benefits of using the patterns and applying the pattern sequence in multiple case studies. The analyzed benefits are versatility, stability and easy configuration.

In summary, the main contribution of our work is an evaluated and structured collection of patterns augmented with a step-wise approach for the architecture, design and implementation of reusable aspect libraries with easy configuration.

Beknopte samenvatting

Hedendaagse softwaresystemen zijn een complexe samenstelling van uiteenlopende soorten van functionaliteit. Een belangrijke factor in het verminderen van de ontwikkelingskost van software is de beschikbaarheid van herbruikbare bibliotheken. Aspectgeoriënteerde software-ontwikkeling (AOSD) stelt ons in staat functionaliteit te lokaliseren die gewoonlijk de opbouw van een systeem overwoekert. Het blijft echter een uitdaging om herbruikbare aspect bibliotheken te ontwikkelen. De reden hiervoor is de specifieke aard van aspectgeoriënteerde (AO) compositietechnieken. Deze leiden tot een strakke en fragiele koppeling tussen de aspecten en het basisprogramma en maakt het moeilijk de impliciete interacties die er bestaan tussen de verschillende aspecten te beheren.

Het doel van dit proefschrift is het formuleren, beschrijven, classificeren en integreren van idiomen en patronen voor de architectuur, ontwerp en implementatie van herbruikbare aspect bibliotheken. Deze bibliotheken zouden drie kwaliteiten met betrekking tot hergebruik moeten vertonen: veelzijdigheid, stabiliteit en eenvoudige configuratie. Veelzijdige aspecten zijn niet gekoppeld aan een specifieke toepassing, maar zijn eerder van toepassing op een brede waaier aan toepassingen. Stabiele aspecten zijn robuust met betrekking tot veranderingen in het basisprogramma en uitbreidbaar om bijkomende vereisten te realiseren. Eenvoudige configuratie beperkt de inspanning en specifieke (AO) kennis die nodig is om de bibliotheek te gebruiken in combinatie met een specifieke toepassing.

Initieel onderzoek naar patronen en idiomen voor AOSD presenteren deze vooral als onafhankelijke oplossingen voor geïsoleerde problemen. Het zijn fragmentarische bijdragen die niet gericht zijn op een gemeenschappelijk doel of probleemcontext. Ze missen overkoepelende kennis die deze oplossingen bindt en in een samenhangend kader integreert. Als gevolg hiervan is er weinig ondersteuning in de selectie van het juiste patroon, met name in het kader van herbruikbare aspect bibliotheken.

Deze tekortkomingen vormen de basis van de belangrijkste bijdragen voor dit proefschrift. We presenteren een gestructureerde catalogus van patronen toegespitst op één samenhangende probleemcontext, namelijk, hoe herbruikbare aspect bibliotheken ontwerpen die gemakkelijk kunnen worden ingezet in diverse toepassingen, en die stabiel zijn ten opzichte van systeemwijzigingen. We breiden deze catalogus verder uit met een sequentie van patronen die de ontwikkeling ondersteunt van herbruikbare aspect bibliotheken met een eenvoudige configuratie. Deze sequentie stelt een stapsgewijs aanpak voor en vormt een eerste stap naar een patroontaal. In opeenvolgende stappen levert het een uitbreidbaar en configureerbaar basisontwerp op met bibliotheek-gecontroleerde bemiddeling van de interne aspect interacties en meerdere alternatieve configuratiemogelijkheden. Daarnaast evalueren we de voordelen van het gebruik van de patronen en de toepassing van de patroonsequentie in meerdere gevalstudies. De geanalyseerde voordelen zijn veelzijdigheid, stabiliteit en eenvoudige configuratie.

Kortom, de belangrijkste bijdrage van ons werk is een geëvalueerde en gestructureerde collectie van patronen aangevuld met een stapsgewijze aanpak voor de architectuur, ontwerp en implementatie van herbruikbare aspect bibliotheken met eenvoudige configuratie.

Acknowledgements

Although my name is on the cover, this thesis really is the result of the efforts of many people. Here I take the opportunity to say a big thank you.

First and foremost I would like to thank Wouter for accepting me as a doctoral student and being a mentor throughout the entire journey. Although difficult to get in touch with, Wouter was always there to support me when I needed it. His insights, honest feedback and motivation have been (and sill are!) inspirational. Wouter taught me many things, large and small, that will remain valuable for many years to come.

Secondly, I would like to thank Eddy for guiding me on a daily basis. His impact cannot be underestimated. Meeting his high expectations made it almost easy to face the reviewers of conferences and journals. Deserving Eddy's infamous *Seal of Approval* has become an implicit motivation for every new endeavor.

Furthermore, I would like to thank Eric and Yolande for being part of my supervisory committee. Additionally, I would like to thank Eric for introducing me to the wonderful world of software development and sparking my interest in the underlying concepts. Eric supervised my master thesis and pointed me to Wouter for the opportunity of doing a Ph.D. I also thank Prof. dr. Bart Jacobs, Prof. dr. Danny Weyns and Prof. dr. Bertil Folliot for accepting the invitation of being part of the jury and their constructive and interesting feedback on my work.

A special thanks goes out to my two still standing office mates Steven and Dimitri for providing a productive, interesting and joyful atmosphere. Dimitri provided me with a real challenge that made me question my vision on ping pong (or was it research?). Steven was always ready for a joke, especially when you least expect it, and made all those trips to alma and *de moete* less about the food, thank you for that! Also thank you to everyone that used to share the office with me: Naeem, Dries, Ruben, Bert, Yves (both of you), Adriaan and Eddy. An additional thanks to Bert for his inspirational work on elementary

pointcuts, which lead to the realization that there was some unexplored terrain on aspects and patterns. Thank you to Wouter De Borger, Bart Van Brabant and Stefan Walraven for their persistence in inviting me to join them for coffee although the answer was typically negative. I would also like to thank all other members of the lantam task force and the DistriNet research group for being an inspiration for all these years.

Finally, I would like to thank my family and friends. Thank you my Telstar team mates for keeping me in shape both on and off the court. Thank you Trui, Tuur, Saar, Benoit, Moe en Va for making me feel at home wherever we meet and all the support for more than 30 years! Also thank you to Patrick, Mia, Ilse and Johan. Thank you all for keeping to challenge me to find better ways to explain what I was doing all these years, and for everything else. My last and most loving thanks goes to Karen and Fons. Thank you for always being there for me and giving me the pleasure of always being there for you too.

Abbreviations

AO Aspect-oriented
AOP Aspect-oriented programming
AOSD Aspect-oriented software development

CBC Coupling between components
CCCS Cyber Conference Chair System
COF Cities of Faith
CR Change request

DIT Depth of inheritance tree
DPar Number of parent declarations
DPrec Number of precedence declarations

EJP Explicit join points

GoF Gang of Four

ITD Inter-type declaration

JPA Join point abstraction
JPM Join point model

LCOO Lack of cohesion over operations
LOC Lines of code

MMS Manage-My-Sales

NOA Number of attributes
NoAdv Number of advices
NoPc Nimber of pointcuts

OOSD Object-oriented software development

PA Pointcut-advice mechanism
PCL Pointcut language

VS Vocabulary size

WOC Weighted operations per component

Contents

Abstract i

Contents ix

List of Figures xiii

List of Tables xv

1 Introduction 1

1.1 Reusability challenges of aspect libraries 1

1.1.1 The nature of aspect-oriented composition 2

1.1.2 The need for reusable aspect libraries 3

1.1.3 The challenges of aspect reuse 4

1.2 Pattern-based approach for reusable aspect libraries 5

1.2.1 Patterns for software quality 6

1.2.2 Patterns for AOSD 7

1.3 Goals and approach 7

1.4 Achievements 8

1.5 Outline 9

2 Aspect-oriented programming: challenges and related work 11

2.1 Introduction to aspect-oriented programming 11
2.1.1 Behavioral crosscutting 11
2.1.2 Structural crosscutting 13
2.1.3 Aspects. 14
2.2 Problems of AOP and related work 14
2.2.1 Quantification challenges 14
2.2.2 Fragile pointcuts 15
2.2.3 Aspect interactions 16
2.2.4 Discussion 17
2.3 Summary 17

3 A catalog of patterns for reusable aspect libraries 19

3.1 Catalog overview 20
3.1.1 Architectural pattern 20
3.1.2 Categories of programming idioms 23
3.1.3 Pattern template & catalog structure 27
3.1.4 Running example 28
3.1.5 Related work 29
3.2 Categories and associated idioms 30
3.2.1 Aspect awareness 30
3.2.2 Join point abstraction 40
3.2.3 Decomposition 52
3.2.4 Mediation 62
3.3 Pattern generality 69
3.4 Example case: a reusable aspect for access control 70
3.4.1 A reusable access control aspect 71
3.4.2 Composition with oblivious base program 74

3.4.3 Composition with aspect-aware base program 75
3.4.4 Revisiting the requirements 76
3.5 Conclusion 76

4 A sequence of patterns for reusable aspect libraries with easy configuration 79

4.1 Pattern sequence overview 80
4.2 Case study pricing library 81
4.2.1 Original design of the pricing library 82
4.2.2 Limitations of the original design 83
4.3 Applying the pattern sequence to build reusable aspect libraries with easy configuration 84
4.3.1 Core 85
4.3.2 Mediation 86
4.3.3 Flexible composition 87
4.3.4 Connecting the aspect library to an application 89
4.4 Related work 89
4.5 Conclusions 90

5 Pattern evaluation: measuring impact on aspect reusability 93

5.1 Evaluation approach and objective 94
5.2 Applying patterns: practical experiences 95
5.2.1 Studied applications 95
5.2.2 Implemented aspects 96
5.3 Predicting reusability 105
5.3.1 Experimental setup 105
5.3.2 Measurement model 105
5.3.3 Predictive measurement 107

5.4 Assessing reusability 109
5.4.1 Experimental setup 110
5.4.2 Measurement model 111
5.4.3 Reuse measurement 112
5.5 Study conclusions 116
5.6 Study limitations 117
5.7 Conclusions 118

6 Conclusion 119
6.1 Revisiting requirements and contributions 119
6.2 Concluding thoughts 122
6.3 Future work 123

A Description of change requests 125
A.1 Change requests MMS 125
A.2 Change requests COF 126

Bibliography 127

List of publications 139

List of Figures

3.1 Architectural pattern for reusable aspect libraries 20
3.2 Architectural pattern for reusable aspect libraries in the context of an oblivious base application 21
3.3 Architectural pattern for reusable aspect libraries in the context of matching interfaces 22
3.4 Architectural pattern for reusable aspect libraries in the context of a standard interface of join point abstractions 23
3.5 High level overview of the design patterns 24
3.6 Structure of *pointcut interface* 32
3.7 Structure of naming convention 34
3.8 Structure of *annotation convention* 35
3.9 Example structure of *structural convention* 37
3.10 Design fragment aspect-awareness 38
3.11 Structure of abstract pointcut 41
3.12 Structure of *marker interface* 43
3.13 Structure of type parameter abstraction and connection 45
3.14 Structure of *annotation introduction* 48
3.15 Design fragment join point abstraction 49
3.16 Structure of *template advice* 53
3.17 Structure of *template pointcut* 56

3.18 Structure of *pointcut method* 57
3.19 Structure of *participant connection* 59
3.20 Design fragment decomposition 61
3.21 Possible structure of *chained advice* 63
3.22 Structure of *dynamic annotation introduction* 66
3.23 Structure of *mediation data introduction* 67
3.24 Design fragment mediation 69

4.1 Design of the pricing library based pattern collection. 82
4.2 Example configuration for the original design of the pricing library. 84
4.3 Core design of the pricing library. 85
4.4 Design of the pricing library with support for mediation. 87
4.5 Design of the pricing library with flexible configuration. 88

5.1 Added and changed pointcuts for each change request in the USell application. 113
5.2 Conciseness of the configuration for each change request in the USell application (in LOC). 115

List of Tables

3.1 Overview of the adapter idioms 50

5.1 Overview of studied applications 96

5.2 Overview of implemented aspects 96

5.3 Variation points of the pricing libraries 108

5.4 Modularity of the pricing libraries (total results and average per module) 108

5.5 Conciseness of the pricing library 108

5.6 Conciseness of the configuration interface 109

5.7 Added and changed pointcuts for each change request in the USell application 112

5.8 Degree of library use for the pricing aspect after 1 change request and after all change requests 113

5.9 Degree of library use for the pricing aspect after 1 change request and after all change requests. 113

5.10 Amount of duplicated code in USell application before and after integration of the pricing aspect 114

5.11 Total LOC for different implementations of the aspects 115

Chapter 1

Introduction

Aspect-oriented software development (AOSD) [34] aims at improving the separation of concerns that are traditionally hard to modularize. Although the modularization of a concern is a necessary condition for reuse, the coupling introduced by aspects challenges the design of reusable aspect libraries. We use the term library to designate a packaged and reusable piece of software without implying any specific methodology to design or use this software. In this chapter we present the reusability challenges for AOSD and give an overview of the pattern-based approach we have taken to deal with these. Subsequently, we list the explicit goals and achievements of our work and end with the outline for the rest of this thesis.

1.1 Reusability challenges of aspect libraries

Krueger defines software reuse as the process of creating software systems from existing software rather than building them from scratch [76]. The main motivations to reuse software are the improved quality of the resulting system and the reduced development cost and effort. Obstacles to software reuse vary in nature from managerial, organizational and economic to more technical.

Reusing software is hard or impossible unless the software has been designed and developed explicitly with this goal in mind. However, designing software for reuse is difficult as well, the challenge being the design of reusable libraries so that they are easy to use or extend (in a controlled way), without losing

generality of the provided functionality. Providing clear interfaces, making explicit the concrete assumptions and offerings the library makes, is essential to achieve reuse. Implicit assumptions will result in reuse failure, because of huge, brittle and difficult to maintain systems [40].

Frakes [36] defines the cost of reuse as follows[1]: $C = (b+E/n)R+(1-R)$ where b is the cost of integrating the reusable library, E is the cost of developing the reusable library, n is the number of times the library is reused and R is the proportion of the system that can be implemented by the library. Additionally, when the system evolves and needs to implement new requirements, the integration of the system with the library needs to be adapted/extended (new value for b). Therefore, to decrease the cost of reuse, the following software qualities are essential: versatility, to increase the potential for reuse (n); easy configuration, to decrease the integration cost (b); and design stability, to decrease adaptation cost for the evolution of the base system (subsequent values of b). Qualities that control the factors R and E are not the focus of this thesis. To increase R, the library should provide a complete set of interesting functionality. This is very domain-specific and cannot be solved using patterns. As we see the use of patterns as the enabler for developing reusable libraries, it is difficult to study the impact on factor E.

1.1.1 The nature of aspect-oriented composition

Object-oriented software development (OOSD) is an established paradigm that has increased the reuse potential of well-designed software. It enables a better separation of concerns, a central principle in the design of software [42], by decomposing software into classes and objects. One of the remaining problems, however, is the scattered implementation of certain system-wide requirements like security and persistence. The presence of such so-called crosscutting concerns is a barrier in complying with that principle. A crosscutting concern is a concern that is conceptually coherent but that in practical implementations is scattered over different software modules and tangled with other concerns within those modules. Studies show that crosscutting concerns increase the number of defects in software [30] and the effort to develop the software [7].

Aspect-oriented software development (AOSD) puts forward the concept of an aspect to modularize crosscutting concerns in order to achieve a better conformity between concerns and modules. Aspects have the capability of intervening in the normal program control flow and structure, e.g. in AspectJ [66, 5] by means of pointcuts, advice and inter-type declarations (ITDs). In general, aspect-oriented composition can be characterized as

[1] The cost of reuse is relative to the cost of all new code (for which $C = 1$)

implicit and quantified composition that basically follows a condition-action pattern of behavior. The condition consists of join points or events in the structure or run-time execution of a program. The join point model defines the available program elements or run-time conditions that can trigger functionality provided by aspects. Join points can be explicit (like broadcasting an event in Ptolemy [101]) or implicitly defined by the underlying aspect system (like in AspectJ). Aspects realize quantified composition [33] by defining pointcuts, expressions that select join points with certain relevant commonalities. As a result aspects can have an effect on multiple elements in the base program, without these elements necessarily knowing exactly which functionality is woven in.

AOSD has become a successful software development paradigm in both research an practice. It has had its own research conference since 2002 [10], but work related to AOSD has been published frequently in other impactful journals and proceedings [67, 118, 35, 43, 119, 68]. In practice, AOSD has not only been adopted in the form of programming languages (e.g. AspectJ has proven to be mature and widely supported), but has also been succesful as part of popular middleware platforms [111, 59].

1.1.2 The need for reusable aspect libraries

Software reuse remains pivotal also with the emergence of AO technology. The availability of reusable aspect libraries is an important factor in the mainstream adoption of AOSD [130]. Similar to reusable components, such aspect libraries can be used as basic building blocks for application development. They provide abstractions relevant for a particular concern, which can be refined by the library user, and encapsulate the complexities, like pointcut fragility and aspect interaction (see Chapter 2). In this sense, they are like frameworks, but we refer to them as aspect libraries to avoid confusion with AO frameworks like JBoss AOP [59] and Spring AOP [111]. Some reusable aspects, targeted at a specific concern, have recently been presented. Examples include transactions [73], concurrency [27] and persistence [103]. Traditionally, non-functional concerns such as the ones mentioned are the main candidates to become reusable aspects, as for instance happened in middleware implementations [59, 111]. However, nothing prevents more functional concerns to become reusable aspects as well. Especially in product-line environments and more mature domains.

1.1.3 The challenges of aspect reuse

Software reuse has always been a challenge. Due to the crosscutting nature of aspect-oriented composition, however, there are some specific challenges for developers of reusable aspect libraries. Aspects are defined in terms of characteristics of the base program. As a consequence, there is a tight coupling between aspects and base. The challenge is thus designing aspects in terms of abstractions, not coupled to any specific application. At the same time, the library should hide the complexity associated with the aspects, in order to ease configuration of the library for the user.

We can link this challenge to the reuse qualities discussed at the beginning of this section (1.1). *Versatility* is about the feasibility of the aspect composition, *Easy configuration* is about the difficulty of making that composition and *Stability* about the consequences of the composition on evolution.

Stability. Stable aspects reduce the effort and risk of extending the system. When the functionality of either the base code or the library is extended or refined, correctness of the configuration is no longer guaranteed [121]. For instance, the fragile pointcut problem is a well-known threat to the stability of AO systems [117, 63].

Loose coupling is important for reusable aspects. Both the aspect and the base program should be able to evolve independently, with a minimal impact on each other. Loose coupling between modules in the context of AOP means more than just not referring to internal specifics of other modules. Because of the quantifying abilities of AOP (one module can have an effect on multiple other modules) there is a need for a stable, but flexible, definition of the relation between aspect and base, in order to make it resistant against modification of a related module. It is the pointcut that embodies this coupling. It's hard to find a good balance between making this coupling either too explicit (like enumerating all method signatures) or too implicit (like using a lot of wildcards).

Versatility. Versatility means that the aspect library can be reused across a range of different applications. To achieve versatility the aspect should make as few assumptions as possible about the structure or behavior of the base code it interacts with. Typically, the difficulty of versatility is that certain design choices, that are fixed by the aspect, conflict with the requirements of the application. As a result, the application developer has to workaround this problem by reimplementing the aspect functionality, leading to duplicated code.

Easy configuration. In the context of this thesis we refer to the process of binding the aspects and filling in the variation points as configuration of the library. When the configuration itself consists of relatively complex AO constructs, using the aspect library is not easy for non-AOP experts and might not be beneficial in the long term. Reusable aspects are meant to be used also by developers without much expertise in AOP. Easing the use of reusable aspects thus requires that composition of aspects can be done based on guidelines and examples of good practice.

Configuration of aspects not only involves identifying the relevant join points. Specifying the crosscutting functionality and fine-tuning how different aspects should work together, is just as important. Libraries will consist of multiple different aspects, each acting independently. As a result it is necessary to specify configuration rules to regulate the interaction between the different aspects (e.g. in AspectJ, precedence declarations can resolve simple interactions [87]).

Summary. We can conclude that it is hard to define aspects so that they are

- expressive: able to describe the intended join points to interact at in a concise way;
- robust: not overly dependent on changes in the base program;
- loosely coupled from implementation details of the base program.

As a result, reusable aspects should abstract from their dependencies and specify both their structure and behavior in terms of these abstractions.

1.2 Pattern-based approach for reusable aspect libraries

Software reuse also includes reusing the knowledge associated with software development like concepts, ideas and best practices. In this thesis we provide reusable solutions to the design problem of reusable aspect libraries in the form of patterns.

1.2.1 Patterns for software quality

A popular form of reusing software development knowledge is that of a pattern. Patterns exist for all activities. In this thesis we will mainly focus on architectural patterns, design patterns and idioms. A pattern can be defined as a three-part rule, which expresses a relation between a certain context, a commonly occurring problem and a reusable solution [3]. An idiom is then a pattern specific to a programming language or programming environment. An idiom describes how to implement particular behavior or structures in code using the features of the given language or environment [11].

When successful, patterns become jargon, helping developers communicate with each other about their designs. The concept of patterns is related to other concepts in software development. For instance, software refactoring often leads to patterns being inserted [64, 37]. Also, there is an interplay between patterns and programming language constructs. Successful patterns inspire new language constructs that in turn lead to new idioms and patterns. Such new language construct can make a pattern obsolete or allows it to be implemented as a reusable component. In that sense, patterns are an alternative to using the newest programming languages with new tools, a new learning curve, etc. This is particularly important for mainstream adoption of new ideas.

For patterns to be really useful, they need to be part of a collection so developers can find them. A collection of patterns can add structure in order to guide the developer in choosing the right pattern. An ad hoc collection of patterns becomes a pattern system when it includes a classification, describes the forces and variation points and reports the relations between the patterns. Forces are the features or characteristics of a situation that, when brought together, find themselves in conflict and create a problem. To consider any solution to the problem effective, the forces must be balanced[11]. A system of patterns gives the developer a complete overview of the available patterns, but lacks a process that guides the subsequent use of patterns and illustrates effective combinations. A pattern language is defined as a network of interrelated patterns that define a process for resolving software development problems systematically [11]. When focussing on a particular problem, a pattern sequence is often more convenient and can serve as an intermediate step towards a pattern language. A pattern sequence represents a process made up of smaller processes —its constituent patterns— arranged in an order that achieves a particular architecture or design in response to a specific situation. From the point of view of a pattern language, a pattern sequence represents a particular path through the language [11].

1.2.2 Patterns for AOSD

As always with the introduction of new programming mechanisms, an important question is how to put these facilities to good use. For aspect-oriented programming (AOP) this is particularly important now that AOP has become more mature and increasingly more developers have started to use this technology [104]. Guidelines and concrete examples help developers to understand the benefits of using the paradigm and its facilities. As design patterns for object-oriented software [37] helped programmers to deal with late binding and encapsulation, we believe there is also a need for design patterns for aspect-oriented software that support programmers with expressive aspect-oriented composition, which has the power to change the behavior of a program drastically. With pointcuts and advice, programmers are often uncertain about which code is going to execute when, similar to the days when late binding was a relatively new concept: programmers were often unsure about which method was going to execute. It is important to note that, without a careful design, using aspects can be problematic and a threat to stable, adaptable designs [115, 26, 77]. As a result, AOP needs its own collection of patterns and principles. In the literature, however, mainly examples of idioms are available [51, 70, 81, 94]. One example idiom is the use of a marker interface in the definition of pointcuts and inter-type declarations (ITD) instead of concrete types from the base application. Additionally, these idioms are mostly presented as relatively independent solutions. However, as experience grows in using these idioms, it is possible to increasingly combine related idioms to form patterns and subsequently, pattern languages.

1.3 Goals and approach

The first goal of this work is to provide the developer with a structured catalog of design patterns and idioms to guide the use of AOP, especially with reusability and stability of aspects in mind. The catalog will describe and classify the patterns and will present the architectural pattern concerning the overall problem of reusable aspect libraries. As elaborated in Sect. 1.1.2, the availability of reusable aspect libraries is the main motivating factor for the mainstream adoption of AOP. For these reasons, we use AspectJ as the representative of a family of AOP languages that have a Java-like language background and support pointcuts, advice and ITD's. Preferably, also type parameters and annotations are available. AspectJ has a workable implementation and is the most mature AO language available. We believe that AspectJ will remain the future mainstream AO language as research on AO

languages will lead to changes to AspectJ instead of new languages replacing AspectJ as mainstream. The process for adding changes into AspectJ is actively taken up by some of the leaders in AOP research[2].

Existing works by various authors have already presented design solutions for AOP [51, 81, 70, 94, 46, 82, 44, 108]. Apart from the inspiration for the patterns themselves, these works also prove the relevance of the need for design patterns for AOP. The design patterns described in existing literature are however fragmentary contributions that are not targeted towards a common goal or problem context. A coherent system that unites and integrates these fragmentary contributions is therefore needed: often interesting links between different design solutions are not discussed. Also, documentation of known uses and guidance in selecting patterns are typically not well addressed in these works.

The second goal is to augment the catalog by providing guidance for pattern selection and presenting a process for the design of reusable aspect libraries by combining the idioms described in the collection. We present this process in the form of a pattern sequence. The libraries that result from using this pattern sequence should be applicable to a wide range of applications without the need for complex configuration. We present the creation of the aspect-oriented architecture and design of an example library in terms of the pattern sequence. Although the patterns have their value for AO composition in general, we focus on AspectJ for the example and its implementation.

The third goal is to evaluate and illustrate the patterns using multiple case studies. We use the patterns to implement some reusable aspects related to a set of recurring crosscutting concerns. Using multiple real-world applications (like a webshop and an online game platform), we experience the impact of the patterns on the reusability of the aspect library implementations. We also show the benefits of using the patterns and the pattern sequence more quantitatively by applying a set of metrics related to versatility, stability and easy configuration.

1.4 Achievements

The main achievement of this dissertation is an evaluated and structured collection of patterns augmented with a step-wise approach for the architecture, design and implementation of reusable aspect libraries with easy configuration. More specifically, our contributions are as follows:

[2] http://www.aosd.net/wiki/index.php?title=Getting_Changes_into_AspectJ

- A structured catalog of patterns with a focus on one coherent problem context, namely, how to design reusable aspect libraries that can be easily deployed in various applications, and that are stable in the presence of evolution [19].
  - The catalog presents a three-layered system: an architectural pattern (top layer) and four categories (middle layer), each focusing on a specific sub-problem, that consist of a set of programming idioms (bottom layer) that can be used to implement the abstract solution provided by the category. The categories are Aspect awareness, Join point abstraction, Decomposition and Mediation.
  - The catalog adds forces and variation points that guide the interested application developer in choosing which solution is best suited for his particular programming context.
  - We identify uses of these patterns in academic and industry contexts.
  - Some patterns now have a more general description than their counterpart in related work. It makes the patterns more generally applicable by elevating the description from a specific problem context.
  - We illustrate how the patterns and idioms can be used together by presenting a case study on access control.
- A sequence of patterns that guides the development of reusable aspect libraries with easy configuration [18]. The sequence presents a step-wise apporach that in subsequent steps results in an extensible and configurable core design with library-controlled mediation of the internal aspect interactions and provides multiple alternative configuration modes.
- A thorough metric-based evaluation of the benefits of using the patterns and applying the pattern sequence in multiple case studies. The analyzed benefits are versatility, stability and ease-of-configuration.
- The wide experience of implementing some reusable aspects and using these in the context of multiple real-world applications has given us extensive knowledge about applying the presented patterns and pattern sequence.

1.5 Outline

In the following chapters we will present in detail the contributions of our work. In chapter 2 we discuss more in-depth the context of aspect-oriented

programming. We first introduce the mechanisms of AOP and AspectJ more specifically. Subsequently, three major difficulties of programming with aspects are discussed and related research efforts that deal with these problems are listed.

In chapter 3 we present a catalog of patterns that helps to deal with the discussed problems, and more generally provides a solution for the design of reusable aspect libraries meeting the requirements versatility, stability and easy configuration. The catalog presents an architectural pattern dealing with the general design problem of reusable aspect libraries and four categories of programming idioms dealing with a specific sub-problem. The catalog consistently describes the categories and the idioms and provides an example case that demonstrates their use.

In chapter 4 we augment the catalog with a step-wise approach that guides the design of reusable aspect libraries with easy configuration. We present a sequence of patterns that each combine a set of idioms to achieve a configurable core design of the aspect library with library-controlled mediation of the internal aspect interactions and flexible configuration by providing multiple alternative modes for binding the aspect library to an application.

In chapter 5 we evaluate the benefits of both the pattern catalog and the pattern sequence. First we present our experiences of using the patterns to implement some reusable aspects for a set of recurring concerns and applying these aspects to multiple real-world applications. Secondly, we conduct a more quantitative study by evaluating the patterns using a set of metrics related to versatility, stability and easy configuration.

And finally, we present a summary of our contributions and future work in chapter 6.

Chapter 2

Aspect-oriented programming: challenges and related work

In this chapter we introduce aspect-oriented programming, illustrate three major difficulties of programming with aspects and give an overview of related research dealing with these difficulties.

2.1 Introduction to aspect-oriented programming

This section gives a short introduction to the principles of AOP. More concretely, we introduce the main constructs in AspectJ, which we use as the language of choice to illustrate the patterns and idioms in this thesis. AOP provides new constructs to dynamically intervene in the control flow of a program and to statically alter the structure of a program. We first address the dynamic facilities of AOP and the static facilities afterwards.

2.1.1 Behavioral crosscutting

An aspect intervenes in the control flow by attaching extra functionality at certain join points. A join point is a well-defined point in the execution of a program. The join point model (JPM) defines exactly which join points are available for a certain programming language. Examples in the JPM of AspectJ

are the call or execution of a method, reading or writing an instance variable, ...

A pointcut is a description of a set of join points that share one or more characteristics using a pointcut language (PCL). For example, the following pointcut captures all executions of a method `foo()` as declared in a type `Sensitive`.

```
pointcut sensitive(): execution(public void Sensitive.foo());
```

AspectJ offers a number of primitive pointcuts (such as `execution` in the example above) that can be used and combined to define more complex pointcuts. Besides the primitives concerning methods and fields, other primitive pointcuts match join points related to object creation, exception handling, advice execution and control flow. Also more dynamic primitives that relate to the actual identity or run-time type of a certain object or parameter are available. Typically, pointcut declarations consist of a combination of these primitive constructs by means of logical operators.

To make the description of pointcuts more expressive, the use of wildcards and type patterns is allowed. For example, the following pointcut matches all calls to getter methods on types `Sensitive` or `Confidential` regardless of access modifier[1], return type or number or types of parameters.

```
pointcut getters():call(* (Sensitive||Confidential).get*(..));
```

AspectJ also supports the use of annotations to describe method signatures (e.g. within a call or execution primitive) and types. E.g. the following pointcut matches all execution join points of methods with a `Sensitive` annotation.

```
pointcut sensitive(): execution(@Sensitive * *(..));
```

Additionally, AspectJ supports a number of annotation-based pointcut primitives which can be used to match based on the presence of an annotation at runtime or to expose the annotation value as context.

The crosscutting functionality that needs to be executed at the join points depicted by a pointcut is defined in a construct called advice. For instance, the following piece of advice specifies that before each join point described by the pointcut `sensitive()`, the method `checkAccess()` will be called.

```
before(): sensitive(){
  checkAccess();
}
```

[1]By default a pointcut matches any access modifier (it is the same as using a wildcard). To capture package-protected methods, the following pattern has to be used: `execution(!private !public !protected * *(..))`

As control flow passes each join point two times (on the way in and on the way out), we need not only to specify at which join points, but also if the advice has to execute before, after or around the join point. Before and after advice are self-explanatory, around advice replaces the original behavior at the join point but has the ability to call that original behavior by using `proceed` - comparable with a `super` call in a method override.

If the advice needs context information from the intercepted join point, there are two main options. Within the advice block the keyword `thisJoinPoint` can be used to access reflective information available at the join point. A more explicit alternative is to expose the needed context information by using pointcut parameters. These parameters behave like output parameters (as e.g. in C#) that get bound in the pointcut, rather than parameterizing the pointcut, and can subsequently be used in the advice. In the following example, the pointcut exposes the sensitive entity as a parameter, which can be used to check access rights in the advice.

```
pointcut sensitive(Sensitive entity):
  execution(public void Sensitive.foo()) && this(entity);

before(Sensitive entity): sensitive(entity){
  checkAccess(entity);
}
```

2.1.2 Structural crosscutting

AOP offers inter-type declarations to alter the static structure of a program, e.g. by adding new fields or methods to a class or extending certain inheritance hierarchies. Member introductions can be defined on classes as well as interfaces. They look similar to normal member definitions, except that, additionally, the type where the member needs to be introduced is specified. For example, the following statement introduces the boolean field `isSensitive` to the `Entity` class.

```
public boolean Entity.isSensitive;
```

Changes to an inheritance hierarchy can be made with `declare` statements. The following statement makes `Calendar` a subclass of `Resource`.

```
declare parents: Calendar extends Resource;
```

The `declare` statement can also be used to define the precedence between aspects in case they run advice at the same join point and to signal warnings or errors based on potentially harmful constructs in the base code.

2.1.3 Aspects.

Aspects are similar to classes. They can contain methods, fields and can be part of an inheritance hierarchy. Additionally, aspects contain pointcut declarations, advice and inter-type declarations (ITD). With these mechanisms aspects can intervene in both the structure and control flow of the base program. An important difference is that aspects are instantiated implicitly (e.g. in AspectJ one singleton aspect will be created but other instantiation schemes are supported).

Aspects may extend classes and implement interfaces. Classes, however, cannot extend aspects. Aspects may extend other aspects on the condition that the super-aspect is abstract. Abstract aspects cannot be instantiated and will thus not be active. They can contain abstract pointcuts, based on which, advice can be defined.

2.2 Problems of AOP and related work

In this section, we discuss three major difficulties of programming with aspects in more detail and give an overview of existing solutions in related work. Briefly put, these difficulties are

- the definition of pointcuts to capture the correct join points;
- the fragility of these pointcut definitions with respect to evolution;
- the interaction of the behavior between different aspects.

2.2.1 Quantification challenges

Aspects define pointcuts to designate when its advice will take effect. These pointcuts are defined using a pointcut language (PCL) by expressing the shared characteristics of the relevant join points. This expression is often quite challenging due to limitations in both the PCL and the join point model (JPM). The JPM defines the set of available join points, but some potentially relevant event are not available as a join point. In AspectJ for instance, loops or parts of a method body are not part of the JPM. This leads to refactoring of the base program (like extracting the target in a separate method) or a workaround in the aspect.

Limitations in the PCL prevent the clear expression of relevant commonalities between join points. This leads to enumeration-style pointcuts that simply list all the join points. Most PCL's are limited to the syntactic structure of the base code (and their run-time counterparts). The experimental language Alpha explores new ways of defining pointcuts towards more semantic reasoning [99].

For instance, in Sect. 2.1, pointcut `sensitive` captured calls to sensitive methods. In case such methods are annotated or are defined in sub-classes of a certain type, there are no difficulties expressing this in the PCL. But what if such a structure is not available; how can the aspect express which methods are sensitive and which not?

Lots of language proposals exist to improve the JPM or PCL of AspectJ. Ptolemy [101] and EJP [56, 57] make join points explicit in the base program. As a result, the JPM becomes more complete and uniform. Languages like Sally and LogicAJ [105] allow aspects to reason about the implementation details of the base program. This enables more fine-grained capturing of join points, but also introduces a more tight coupling. Also more specific extensions to the JPM model exist, e.g. for loops [53] or conditional branches [102].

Tracecuts [127] and tracematches [4] extend the PCL with temporal reasoning of events. This is also possible in the Alpha language, which provides the machinery to define more semantic pointcuts based on a database of events and properties captured at run-time.

2.2.2 Fragile pointcuts

Tourwé et al. discuss the maintenance of software systems using AO technology [121]. Although the better modularization suggests easier maintenance of these systems, the tight coupling introduced by the aspects leads to the AOSD-evolution paradox. The underlying reason for this phenomenon is the fragility of the pointcut definitions. Kellens et al. give the following definition for the fragile pointcut problem [63]:
The fragile pointcut problem occurs in aspect-oriented systems when pointcuts unintentionally capture or miss particular join points as a consequence of their fragility with respect to seemingly safe modifications to the base program.
In a sense, fragile pointcuts are the AO equivalent of the fragile base class program. In OO programming, one cannot ensure the safety of the change to a class without checking all its sub-classes. Similarly, the change to elements in the base code can only be validated by checking all pointcuts and evaluating the effect on that pointcut. The underlying reason for fragile pointcuts is of course that now the elements in the base code crosscut the pointcut definitions

(i.e. a certain join point is scattered over multiple pointcuts and tangled with other join points in those pointcut definitions).

Consider for example that we have a reusable aspect that implements access control functionality. An important relation in this aspect describes the sensitive entities in the application. It is crucial that when the software evolves, the resulting relation (which is modified or extended accordingly), is still the one intended originally.

Lots of different work, targeting the fragile pointcut problem, exists using different software techniques. Ptolemy [101], Open Modules [2], XPI [44] and EJP [57] deal with fragile pointcuts by moving the join point or pointcut declaration to the base code. As a result, these join points or pointcuts can evolve together with the code to which it refers. When these exposed events relate to stable abstractions, fragility of pointcuts based on these events is considerably reduced [125].

Model-based pointcuts [63] reduce fragility by defining them in terms of an abstract model that provides more stable abstractions than the implementation itself. Fragile pointcuts can also be managed with tool support, e.g. pointcut delta analysis [117] and pointcut rejuvenation [65].

2.2.3 Aspect interactions

Because aspects localize functionality that logically belongs together instead of functionality that will run together, the interaction between different aspects is not explicit. AspectJ offers the simple construct of a precedence declaration to designate the order of aspects running at the same join point. However, this will only solve trivial aspect interactions as the order may be more fine-grained or depends on run-time values. A more general solution is to use aspects to manage the interaction of other aspects. This also is not always feasible as the JPM and the PCL are not up to the task.

In our example of access control, suppose each access control rule is implemented by a separate aspect. Each of these aspects will have its own response when a sensitive operation is called. The challenge here is to regulate all these different aspects to result in a single correct response.

Aspect interactions have been studied extensively, but most of this work focuses on detection, rather than resolution [106, 74, 1]. AspectOPTIMA gives an example of how aspects can be used to control the interactions between other aspects [72]. Their approach is as follows: for each conflicting pair of aspects, a new interference aspect takes care of the conflict. Marot et al. extend

the JPM and PCL of AspectJ to improve the ability of regulating aspects with other aspects [87]. Their extension includes named advice methods, advice precedence, pointcut conjunction and disjunction and manual aspect instantiation.

2.2.4 Discussion

These problems have an impact on the development and configuration of reusable aspects. The challenge of quantification makes it hard for the aspect user to configure the aspect to capture the relevant join points. The library developer should foresee this and provide an easy-to-use configuration interface. When such an indirection is provided by the library, the quantification challenges can lead to other problems, e.g. callback mismatch [20]. Callback mismatch refers to the difficulty of binding a type abstraction used by an aspect for both defining join points and issuing callbacks. Similarly, the library developer should minimize the fragility of the resulting pointcuts of configuring the library.

Aspect interactions primary affect the internal design of the library. As realistic libraries contain multiple aspects to implement the provided functionality, regulating the overall behavior of these aspects is of utter importance. Related to all these problems are the assumptions aspects make with respect to the environment in which they will operate. Recent studies show that such assumptions threaten safe AO composition [131].

Partial solutions for dealing with these problems of AOP exist in the form of programming language extensions. Unfortunately, none of these solutions provides an integrated solution for all the problems mentioned above. For the rest of this thesis, we choose to focus on the mainstream platform and programming environment of AspectJ and investigate how we can deal with these problems using patterns and principles.

2.3 Summary

In this chapter we have presented a brief introduction to aspect-oriented programming in general and AspectJ more specifically. Using short code examples we discussed both the structural and behavioral crosscutting abilities of AspectJ.

In the second part of the chapter we discussed three problems related to programming with aspects, linked these problems to the challenges of reuse and

gave an overview of related research efforts that deal with them. The problems are pointcut fragility, the challenge of quantification and aspect interaction. Pointcut fragility refers to the effect of changes in the base code on the join points captured by a pointcut. Pointcut fragility affects the stability of aspect designs. To minimize the impact of pointcut fragility, pointcuts should be defined in terms of abstractions. In the following chapters, we will present patterns and idioms to achieve this.

Aspect interactions can occur in various situations. The problem is that the aspects will behave differently when combined than when run in isolation. Prevention of aspect interactions is difficult to achieve with coding techniques alone. In the following chapters we present patterns and idioms to manage conflicts and other interactions between different aspects.

The challenge of quantification refers to the problem of pointcuts capturing exactly the join points related to a particular concern. In the following chapters we will present patterns and idioms to provide partial definitions of pointcuts in order to minimize the quantification challenge for aspect users. As the partial pointcut definition encapsulates the main AO composition complexity and the aspect user only has to fill in the details, the risk for incorrect quantification is reduced.

Besides patterns and idioms that help dealing with the discussed problems, the following chapters will present our pattern-based approach for the development of reusable aspect libraries, meeting the requirements of versatility, stability and easy configuration.

Chapter 3

A catalog of patterns for reusable aspect libraries

A key direction for achieving mainstream adoption of aspect-oriented (AO) programming is the availability of reusable aspect libraries that can be easily applied across a wide range of applications. This chapter presents a catalog of patterns for AO design that provides solutions for recurring problems in the design of such reusable aspect libraries. We have focused on libraries using AspectJ. We identify an architectural pattern and four categories of programming idioms addressing key design problems: managing aspect-awareness, enabling join point abstraction and adaptation, decomposition and mediation. Each category presents an abstract solution that addresses the related design problem and describes the available idioms that can be used to implement the abstract solution. The catalog aggregates the four sets of idioms that include a specific section to guide selection of a specific idiom. The implementation of an aspect library for access control is discussed to illustrate how the pattern catalog can be used and how the different patterns and idioms can be combined. The format of the catalog is based on pattern writing advice provided by the Hillside group [45]. We have analyzed and integrated related work from the literature on design patterns for AOP and identified known uses in existing systems.

3.1 Catalog overview

This section presents the overall structure of the pattern catalog. The patterns are loosely organized in three layers – architectural pattern, categories and programming idioms – following patterns of software architecture [12]. A single architectural pattern is presented that represents the key architectural design decisions that are applicable to the recurring design problem of building a reusable aspect library. This architectural pattern presents the context for the 4 categories of programming idioms that each deal with a particular sub-problem. Each category presents an abstract design and describes the programming idioms for implementing that design in AspectJ[1].

This section is structured as follows. The architectural pattern is first presented. Thereafter a concise overview of the categories and associated idioms is presented. Subsequently the templates used for describing in more detail the categories and idioms as part of Sect. 3.2 is presented. Finally, we give more detail about the running example on an access control library.

3.1.1 Architectural pattern

Our architectural pattern provides a context for the key design problems related to the development of reusable aspect libraries. The pattern distinguishes between multiple elements in the binding between aspect and base components, namely join point abstractions (provided and required) and join point bindings (connector or adapter). Figure 3.1 illustrates the general case of the architectural pattern.

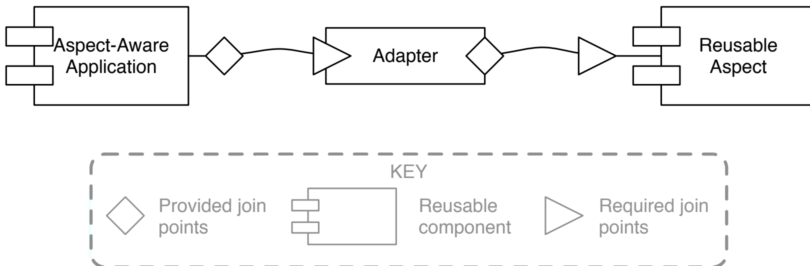

Figure 3.1: Architectural pattern for reusable aspect libraries

[1] We also shortly discuss how to implement similar idioms in CaesarJ, Spring AOP and JBoss AOP.

A first key design decision underlying the architectural pattern is that the aspect-base binding should be defined in terms of join point abstractions. A *join point abstraction* (JPA) is part of an aspectual interface that exposes a set of crosscutting abstract structures or behaviors (represented by ▷). Join point abstractions are defined by reusable aspect components as some kind of expected interface. Optionally, the base application can also expose a set of provided join point abstractions (represented by ◇). In the general case, the join points provided by a base application don't perfectly match the required JPA of the reusable aspect. An adapter is needed to bridge this mismatch. The adapter can be interpreted as a module that exposes both a provided and a required join point interface in such a way that they respectively match the required JPA of the reusable aspect and the provided JPA of the base code (therefore the adapter module in figure 3.1 contains both a triangle and a diamond).

A consequence of exposing join point abstractions, is that they make the base application aspect-aware instead of keeping it oblivious. Making a good trade-off between aspect-awareness and obliviousness is difficult [33, 69, 128] and often depends on the particular software development context in which the aspect library will be used. As a result, the architectural pattern therefore manifests itself in other variants with respect to fully completing the binding of an aspect library to a base application. These variants account for different software development contexts. A first possibility is that the reusable aspect can be bound to an oblivious base application using a *connector*, as depicted in figure 3.2. An oblivious base program is defined as a program that is not

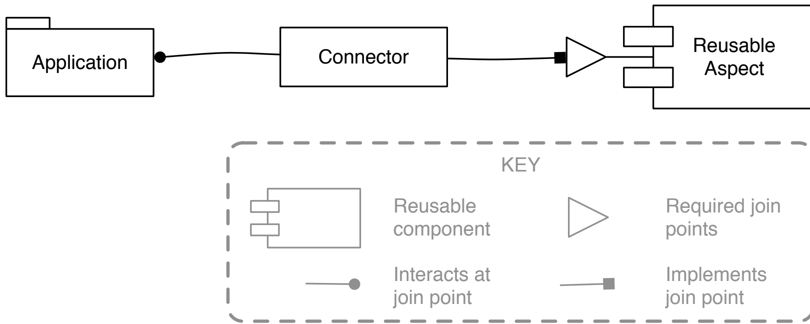

Figure 3.2: Architectural pattern for reusable aspect libraries in the context of an oblivious base application

aware of the fact that any aspects will be applied to it and therefore has not been prepared for providing any explicit join point abstractions to these aspects.

An example scenario for this variant is binding an aspect to legacy code that does not provide explicit join point abstractions. The connector realizes the expected join point abstractions of the aspect library (represented with —■) in terms of the available structure and syntax of base code (represented with —●). The main difference between a connector and an adapter is that a connector implements the abstractions on the required interface of a reusable aspect, while an adapter bridges the abstractions on the required and provided interface of a reusable aspect and aspect-aware application respectively. That is the reason why in the general case (figure 3.1), join point interaction (—●) and implementation (—■) are not explicitly depicted. They are hidden by the interfaces.

A second possibility is that aspect and base expose matching join point interfaces. In such a case, the aspect and the base application can be composed as is (figure 3.3). This is realistic e.g. in the context of a mature domain with

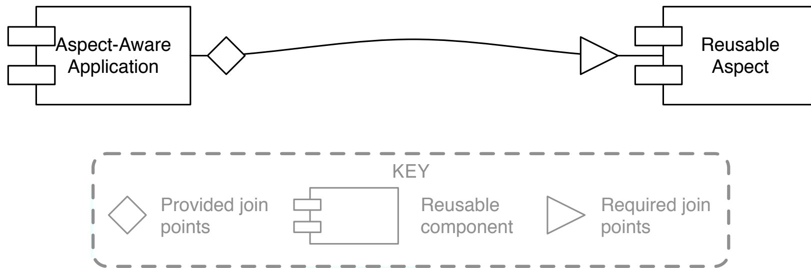

Figure 3.3: Architectural pattern for reusable aspect libraries in the context of matching interfaces

a standard component model like Enterprise Java Beans, where provided join point abstractions are expressed as annotations that provide metadata related to persistence and transactions. As in this case the join point abstractions are agreed upon for a particular domain, figure 3.4 might be a more accurate representation.

Indifferently which variant is selected, to meet the requirements of stability and versatility, it is important that join point abstractions are abstract enough so that they can hide any changes in the implementation of the base application from the aspect components and, conversely, the aspect is applicable to a broad range of applications.

A reusable aspect library not only requires stability and versatility of the binding between aspect and base, but also the stability and versatility of the internal structure of the aspect library. The internal structure of the aspect

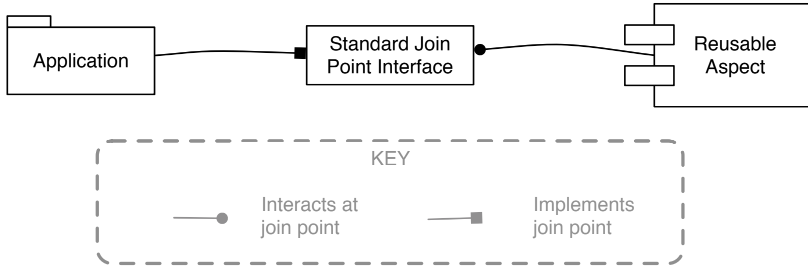

Figure 3.4: Architectural pattern for reusable aspect libraries in the context of a standard interface of join point abstractions

libraries should therefore be decomposed into a set of common and open parts which can be refined into a set of sub-aspects. The aspect library also has to encapsulate support for orchestrating the run-time interactions in the behavior of those sub-aspects.

As a result, the architectural pattern identifies four design problems: two related to the binding between aspect and base (how to abstract from concrete join points in aspect and base code respectively) and two related to the internal structure of the aspect (how to decompose the aspect and how to orchestrate the run-time interaction between the resulting sub-aspects). The next subsection presents the categories and associated programming idioms that deal with these four identified design problems.

3.1.2 Categories of programming idioms

The architectural pattern itself depends upon four categories of programming idioms that each tackle a specific design problem related to the design and integration of reusable aspect libraries. Figure 3.5 gives an overview of the categories by linking them together. In this graph, circles represent categories of related idioms and rectangles represent design problems. Edges connect problems and categories. Initial nodes, indicated by the diamonds, specify the particular development context and the role of the developer involved, i.e., library designer, base developer, and system composer who is responsible for binding the aspect library to the base application. In some development contexts the base developer also plays the role of the system composer. Note that the solution to the design problem of how to connect the aspect library to the target application depends on the solution chosen by the base

developer. This is depicted by the dotted line and corresponds with the different manifestations of the binding in Fig. 3.1.

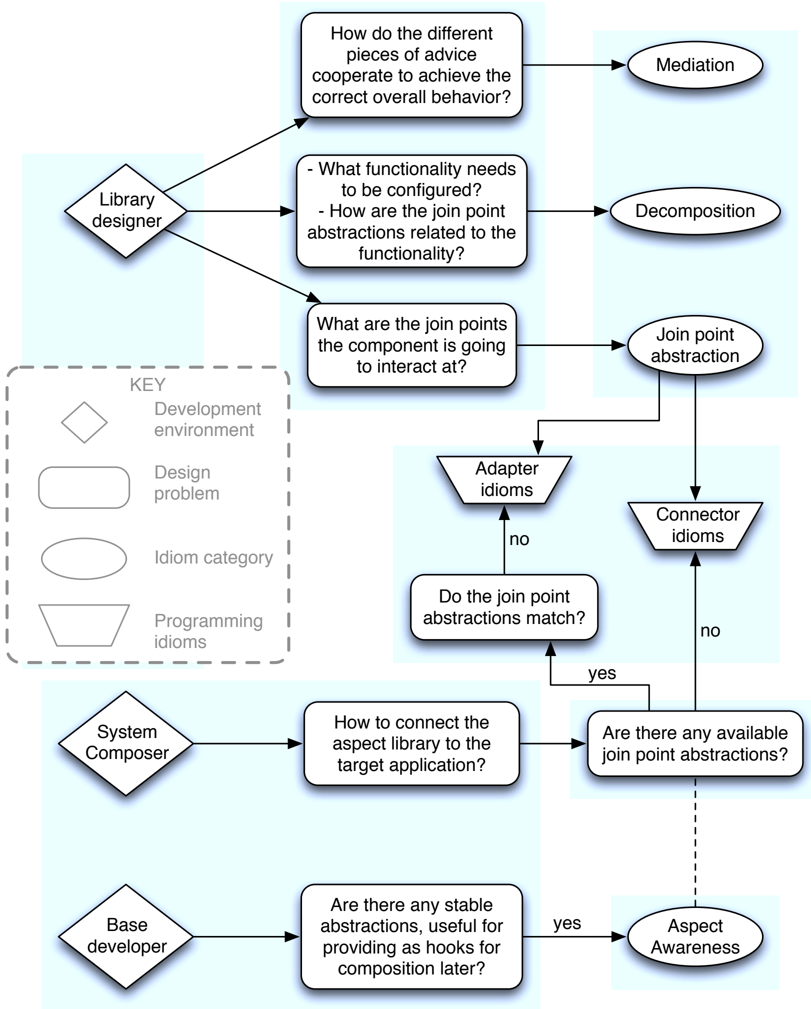

Figure 3.5: High level overview of the design patterns

Each category groups a set of programming idioms. Different forces play a role in the particular software development context corresponding to the category. These forces are contributory factors in the choice for the most suitable idiom or combination of idioms. This means that a force is neither a sufficient or

necessary condition for applying an idiom. Multiple forces can be relevant in a particular problem context. Depending on the relative importance of the these forces, the developer needs to consider whether one force dominates the others and apply the associated idiom or that a combination of idioms is more appropriate. As a result, some idioms are positioned as different variants from which to choose one and other idioms are complementary strategies that each focus at a different dimension of the overall problem of the design pattern. The remainder of this section gives a short overview of the different categories based on Figure 3.5 and the associated idioms.

Join point abstraction idioms are used by the library designer to tackle the first problem at hand, i.e. what is the appropriate set of expected join point abstractions on which the aspect library should depend. The table below lists the different alternative programming idioms for implementing this design pattern in AspectJ. We refer to Section 3.2.2 for a detailed overview of the design problem and the idioms.

Join point abstraction idioms	
Marker interface	A (usually) empty interface that can be used as a type abstraction
Type parameter	Abstract from type information using type parameters in a generic aspect
Abstract pointcut	Use an abstract pointcut to abstract from any crosscutting behavior
Annotation introduction	Use an annotation to abstract from structural join points

Join point abstraction depends further on a set of programming idioms for binding the expected join point abstractions of the aspect with the base application. As already stated above, there are 2 variants of the architectural pattern for which an additional element is needed for completing a full binding between a reusable aspect and a base application. In case a base application does not provide any join point abstraction, the system composer has to use a *Connector* idiom. In case both sides have an explicit interface with join point abstractions, but these join point abstractions do not match or are implemented using incompatible programming idioms, the system composer must use an *Adapter* idiom. For each Join point abstraction idiom there is one compatible Connector idiom and we refer to Section 3.2.2 for a detailed overview. We refer to Table 3.1 in Section 3.2.2 for an overview of the possible Adapter idioms. There exists a separate Adapter idiom for each element in the cartesian product of Join point abstraction and Aspect awareness idioms.

Decomposition idioms target the development of complex aspect components that are too large to be manageable as a single unit. The category proposes

to decompose the aspect component in several smaller (more basic) elements. In the context of AspectJ such elements subject to decomposition can be aspects, but also, at a more fine-grained level, pointcuts, advice and inter-type declarations. A detailed overview of the design problem and related idioms are presented in Section 3.2.3.

Decomposition idioms	
Template pointcut	Decompose a pointcut into hook pointcuts, separating stable and variable parts
Pointcut method	Encapsulate a dynamic condition in a method to be called from advice
Template advice	Decompose advice into hook methods, separating stable and variable parts
Participant connection	Instead of specifying the connection between aspect and base globally in one place, divide the connection in multiple participant connections, each integrated with a particular part of the base application (e.g. to enable simple pointcut definitions)

Mediation targets the behavior of a number of sub-aspects, which are meant to behave as a single unit. This category of idioms is typically applied after decomposition as it brings the decomposed sub-aspects back together to let them cooperate and achieve a common goal. The overview of this category and related idioms are presented in Section 3.2.4.

Mediation idioms	
Chained advice	Let the cooperating aspects interact at the same join point and specify an order between their pieces of advice
Mediation data introduction	Introduce mediation-specific data members into base objects and mediate the cooperating aspects using these data members.
Dynamic annotation introduction	Attach an annotation to a certain control flow by annotating advice and mediate the cooperating aspects using this attached annotation.

The single category of idioms which is relevant for the base developer is *Aspect awareness*. This category targets the development of the base application that exposes a set of provided join point abstractions at which aspects might interact. Aspect awareness does not mean that the base application is specific to some particular aspect, but that it is prepared for composition with aspects. It is aware of the concept of aspectual composition. A detailed overview of this

category and its related idioms is presented in Section 3.2.1.

Aspect-awareness idioms	
Pointcut interface	Expose a set of stable crosscutting behaviors as named pointcut definitions
Naming convention	Consistently use a stable naming scheme in the base code
Annotation convention	Consistently use a stable annotation scheme in the base code
Structural convention	Expose a crosscutting concern by consistently applying the same structure to implement a particular concern

3.1.3 Pattern template & catalog structure

This section presents the structure of the description of the abstract design in the categories and associated programming idioms in Sect. 3.2. We describe each category and its associated idioms in a separate subsection. For reasons of clarity, Connector and Adapter idioms are presented in the same section as the Join point abstraction idioms.

The template we use, is loosely based on the style described by Meszaros and Doblein [88]. The template for describing the abstract design of a category is as follows:

- *Problem.* What problem is addressed by the category?
- *Context.* The circumstances that impose constraints on the solution are called the context and determines the relative importance of its forces.
- *Abstract solution.* How is the problem solved by using this category? The category presents an abstract solution which is refined or extended by the idioms.
- *Abstract forces.* Considerations that must be taken into account when choosing the solution are called forces. The description presents the abstract forces which describe in general when this category is applicable. These considerations should not be interpreted as a set of necessary conditions for applying the category.
- *Rationale.* Discusses why the solution is appropriate. The rationale refines the problem, motivates the solution and introduces an example.
- *Related work and known uses.* This paragraph enumerates known uses and related descriptions of this category.

- Next, each idiom is described by refining and/or extending the general description of the category. It consists of the following elements:
  - *Name.* Every idiom has a name by which it can be referenced.
  - *Problem.* Only present if the idiom solves a more specific problem than its design pattern.
  - *Solution.* How is the problem solved by using this idiom?
  - *Forces.* The forces of an idiom discuss when the idiom is applicable in comparison to the other idioms for the same pattern.
  - *Rationale.* Why is this idiom an appropriate solution? It completes the example and discusses related issues.
  - *Implementation in other AOP technologies.* Discusses how this idiom could be implemented in other AOP languages and frameworks besides AspectJ. CaesarJ, JBoss AOP and Spring AOP are included.
  - *Related work and known uses.* The last paragraph enumerates known uses and descriptions of this idiom.
- *Usage of associated programming idioms.* Clarifies how the related idioms are typically applied to solve the problem. Further guidance is given on using the appropriate idioms for implementing the abstract design, depending on the specific development context and forces that need to be resolved.
- *Ease-of-use.* Discusses the impact of each idiom on the ease-of-use requirement. Some idioms result into more easy-to-use aspect libraries then other idioms.

3.1.4 Running example

In the rationale section of each pattern description we continue with the access control aspect and use the calendar system from Verhanneman et al. [126] as a running example. This calendar system allows users to book appointments and resources (like laptops, projectors, . . .). Three main types of users can be distinguished: calendar owners, secretaries and employees. Their access rights can be determined as follows:

- the owner of a calendar can book, edit and delete entries, but cannot book a resource;
- a secretary can edit entries and book resources;

- all employees can view the entries in a calendar but cannot change them.

For instance, if we look at the Calendar interface, methods `showEntries`, `newEntry` and `deleteEntry` require access control, while `getOwner` doesn't. `showEntries` is accessible to all `Employee` objects, but `newEntry` and `deleteEntry` are only accessible to the registered owner of the `Calendar` object.

```
public interface Calendar{
 Entry[] showEntries();
 void newEntry(TimeInterval t, Resource[] r);
 void deleteEntry(Entry e);
 Employee getOwner();
}
```

Listing 3.1: Calendar interface from the access control example

Figure 3.2 presents a naive AO implementation for access control.

```
public aspect AccessControl {
  pointcut calendarAccessDenied(Object e, Calendar c):
    target(c) && this(e) &&
    !(this(Employee) && call(* Calendar.showEntries())) &&
    !(this(Secretary) && call(* Calendar.editEntry())) &&
    !(this(Employee) && call(* Calendar.*Entry(..))
      && if(c.owner==e));

  pointcut resourceAccessDenied():
    target(Resource) &&
    !(this(Secretary) && call(* Resource.book()));

  Object around(): calendarAccessDenied(*,*) ||
    resourceAccessDenied() {
      throw new SecurityException();
  }
}
```

Listing 3.2: Naive implementation of access control

This aspect is very specific to the application described above. In the following section we present the categories and their associated idioms to improve the implementation of access control piece by piece and make it more reusable.

3.1.5 Related work

Here we give a brief overview of related work on design patterns for AOP. More details about related work is part of each pattern description. To the best of our knowledge, this is the first comprehensive system of AOP patterns that supports the construction of libraries. We obviously build upon earlier publications that contain initial description of design patterns, and possible small collections of these [51, 70, 81, 44, 46, 94, 109, 108], as well as lessons

learned from developing reusable aspects [72, 27, 129, 103], some empirical studies [62, 93] and proposed language extensions for AOP [2, 56, 101]. Clearly this work is not about supporting classical OO design patterns by using AOP [52, 39].

Patterns for AOP have also been presented in the context of adoption of AOP in commercial projects [110]. It presents best practices with respect to the organizational problems on how to integrate AOP in a software project. Hanenberg and Costanza discuss the merits of their work in the context of the discussion of patterns versus strategies [47]. They conclude that their presented strategies are not patterns, but can be seen as important first steps towards patterns for AOP.

3.2 Categories and associated idioms

This section gives a template-based description of each category and related programming idioms in a separate subsection.

3.2.1 Aspect awareness

Problem. How to design base code that is not tightly coupled to aspects while still facilitating the expressive power of AOP? How to get a grip on fragile pointcuts from the base code perspective?

Context. Development of base code that can be easily composed with multiple aspects at once and reused in combination with different aspects over time.

Abstract solution. Explicitly expose relevant join points in the base code, which aspects can use to interact with. These exposed join points include context information and encapsulate the positions where this information is relevant.

Abstract forces. Use an aspect awareness idiom when

- the application depends on aspects to implement certain crosscutting concerns not handled by the base code;
- the base program should not be specific to one particular aspect;

- the program abstractions currently representing the crosscutting concern are not stable. Using them could result in fragile pointcuts;
- it doesn't take much effort to explicitly expose relevant join points.

Rationale. A good place to start looking for join points to expose are intrinsic properties of the program. In our access control example the notion of sensitive operations is an important intrinsic property. An access control component needs to know which objects or operations are sensitive and thus must check access to. Whether it is an aspect or any other technology that implements the authorization behavior, it would benefit significantly if such a notion is exposed.

The fragile pointcut problem will not be solved by using abstract awareness patterns, but it will become more controllable. As the join point abstraction (whether it is a pointcut or something else) is closer to the join points themselves, the impact of a discrepancy between them will be localized. This enables the join point abstraction to evolve together with the join points it refers to. Additionally, aspect awareness allows modular reasoning about the behavior of the application in the presence of aspects.

Related work and known uses. Opening up base code is not new. For instance, C# provides event handling with delegates to enable future extensions. Ptolemy [101] is a language with quantified, typed events that aims to combine the advantages of event-based programming and AOP. Also related are *explicit join points* [56, 57] that make base code aspect-aware and allows explicit interaction in order to increase aspect modularity.

Noble et al. describe patterns for AOSD on an architectural level [94]. Their *extension* pattern corresponds to our aspect-awareness design pattern. In requirements engineering, the principle of preparing the base requirements is called *scaffolding* [23].

In practice, the principle of opening up some modules for future extension is well known, e.g. in white-box frameworks, Eclipse extension points, etc. Spring Insight [113] supports visibility into web application performance through instrumenting Spring and demonstrates the value of having code that is consistently structured by frameworks. It basically is an aspect library with a user interface on top that leverages the consistent structure of Spring applications.

Pointcut interface

Solution. A *pointcut interface* exposes a set of stable crosscutting behaviors as named pointcut signatures.

Forces. Use a *pointcut interface* to expose join points when

- the abstraction to be exposed captures certain points in the control flow of the program, as opposed to certain points in the static structure of the program. We call them behavioral join points;
- the join points to be exposed can be adequately expressed in a PCL;
- the abstractions are exposed with aspect-based composition explicitly in mind.

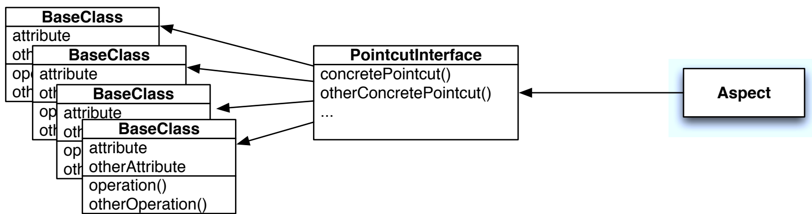

Figure 3.6: Structure of *pointcut interface*

Rationale. Using a *pointcut interface*, we could expose the notion of sensitive operations as follows:

```
public interface SensitiveOperations {
  pointcut sensitive(Object caller): call(* Calendar.*(..)) &&
   this(caller) && (call(* showEntries()) ||
                    call(* newEntry(..))  ||
                    call(* deleteEntry(..)));
}
```

Listing 3.3: *Pointcut interface* that captures sensitive operations

Since a pointcut interface exposes certain behaviors instead of the operations themselves, we must make a choice whether to capture the execution of the operation or the calls to it. Because in the context of access control caller info might be important, we choose to expose the calls to sensitive operations as a crosscutting behavior.

In other contexts this might result in *pointcut interface* not being useful. E.g. if we expect that an aspect will need to add members using an ITD, a pointcut interface is of no use.

A benefit of *pointcut interface* is that it can be used hierarchically. E.g. when a program consists of a number of subsystems (packages, classes), each exposing its own pointcut interface, an umbrella interface can combine the exposed behaviors into a single pointcut interface.

Implementation in other AOP technologies. Classes in CaesarJ can contain pointcuts. If the pointcut is public it can be reused just as in AspectJ, otherwise mixin composition will do the trick. JBoss AOP and Spring can provide a pointcut interface either in an XML file or as annotations in an aspect class.

Related work and known uses. The notion of a pointcut interface was first presented by Gudmundson and Kiczales [46]. Grisworld et al. [44] refine this notion as crosscutting interfaces (or XPI). A crosscutting interface decouples aspect code from the unstable details of advised code in a similar way as a pointcut interface. Furthermore, an XPI imposes a contract and design rules.

The notion of providing pointcuts in the base code is also presented as a programming language construct. The underlying theory is called *Open Modules* [2].

Example usage of *pointcut interface* can be found in the DigiNews application [122, 96]. A domain-driven methodology was used to find stable abstractions for the design with pointcut interfaces [123]. The same methodology was applied to a common case study for AO modeling [124]. In the context of extending OO frameworks, extension join points [78] are used to denote the hot-spots of the framework. The reference documentation of the Spring application framework [112] shows how to share common pointcut definitions. They recommend defining a `SystemArchitecture` aspect that captures common pointcut expressions, which is similar to *pointcut interface*.

Naming convention

Solution. Consistently use a stable naming scheme for types and methods in the base program.

Forces. Use *naming convention* to expose join points when

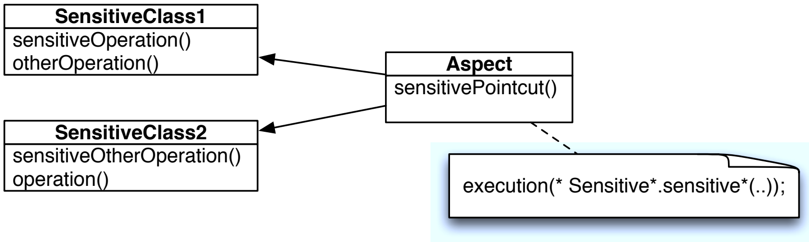

Figure 3.7: Structure of naming convention

- the program structure aligns well with the join points to be exposed. The join points map easily to certain methods, types, . . . ;
- it is the dominant join point abstraction for the program elements (it is cumbersome to expose multiple join point abstractions for the same program element using a naming convention);
- other tools or technology can benefit from the exposed abstraction.

Rationale. If there are no other concerns that interfere, we can expose all sensitive operations by starting their name with `sensitive` and making sure that other methods do not. As a result a PCL can easily express the notion of sensitive operations using a syntax pattern. The `Calendar` interface could then be specified as follows:

```
public interface SensitiveCalendar{
  Entry[] sensitiveShowEntries();
  void sensitiveNewEntry(TimeInterval t, Resource[] r);
  void sensitiveDeleteEntry(Entry e);
  Employee getOwner();
}
```

Listing 3.4: *Naming convention* that captures sensitive operations

These naming conventions can then easily be exploited in an aspect as follows:

```
pointcut sensitiveCalls: call(* sensitive*(..));
```

Implementation in other AOP technologies. With few exceptions CaesarJ uses the same pointcut language. Naming conventions can be used without any difference. Also in JBoss AOP and Spring naming conventions can be taken advantage of. They provide wildcards and JBoss AOP even provides some powerful patterns to describe types.

Related work and known uses. Kiczales and Mezini briefly discuss naming conventions [70] and compare them with a number of other implementation strategies for achieving separation of concerns. Laddad [81] mentions the benefits of using naming conventions to simplify pointcut expressions. Subject-oriented programming techniques [54, 119, 6] uses names as the basis to compose programming elements from the different subjects.

Naming conventions were one of the implementation strategies that Wampler used when developing Contract4J [129].

Annotation convention

Solution. Consistently use a stable annotation scheme in the base code.

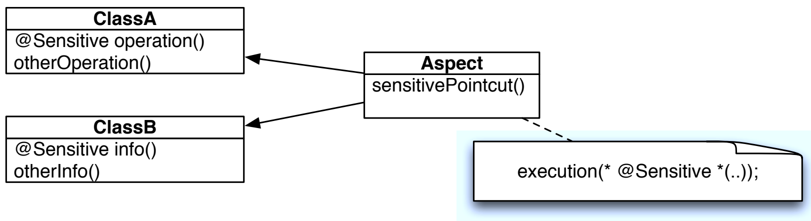

Figure 3.8: Structure of *annotation convention*

Forces. Use *annotation convention* to expose join points when

- the program structure aligns well with the join points to be exposed. The join points map easily to certain methods, types, . . . ;
- some program elements need to expose multiple join point abstractions;
- there are other uses for the exposed join points besides aspect-based composition.

Rationale. No matter which other concerns interfere, we can annotate sensitive operations with a designated annotation (e.g. @*Sensitive*()). The Calendar interface could then be specified as follows:

```
@Sensitive public interface Calendar{
  @Sensitive Entry[] showEntries();
  @Sensitive void newEntry(TimeInterval t, Resource[] r);
```

```
  @Sensitive void deleteEntry(Entry e);
  Employee getOwner();
}
```

Listing 3.5: *Annotation convention* that captures sensitive operations

Additionally, annotation parameters can be used to specify sensitivity levels, accepted roles, ...

These annotations can then easily be exploited in an aspect as follows:

```
pointcut sensitiveCalls: call(@Sensitive * *(..));
```

Implementation in other AOP technologies. As CaesarJ is based on Java2, there is no support for annotations (yet). Annotations are available in JBoss AOP and Spring and can be captured in pointcuts.

Related work and known uses. Kiczales and Mezini briefly discuss annotation conventions [70] and compare them with a number of other implementation strategies for achieving separation of concerns. Laddad [79, 80] describes the use of annotations in combination with aspects in more detail.

In practice, annotations are commonly used, e.g. in Junit [61] or in EJB [89] to configure container services like persistence and authentication. Also Spring Roo makes extensive use of annotations to add extra behavior to Java applications [114] (by generating aspects in the background).

Structural convention

Solution. Expose a crosscutting concern by consistently applying the same structure to implement a particular concern.

Forces. Use *structural convention* to expose join points when

- the join points to be exposed are difficult to express using a PCL;
- the join points to be exposed do not correspond well with available method and type names and annotations;
- there are other uses for the exposed join points besides aspect-based composition.

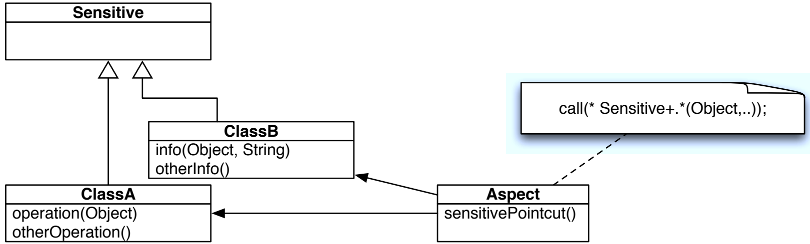

Figure 3.9: Example structure of *structural convention*

Rationale. To expose sensitive operations by means of a structural convention, we need some pattern of behavior that can be recognized and apply it to every sensitive operation.

```
public interface Calendar implements Sensitive{
  Entry[] showEntries(Object caller);
  void newEntry(Object caller, TimeInterval t, Resource[] r);
  void deleteEntry(Object caller, Entry e);
  Employee getOwner();
}
```

Listing 3.6: *Structural convention* that captures sensitive operations

The pattern we use here, is that every method that is called on a subtype of `Sensitive` with the caller object as first parameter, is a sensitive operation. This pattern of behavior can be captured by an aspect and the caller object can be used to determine access rights. The aspect can also be used to check that the first parameter is in fact the caller object.

```
pointcut sensitiveCalls(Object caller):
  call(* Sensitive+(Object,..) && callerArgument(caller,*);
pointcut callerArgument(Object caller,Object arg):
  this(caller) && args(arg,..) && if(caller==arg);
```

Implementation in other AOP technologies. Any structural convention that is limited to constructions from Java2 can be used in CaesarJ (e.g. no annotations or type parameters). Structural conventions are very suitable in JBoss AOP and Spring, also because JBoss AOP provides powerful constructs to define types.

Related work and known uses. The most common structural convention is providing method hooks for interesting join points [62, 93]. This can even lead to the extraction of an empty method, just for the sake of providing a

join point. In requirements engineering, this is called scaffolding [23]. Another example of a structural convention is the use of OO design patterns. They have known names and structures and are thus ideal candidates, as e.g. shown by Chakravarty, Regehr and Eide [22].

The standard practice in Java of implementing certain interfaces (e.g. Remote, Serializable) can be considered a structural convention.

Summary Aspect awareness

To finish the description of the Aspect awareness category, we give an overview of the main forces that drive the selection for the most appropriate idiom or combination thereof and briefly discuss their impact on ease-of-use.

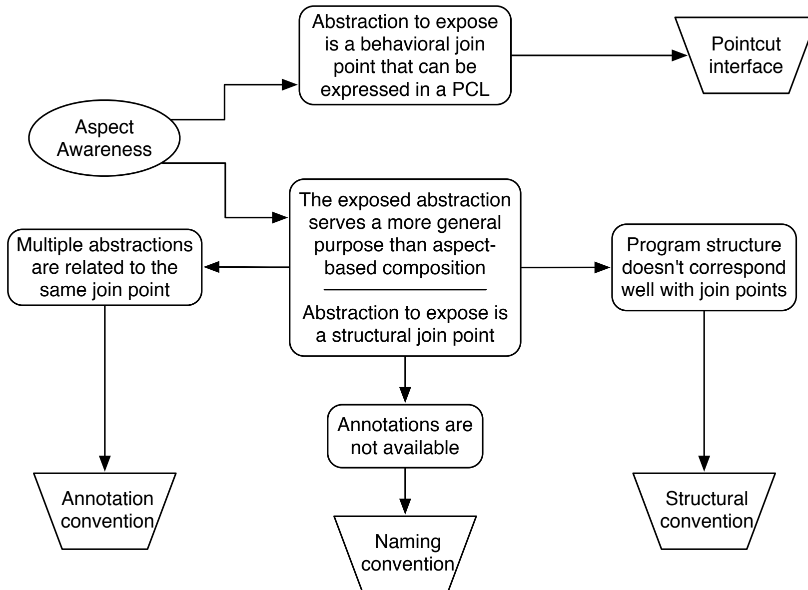

Figure 3.10: Design fragment aspect-awareness

Usage of associated programming idioms. Figure 3.10 presents the design fragment for the Aspect awareness idioms. The figure links the different programming idioms together with the forces related to aspect awareness in a graph. Starting node of the graph is the Aspect awareness design problem. At each node, arrows lead to different forces that may be relevant. These forces

lead to the appropriate idiom or to other forces. It is perfectly possible that at a certain node, multiple forces are relevant[2]. It is then up to the application developer to decide which force is dominant or to apply a combination of idioms, e.g. a pointcut interface can have pointcuts based on an annotation convention. Navigation through the graph is thus basically a process of elimination. For each choice of one or more forces, some idioms will not be reachable anymore and are thus not good solutions for that particular problem[3].

The main variation point is the use of explicit base-level pointcuts (written using a PCL) versus the consistent use of a convention. Important factors regarding this decision are the ability of the PCL to capture the needed crosscutting behavior and whether the exposed abstractions should have other purposes besides aspect-based composition. For example, other program transformation tools or middleware already rely on certain annotations in the base program. Annotations could also be used for creating domain-specific languages or for verification or analysis purposes. The use of a convention can be based on:

- providing annotations;
- naming of program elements (classes, methods, ...);
- program structure (calling certain methods, implementing an interface, ...).

The use of *naming conventions* is discouraged especially if the AOP language supports annotations. Both techniques attach additional semantics to a method or type, but annotations are less error-prone.

Ease-of-use. *Aspect awareness* has a positive impact on the ease of integrating an aspect with the base code, because the exposed abstractions can be leveraged when defining the binding of the aspect. With respect to definition of the exposed abstractions *pointcut interface* requires some knowledge and experience of the PCL. Otherwise, structural conventions potentially require synthetic code structures. The use of annotations offers a very flexible and lightweight composition technique.

[2] If none of the forces at a certain node seem relevant proceed as if all forces are equally important, i.e. the whole sub-graph from that node needs to be taken into account.

[3] To keep the figure clean, some nodes may contain multiple forces. If at least one of the forces is relevant, the node can be taken into account.

3.2.2 Join point abstraction

Problem. How to design an aspect that is not tightly coupled to the join points at which it interacts.

Context. Development of an aspect that can be easily composed with multiple base components at once and reused in combination with different base applications over time.

Abstract solution. Define pointcuts in terms of join point abstractions and expose these abstractions to form some sort of expected interface for the composition of this aspect.

Abstract forces. Use join point abstraction when

- a versatile aspect is needed, which should be applicable to a wide range of applications. As a result we cannot depend on knowledge about the available join points;
- the stability of the available join points is not guaranteed;
- the available join points are known, but do not align well with the program structure;
- base code might already have exposed the necessary abstractions (e.g. EJB annotations);

Rationale. A join point abstraction can come in many shapes and forms as both behaviors and structures can be abstracted. The join point does not have to be entirely abstract. By giving a partial definition of a join point, we can put more constraints on the structure of the binding between aspects and base.

A reusable aspect for access control will interact at join points related to sensitive objects, objects in the application that need protection against unauthorized access. Instead of making assumptions about the name, structure and other characteristics of these join points, we can make abstraction of some or all of these characteristics.

Related work and known uses. Making abstraction of the join points at which an aspect interacts is not new. E.g. model-based pointcuts [63] have been proposed as an approach to manage the fragile pointcut problem. Also aspect integration contracts [83], an interface-based approach towards more robust AO composition, include base code properties.

Abstract pointcut

Solution. Use an abstract pointcut to abstract from any crosscutting behavior.

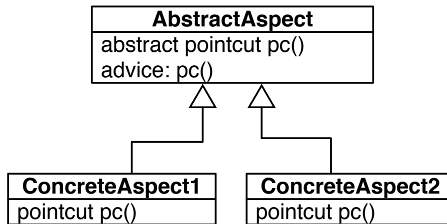

Figure 3.11: Structure of abstract pointcut

Forces. Use an abstract pointcut to abstract from join points when

- there is no knowledge on the kind of join point the aspect is going to interact at;
- different kinds of behavioral join points are abstracted (method join points, join points on certain objects, control flows, . . .);
- the abstraction is only used to bind advice (not for ITD's);
- the join point abstraction is shared between related aspects (they could share a super-aspect);
- the join point abstraction describes a heterogeneous concern (i.e. the advice may vary for different sub-aspects).

Rationale. An abstract pointcut is the most straightforward mechanism to abstract from join points. This pattern leaves it up to the base developer or aspect composer to specify the correct pointcut declaration. As a result, an

abstract pointcut is a very flexible technique, but offers no means to restrict the range of join points. In such a case, documentation is vital.

```
public abstract aspect AccessControl {
  //...
  abstract pointcut checkAccess(AccessSubj s, AccessObj o);
}
```

Listing 3.7: Abstract pointcut example

Connect the abstract pointcut by giving it a concrete definition in a sub-aspect. E.g.,

```
public aspect MyAccessControl extends AccessControl {
  pointcut checkAccess(AccessSubj s, AccessObj o):
    call(* AccessObj+.*(..)) && this(s) && target(o);
}
```

Implementation in other AOP technologies. Abstract pointcuts are available in CaesarJ. They are even more flexible as CaesarJ supports mixin composition and not only abstract aspects can be refined further. In JBoss AOP it is not possible to define abstract pointcuts. However, because binding of advice to pointcut is specified separately, this is not always a limitation. Spring supports abstract aspects using AspectJ or the @AspectJ development style.

Related work and known uses. The concept of an abstract pointcut stems directly from the language [120] and is therefore a straightforward idea. It is described in more detail by Hanenberg [51], who also elaborates on what its role could be in a bigger picture [49].

Example uses of an abstract pointcut can be found a.o. in Cunha's framework for concurrency [27], Hannemans implementation of the GoF design patterns [52] and the ajlib incubator project [9], which implements some AspectJ aspects which are meant to be reusable.

Marker interface

Solution. Separate type information from a pointcut or ITD and refer to a marker interface instead. The marker interface is a (usually empty) interface that serves as an abstraction for these types in the base program.

Forces. Use *marker interface* to abstract from join points when

- the join points are described using type information;
- the abstraction is used in both advice and ITD's;
- the join point abstraction is shared between multiple non-related aspects (i.e. the marker interface can be used also to describe other concerns);
- the aspect makes callbacks on the object underlying the JPA;
- the join point abstraction describes a homogeneous concern (i.e. the same advice applies to all connections of the marker interface).

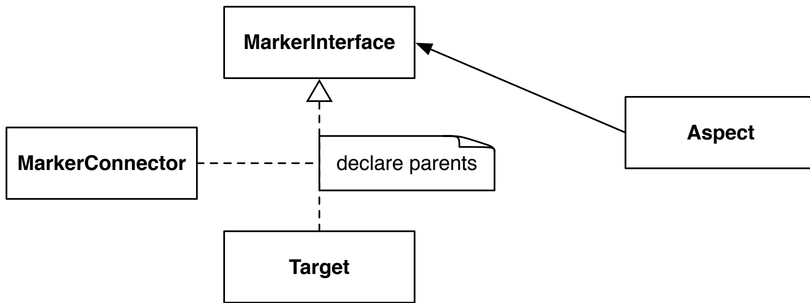

Figure 3.12: Structure of *marker interface*

Rationale. We can observe that the structure of join points related to the sensitive operations is always the same: the call of a method on a type `AccessObject` from an object of type `AccessSubject`. Only the exact interpretations of these types need further specification.

```
public aspect AccessControl {
  //...
  pointcut checkAccess(AccessSubject s, AccessObject o):
    this(s) && call(* AccessObject+.*(..)) && target(o);
}
public interface AccessObject{}
public interface AccessSubject{}
```

Listing 3.8: Using *marker interface* to abstract from types that define join points

If the marker interfaces are not defined locally (as in List. 3.8), they can also be used in other aspects. Adding methods to a marker interface puts more constraints on the join points at which the aspect might interact.

The marker interfaces from List. 3.8 could be connected as follows:

```
declare parents: Calendar || Resource extends AccessObject
declare parents: Employee extends AccessSubject
```

Listing 3.9: Connecting marker interfaces to concrete application types

The fact that a marker interface is an upper bound, means that it can be used to connect an aspect to multiple types at once, but also that advice cannot be made specific depending on a certain connection as the marker interface describes the same join points wherever it is used.

If a marker interface is used within a member introduction, this is sometimes referred to as a separate idiom named *Container Introduction*. For example, we could add to each protected resource a log in which all access attempts to that resource can be stored.

```
List<String> AccessObject.accessAttempts;
```

Implementation in other AOP technologies. CaesarJ supports bindings (wrapper construction). These can be used to introduce behavior (comparable to introductions). Unfortunately, these bindings cannot be used to abstract from join points. Advice in CaesarJ is part of the binding and will always be application-specific (in terms of the join points it interacts at, not in terms of the functionality it provides). JBoss AOP and Spring support inheritance declarations just as AspectJ (only by means of XML or annotations).

Related work and known uses. Hanenberg, Unland and Schmidmeier have described *marker interface* and its connection [51], also under the name *indirect pointcut connection*[4][47]. The AspectJ Cookbook [90] describes the *director* pattern. It consists of an abstract aspect with multiple roles as nested interfaces. These roles have the same purpose as a marker interface. *Marker interface* is an important concept in many of the refactorings in Monteiro's refactoring catalog [91, 92].

As *marker interface* is one of the most well known practices in AO design, it can be found in many open-source applications. To name a few: the AspectOPTIMA framework for transactions and concurrency [71]; the GoF design pattern implementations by Hanneman [52]; AJHotDraw [86], an AO implementation of JHotDraw [60]; HealthWatcher [43]; NVersion [8].

[4] And *indirect introduction* for *container introduction*.

Type parameter

Solution. Abstract from type information using type parameters in a generic aspect. These type parameters can be used in pointcuts and ITD's.

Forces. Use a type parameter to abstract from join points when

- the join points are described using type information;
- the abstraction captures the relation between multiple types;
- the abstraction is used in both advice and ITD's;
- the join point abstraction describes a heterogeneous concern (i.e. advice may vary for different sub-aspects);
- the join point abstraction is shared between related aspects (they could share a super-aspect);
- the abstraction captures certain methods and using wildcards is not helpful.

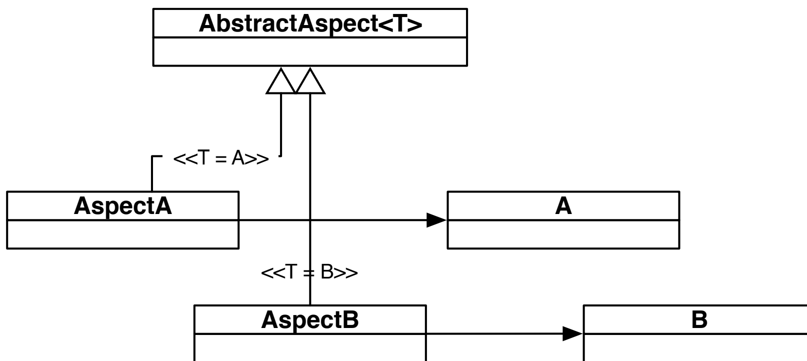

Figure 3.13: Structure of type parameter abstraction and connection

Rationale. Suppose we want to extend the aspect from List. 3.8 with a pointcut `accessAllowed` that describes which accesses should be allowed. Looking at the access rules from our example (Sect. 3.1.4), we see that the decision is based on who accesses what. Listing 3.10 shows how this can be represented in a pointcut using type parameters.

```
public abstract aspect AccessControl<AllowedSubj,AllowedObj>{
  //...
  pointcut accessAllowed(AllowedSubj s, AllowedObj o):
   this(s) && call(* AllowedObj.*(..)) && target(o);
}
```

Listing 3.10: Using type parameters in a generic aspect to abstract from types that define join points

Connecting a generic aspect to a concrete base application is accomplished through an (empty) aspect that extends the generic aspect with concrete types for its type parameters. E.g. binding the generic aspect from List. 3.10 could look as follows:

```
public aspect ResourceAccess extends
  AccessControl<Secretary,Resource>{}
public aspect CalendarAccess extends
  AccessControl<Employee,Calendar>{}
```

Listing 3.11: Independent connections of a type interaction using type parameters

Each concrete sub-aspect represents an access rule, e.g. `ResourceAccess` defines that a secretary can access a resource. Using marker interfaces (Sect. 3.2.2) it would not be possible to implement multiple independent rules. Because marker interfaces have a global meaning, the result would be that all types representing an `AllowedSubj` are allowed access to all types representing an `AllowedObj`.

A type parameter can also be used to parameterize a pointcut in terms of which methods to match. E.g. in List. 3.11 `CalendarAccess` defines the rule that each employee can access all calendar operations. How can we use type parameters if only a subset of calendar operations should be allowed to each employee? Listing 3.12 specifies this subset of methods in a separate interface and declares it to be a super-type of the target type. As a result it is able to use this interface as target type in the pointcut[5].

Implementation in other AOP technologies. CaesarJ doesn't include type parameters, it provides virtual classed instead. They are too limited to be used as abstraction for join points. Also in JBoss and Spring, type parameters are not available.

[5] This idiom works because of the semantics a type has in the different primitive pointcuts. Within a `call()` or `execution()` pointcut, the semantics of the type is that of a set of methods, while for instance in a `this()` or `target()` pointcut, a type represents a set of objects.

```
public abstract aspect
  AccessControl<AllowedSubj, AllowedObj, AllowedOps> {
  //...
  declare parents AllowedObj implements AllowedOps;
  pointcut accessAllowed(AllowedSubj s, AllowedObj o):
   this(s) && call(* AllowedOps.*(..)) && target(o);
}

public aspect AccessConnector{
  interface AccessOperations{
    Entry[] showEntries();
  }

  aspect ConcreteAccessControl extends
    AccessControl<Employee, Calendar, AccessOperations>{}
}
```

Listing 3.12: Using a type parameter to abstract from method information

Related work and known uses. The need for genericity when designing aspects had been discussed already before AspectJ introduced generic aspects [50, 85, 21, 75].

Hoffman and Eugster apply type parameters in their empirical study [57] and show their benefits with respect to the reusability of aspects.

Annotation introduction

Solution. Use an annotation to abstract from structural join points. Connect it to the base code by introducing the annotation using an ITD (or using a workaround if the relevant join points are only available as a pointcut).

Forces. Use *annotation introduction* to abstract from join points when

- the join points are described by annotatable program elements[6];
- the join point abstraction is shared between multiple non-related aspects;
- the abstraction is used in both advice and ITD's;
- the join point abstraction describes a homogeneous concern (i.e. the same advice applies to all connections of the marker interface).

[6] Currently these are methods, types and other class members.

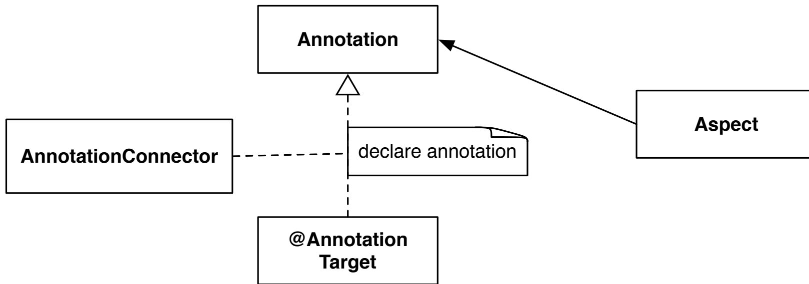

Figure 3.14: Structure of *annotation introduction*

Rationale. Annotations provide a flexible mechanism to add metadata to program elements. Multiple annotations can be attached and parameters can encapsulate metadata items.

We could for example define our `checkAccess()` pointcut in terms of annotations that are available in the base program.

```
public aspect AccessControl {
  //...
  pointcut checkAccess(): call(@CheckAccess * *(..));

  @Retention(RetentionPolicy.RUNTIME) @interface CheckAccess{}
}
```

Listing 3.13: An annotation that abstracts from sensitive method join points

Additionally, the aspect developer needs to define the annotation (usually with run-time retention[7]). The aspect is then defined in terms of this annotation as though it were part of the base program.

The following ITD's illustrate the introduction of the annotation `checkAccess` on both types and methods.

```
declare @type: Resource || Calendar: @CheckAccess;
declare @method: * Calendar.showEntries(): @CheckAccess;
declare @method: * Resource.book(Entry): @CheckAccess;
```

Implementation in other AOP technologies. CaesarJ has no support for annotations (yet). In JBoss AOP and Spring annotations can be introduced just like in AspectJ.

[7] So aspects have access to annotations attached to objects at run-time (necessary for `@this`, `@target` and `@args`).

Related work and known uses. The AspectJ notebook [120] describes how annotations are used, while Laddad elaborates on their potential [79, 80]. The combination of aspects and annotations is not exclusive to AspectJ, e.g. Havinga, Nagy and Bergmans discuss the use of annotations in Compose* [55].

Wampler uses annotations as one of the implementation strategies in Contract4J [129]. Annotations are also used by Cunha in his framework for parallel computing [27] and in the ajlib incubator project [9].

Summary join point abstraction

To wrap up the description of Join Point Abstraction, we give an overview of the main forces that drive the selection for the most appropriate idiom or combination thereof and briefly discuss their impact on ease-of-use.

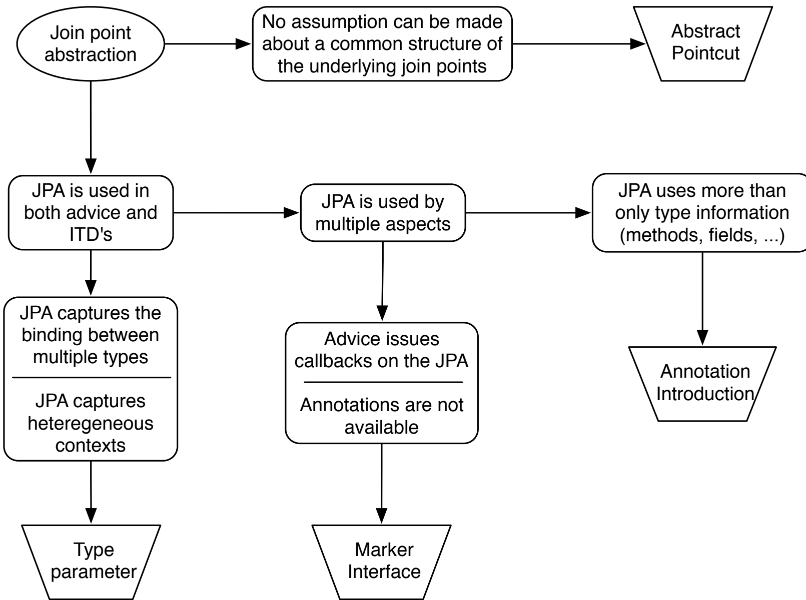

Figure 3.15: Design fragment join point abstraction

Usage of associated programming idioms. Figure 3.15 presents the design fragment for Join Point Abstraction. The figure guides the selection of the idiom that is appropriate in a certain situation. Navigation through the graph is the same as in Fig. 3.10.

The main factors that drive the selection are the way the join point abstraction (JPA) is used and whether the JPA is used by various aspects (which are not in an inheritance hierarchy). If the JPA is used in both advice and ITD's, *abstract pointcut* cannot be used. If the JPA is used by various aspects, *marker interface* and *annotation introduction* are most appropriate. *Abstract pointcut* is most appropriate when there is no (known) common structure among the join points. *Type parameters* are most suitable when the relation between multiple types needs to be captured (e.g. in the access control example the relation between the caller and the object that is called could be captured with type parameters) or the type abstraction represents heterogeneous contexts. Also for this pattern, the most suitable solution can be a combination of idioms, e.g. an abstract pointcut that is refined by a pointcut using annotations in a sub-aspect.

Ease-of-use. In terms of ease-of-use, the idioms either require an AO construction or some changes to the base code. These changes are the addition of annotations and marker interfaces. The required AO constructions can be as easy as an inter-type declaration (for binding a marker interface or annotation) or an empty aspect definition that binds a type parameter. Only abstract pointcut may require more knowledge about AO technology.

Connecting the abstractions

Table 3.1 summarizes how each abstraction can be connected to the base application in terms of the kind of join points that are exposed. Essentially, it points out how the adapter in Fig. 3.1 is implemented in terms of the interfaces it connects.

<table>
<tr><th rowspan="2">Join point abstraction</th><th colspan="4">Exposed join point</th></tr>
<tr><td>Structural convention</td><td>Naming Convention</td><td>Annotation Convention</td><td>Pointcut interface</td></tr>
<tr><td>Type parameter</td><td>Concrete type parameter</td><td colspan="2" rowspan="2">(ITD)*</td><td rowspan="2">Dynamic Type Wrapper</td></tr>
<tr><td>Marker Interface</td><td>Subtype/ITD</td></tr>
<tr><td>Annotation</td><td colspan="3">Annotation Introduction</td><td>Dynamic Annotation Introduction</td></tr>
<tr><td>Abstract Pointcut</td><td colspan="4">Concrete pointcut</td></tr>
</table>

* if the convention applies to the names or annotations of types

Table 3.1: Overview of the adapter idioms

Most of these cases are similar to what has been explained before (in the descriptions of the aspect awareness and join point abstraction patterns), except for *dynamic type wrapper* and *dynamic annotation introduction*. These are used in the case of a connection between a structural abstraction as provided join points (type parameter, marker interface and annotation) and a pointcut interface as exposed join point abstraction (e.g. base application provides a pointcut interface). The structure of both these workarounds is essentially the same. We need to wrap some run-time behavior into a program element. These techniques make the code rather complicated and should be avoided if possible. Let's have a look at *dynamic annotation introduction* first.

So, we have a pointcut with an annotation abstraction, e.g. from List. 3.13:

```
pointcut checkAccess(): call(@CheckAccess * *(..));
```

and we need to connect it to a pointcut exposed by a pointcut interface, e.g. from List. 3.3

```
public interface SensitiveOperations {
  pointcut sensitive(Object caller);
}
```

We therefore need to do two things: make sure that join points captured by `sensitive` appear to be method calls and that the method carries the annotation.

```
public aspect AnnotationConnector {
  interface SensitiveCall {
    @CheckAccess doProceed();
  }

  Object around(Object caller):
    SensitiveOperations.sensitive(caller) {
      new SensitiveCall(){
        @CheckAccess void doProceed() {
          return proceed(caller);
        }
      }.doProceed();
  }
}
```

Listing 3.14: Connection of an annotation to the join points of a pointcut

To make it look like a method call we need to use an anonymous inner class with the annotated method that incorporates the proceed context. The same structure can be used for *dynamic type wrapper* by making `SensitiveCall` a subtype of the type abstraction.

If the original aspect developer (of the aspect in List. 3.13) is able to anticipate such a situation, he can use a more general pointcut declaration that captures both annotated methods and annotated advice. For example,

```
pointcut checkAccess(): @annotation(CheckAccess);
```

uses the `@annotation` construct that matches any join point of which the subject has a certain annotation.

3.2.3 Decomposition

Problem. How to deal with aspects, pointcuts or advice of high complexity, that are implemented monolithically, compromising reusability and variability.

Context. Developing an aspect that might evolve and be composed with other aspects.

Abstract Solution. Decompose aspects into manageable parts according to one or more properties (variability, location, run-time dependency).

Abstract Forces. Use a decomposition idiom when

- the aspect defines basic functionality, used by multiple other aspects via extension;
- the aspect needs to separate between stable and variable parts;
- the aspect embodies too many dependencies:
  - aspect is connected to many base modules;
  - its pointcuts represent many concepts;
  - its advice specifies many actions.

Rationale. As an example of an aspect with high complexity, let's again take the `AccessControl` aspect (see figure 3.2). It should throw a security exception when an unauthorized access to a protected resource occurs. The aspect contains a pointcut that describes the unauthorized accesses and advice that throws the exception. We can see that the implementation of this aspect makes it difficult to adapt to future changes or to reuse it in a different context. There are multiple reasons for this. The behavior of the aspect is fixed. This hampers reuse of the aspect since its functionality cannot be refined when used in another context. The locations where this behavior is injected are specified by the pointcuts. They contain too much information, making them hard to understand and not easy to reuse or adapt.

Related work and known uses. Noble et al describe some sort of decomposition on the architectural level and call it heterarchical design [94].

The AspectOPTIMA framework [71, 73, 72] uses coarse-grained decomposition by providing a large number of small aspects to enable configurability.

Template advice

Problem. How to manage advice definitions that should specify behavior that, traditionally, would be defined in multiple methods or classes?

Solution. Decompose the behavior of a piece of advice into one or more *hook* methods. The advice itself specifies the stable structure of the behavior while the hook methods contain the variable parts of the behavior.

Forces. Use *template advice* when

- advice is subject to refinement;
- advice needs to be reused in different situations with some variability;
- advice functionality should be available as a method (so it can be called explicitly).

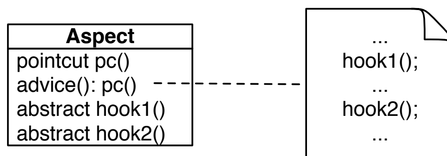

Figure 3.16: Structure of *template advice*

Rationale. In general the advice only retains the basic structure of the crosscutting functionality, which is not expected to change. Hook methods contain the variable parts (and can be left abstract).

Instead of giving our response to unauthorized accesses a fixed implementation in the advice body, it is better to delegate the actual response to a method that implements the appropriate behavior (as in List. 3.15). Even better would be to

```
public abstract aspect AccessControl {
  //...
  Object around(): accessDenied() {
    handleUnauthorizedAccess();
  }

  void handleUnauthorizedAccess(){
    throw new SecurityException();
  }
}
```

Listing 3.15: *Template advice* example

define this behavior in its own class and, thus, releasing it from the restrictions that aspects have with respect to reuse and evolution. Although *template advice* is vital in the context of anonymous advice (as in AspectJ), it also has its value on its own. By defining crosscutting functionality in a separate module and calling it from within advice, it is also accessible for other aspects and even ordinary classes.

If the behavior that needs to be activated by a certain `around` advice is defined in a separate class and needs to call the original behavior at the current join point (i.e. call `proceed`), how can we accomplish this? Because the keyword `proceed` is only available inside the body of the advice, we have to use a variant of the *Worker Object* idiom [81].

```
public abstract class UnauthorizedAccessHandler {
  public abstract Object doProceed();
  public void handleUnauthorizedAccess(){
    //...
    doProceed();
    //...
  }
}

public abstract aspect AccessControl {
  //...
  Object around(): accessDenied() {
    new UnauthorizedAccessHandler(){
     public Object doProceed(){
       proceed();
     }
    }.handleUnauthorizedAccess();
  }
}
```

Listing 3.16: *Template advice* based on Worker Object idiom

The class that defines the crosscutting behavior (`UnauthorizedAccessHandler` in List. 3.16) represents the behavior from the base program that it needs to call, with an abstract method (`doProceed()`). The problem is that calling this behavior is only possible from within the advice by using `proceed`. Our

only option is to use an anonymous inner class in the advice that extends the abstract class and implements the abstract method as a call to `proceed`.

Implementation in other AOP technologies. *Template advice* can be used in CaesarJ exactly the same way as in AspectJ. The same is true for JBoss AOP and Spring as there is no difference between advice and methods.

Related work and known uses. Hanenberg et al. present *template advice* [51, 47, 48], but do not describe the opportunity of defining crosscutting behavior in separate classes and the issues and benefits of doing so.

Example uses of *template advice* can be found in Cunhas concurrency framework [27], the Eclipse JDT weaving service [32] and Hannemans implementation of the GoF design patterns [52]. Examples where the implementation of proceed is outsourced to another class can be found in the worker object creation idiom [81] and the ajlib incubator project [9].

Template pointcut

Solution. Define the general structure of a pointcut by decomposing it into more basic hook pointcuts. These hook pointcuts describe the variable parts of the pointcut while the template itself remains stable.

Forces. Use *template pointcut* when

- the pointcut is too complex (it represents more than 1 concept, which compromises reusability);
- the pointcut contains both stable and variable parts.

Rationale. The template pointcut is decomposed into a number of (abstract) hook pointcuts, which can be overridden in sub-aspects. For instance, each sub-aspect could be appropriate for a different base application.

Let us look back at the pointcuts from List. **??**. A standard approach to determine unauthorized accessed is to define the total scope of access control and explicitly list all accesses that should be allowed. Therefore, the pointcut intercepts all calls to sensitive objects, except those actions that are explicitly allowed. The pointcuts, as they are now, use the notion of sensitive objects,

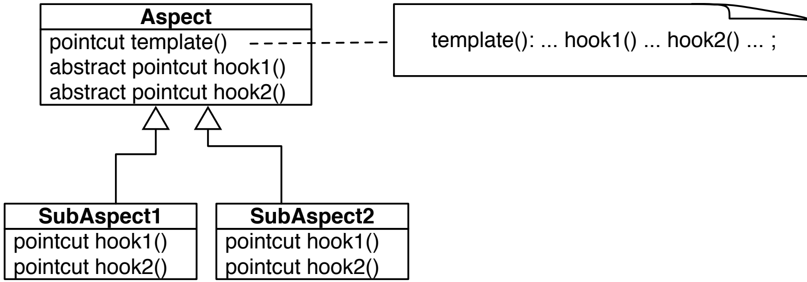

Figure 3.17: Structure of *template pointcut*

but this notion is not available as an abstraction because it is tangled with the access rules within the pointcut. How can we improve this situation?

We can define the general structure of the `accessDenied` pointcut in terms of two hook pointcuts: `checkAccess` and `accessAllowed`.

```
public abstract aspect AccessControl {
  pointcut accessDenied(): checkAccess() && !accessAllowed();
  abstract pointcut checkAccess();
  abstract pointcut accessAllowed();

  Object around(): accessDenied() {
   throw new SecurityException();
  }
}
```

Listing 3.17: *Template pointcut* example

As a result, the notion of sensitive objects and the access rules are specified separately and are thus reusable and evolvable. Of course, this pattern can be applied recursively in case a hook pointcut is still too complex.

The reusability of the super-aspect is largely determined by the general structure that is defined by the template pointcut. If the decomposition into hook pointcuts is orthogonal, these hooks become reusable and modifiable separately, without affecting each other.

Implementation in other AOP technologies. CaesarJ has better support for *template pointcut* than AspectJ. Not only abstract aspects can be reused; pointcuts can be refined, also in terms of the 'super' pointcut. Composite pointcut[8] can be used in JBoss AOP, but not *template pointcut* as there are

[8]Composite pointcut decomposes a concrete pointcut definition into a number of smaller concrete pointcut definitions. No abstract pointcuts are used.

no abstract pointcuts and pointcuts cannot be overridden. In Spring *template pointcut* can be used the same as in AspectJ.

Related work and known uses. This pattern is inspired by the classic *Template Method* design pattern from the GoF [37], which decomposes a method into hook methods for the same reasons. Lagaisse et al. describe the *template pointcut* pattern in more detail under the name of *elementary pointcut* [82, 17]. It elaborates on the consequences of applying the pattern in the context of pointcut inheritance and on the principles of how to decompose a pointcut into hook pointcuts (or elementary pointcuts). Hanenberg et al. describe the *composite pointcut* pattern [51, 47], which is very similar, but does not take aspect or pointcut refinement into account.

Santos describes the use of *template pointcut* to implement modular hotspots in frameworks [108]. In the ajlib incubator project, the tracing aspect also applies the *template pointcut* pattern.

Pointcut method

Solution. *Pointcut method* postpones part of the decision whether a certain join point should be advised until that advice is actually run, by first calling a method, the pointcut method.

Forces. Use *pointcut method* when

- a pointcut contains both static and run-time decisions;
- a partially different advice execution is needed depending on the outcome of a run-time decision .

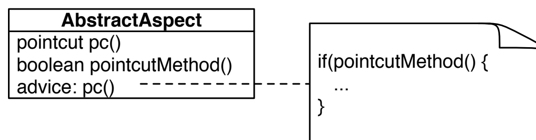

Figure 3.18: Structure of *pointcut method*

Rationale. *Pointcut method* is a variant of *template advice* (3.2.3), but since it is commonly used and has a specific purpose totally different from *template advice*, it is described separately.

The problem that *pointcut method* addresses is the presence of a dynamic condition. AspectJ offers a number of primitive pointcut constructs to capture run-time information. Examples are `this()` and `args()` which capture the type or identity of the current executing object and the current arguments respectively. For more general conditions concerning run-time information, the AspectJ pointcut language provides the `if()` construct. However, the if-expression that represents the dynamic condition cannot call non-static methods, which severely limits its potential for reuse.

With respect to our access control example, dynamic conditions are needed e.g. to check the access to a calendar for adding a new entry (as in the pointcut `calendarAccesDenied` in List. ??). This decision cannot be based on the types of the objects alone, because only the registered owner of the calendar can add new entries. If we expect this dynamic condition to change in the future, how can we prepare for it? By separating the dynamic condition in a method. As part of the pointcut is now represented as a method, it is more susceptible to future changes.

In the access control example, we can separate the ownership check into a pointcut method (`condition` in List. 3.18).

```
public aspect AccessAllowed extends AccessControl {
  pointcut calendarAccessAllowed(Employee e, Calendar c):
    this(e) && call(* Calendar.*Entry(..)) && target(c);

  boolean condition(Calendar c, Employee e){
    return e==c.owner;
  }

  Object around(Employee e, Calendar c): accessAllowed(e,c) {
    if(!condition(c,e))
      throw new SecurityException();
    proceed(e,c);
  }
}
```

Listing 3.18: Using *pointcut method* to capture a dynamic condition.

A common pitfall is the combination of *pointcut method* and around advice. As around advice totally replaces the original functionality at the current join point, `proceed()` (or any other default behavior) needs to be called explicitly in case the pointcut method decides the advice doesn't apply after all.

Implementation in other AOP technologies. As, mechanically, *pointcut method* is only a specific case of *template advice*, CaesarJ also supports it.

The same is true for JBoss AOP and Spring.

Related work and known uses. Hanenberg, Unland and Schmidmeier give an extensive description of *pointcut method* [51, 48]. Example usage of *pointcut method* can be found in AJHotDraw [86], an AspectJ implementation of JHotDraw [60], in the CommandObserver aspect. Also, the mobility aspect pattern [38] uses *pointcut method* in its implementation.

Participant connection

Solution. Instead of specifying one global connection for the whole base program, divide the connection in multiple participant connections, each integrated with a particular part of the program.

Forces. Use a *participant connection* when

- a pointcut becomes coupled to too many base modules;
- evolution of certain parts of the base program is important.

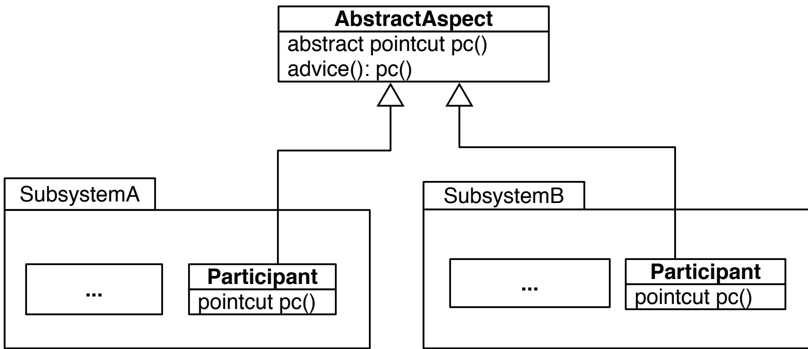

Figure 3.19: Structure of *participant connection*

Rationale. *Participant connection* is based on the Participant pattern described by Laddad [81], which proposes an abstract pointcut (Sect. 3.2.2) being implemented independently for each subsystem. The same reasoning can also be applied to the other join point abstractions.

Referring to the access control example, describing all accesses that need authorization in a single pointcut would lead to a fragile design. Because, in that case, nearly every change in the base program would have an effect on that pointcut.

The core structure of the pattern is as follows: specify one main abstract aspect that defines the main behavior, but is based on an abstract pointcut. Subsequently, for each subsystem, specify a sub-aspect that gives a concrete definition of the pointcut for that specific subsystem. Because the pointcut is now local to the join points it describes, it will be both simpler and more robust.

If we consider each class a subsystem, the participant aspect can be defined as a nested aspect in that class. E.g. binding an abstract pointcut `checkAccess` that represents all the sensitive operations, could look as follows:

```
public class Calendar {
  //...
  static aspect CalendarAccess extends AccessControl{
    pointcut checkAccess(): execution(* Calendar.*(..)) &&
      (execution(* showEntries())||execution(* newEntry(..))||
       execution(* deleteEntry(..)));
 }
}
```

Listing 3.19: Participant connection of an abstract pointcut.

Implementation in other AOP technologies. *Participant connections* are also possible in CaesarJ. Since, Caesar classes can contain pointcuts, there is no need for the inner class construction. Using XML, separate bindings can be specified for advice in JBoss AOP and Spring.

Related work and known uses. One specific case of a participant connection (which connects an abstract pointcut in a participant) is described by Laddad [81]. It is the original design pattern *Participant* and is later discussed in the context of annotations [79, 80].

Summary decomposition

To wrap up the description of Decomposition category, we give an overview of the main forces that drive the selection for the most appropriate idiom or combination thereof and briefly discuss their impact on ease-of-use.

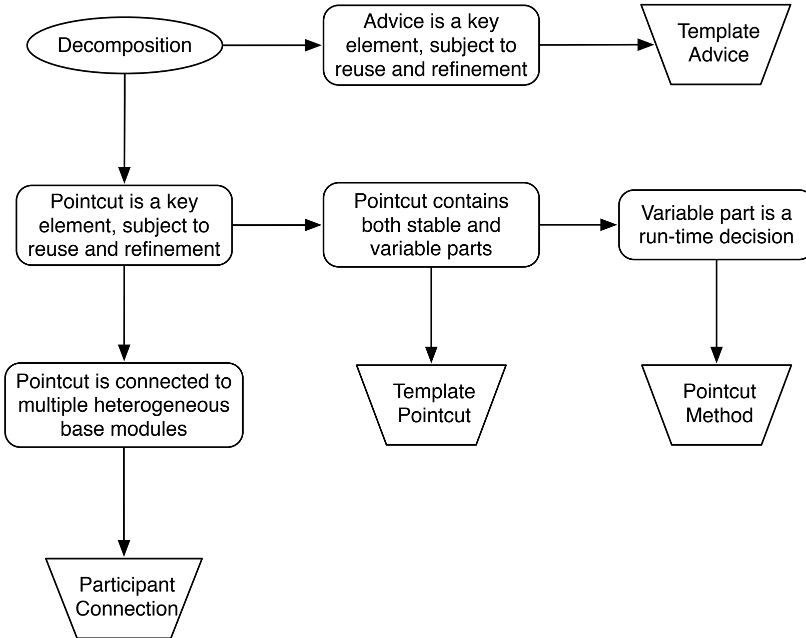

Figure 3.20: Design fragment decomposition

Usage of associated programming idioms. Figure 3.20 presents the design fragment for *Decomposition*. The figure links the different programming idioms with the main forces that drive selection of the appropriate idiom.

When choosing a decomposition idiom one needs to know what to decompose (pointcuts, advice) and based on which principle. If advice is a key element in the aspect *template advice* can be used to decompose it. For decomposing a pointcut there are three options. If the pointcut represents more than one concept, with both stable and variable parts, *template pointcut* is the solution. If the variable part is a run-time decision *pointcut method* is a better alternative. In the case that the pointcut is connected to varying kinds of join points with varying context structures (heterogeneous connection), *participant connection* can be used to provide a separate connection for each subsystem. Similar to the other design patterns, multiple decomposition idioms can be combined. E.g. a pointcut can be decomposed into a template, multiple hook pointcuts and a run-time decision using *template pointcut* and *pointcut method* respectively.

Ease-of-use. Decomposition idioms allow the developer to hide the aspectual elements from the user of the aspect library; or, at least, to make them more basic and thus easier to implement. *Template advice* and *pointcut method* enable to see respectively advice and pointcuts as normal method calls. *Template pointcut* and *participant connection* respectively make the pointcut to connect and the connection itself smaller in scope. Decomposition idioms will thus increase the ease-of-use of complex aspects.

3.2.4 Mediation

Problem. How to coordinate the behavior of a group of aspects that need to cooperate to achieve a shared goal?

Context. Aspect behavior is spread over multiple aspects, e.g. as the result of using a decomposition pattern.

Abstract solution. Provide a mechanism to regulate if and when an aspect needs to be activated (with respect to other aspects).

Abstract forces. Use a mediation idiom when

- decomposed aspects need to be coordinated to work as a team;
- aspect behavior is difficult to control if not coordinated;
- it is not feasible to bring the aspects back together into a single aspect.

Rationale. The need for mediation is often the result of the use of a decomposition idiom. E.g. if *template pointcut* (Sect. 3.2.3) is used, there can be multiple sub-aspects having their own implementation of a shared hook pointcut. For instance, based on the abstract aspect from List. 3.17, we can implement multiple sub-aspects, that, if we are not careful, will counteract each other. Let us take a look at List. 3.11. It defines two aspects based on the same generic aspect (List. 3.10). It is not apparent straight away, but these two aspects are in conflict (the overall behavior is that each access will be denied). The problem is that both aspects take a decision autonomously, which is allowing access in some specific cases and denying access in all other cases. Since allowing access is implemented as proceeding to the next aspect and denying access as throwing an exception, all accesses result in an exception.

The root of the problem is that each aspect has its own effect on the base program. We must achieve that each aspect can indicate its intentions and that a resulting action based on all the intentions takes effect in the application.

Related work and known uses. The principle of making cooperation explicit also applies to objects, as shown by the *Mediator* design pattern [37]. More closely related is the work of Schmidmeier [109] in which he describes two core patterns of aspects interacting and cooperating with other aspects.

Chained advice

Solution. Let the cooperating aspects interact at the same join point and define a chain of advice. The order of the chain can be defined explicitly (with a precedence declaration) or implicitly (when each chain element provides a join point for the next one).

Forces. Use *chained advice* when

- advice runs or can be attached at the same join point;
- there is a clear and fixed order between the different pieces of advice.

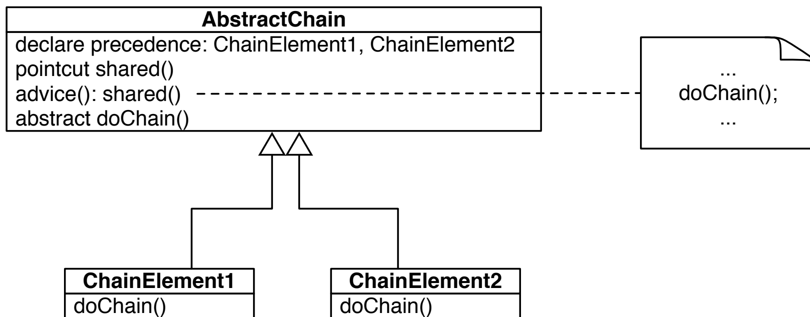

Figure 3.21: Possible structure of *chained advice*

Rationale. When two or more aspects define advice that might be activated at the same join point, the order in which the different pieces of advice will run is

undefined. Therefore, the developer should anticipate all possible interactions between such aspects and control their execution order.

Specifying the order of execution between the chain elements can be done explicitly or implicitly. An explicit order is specified using a precedence declaration. This keeps the chain elements independent of each other, but the declaration itself is static and unique per concrete composition of aspects.

If the chain elements don't necessarily have to be independent, the order of execution can also be specified more implicitly. In this case each chain element provides a join point for the next. Since advice is not a single identifiable join point, a workaround is necessary; e.g. by annotating the advice or using *template advice* (Sect. 3.2.3). Instead of the shared pointcut, the join point from the previous chain element needs to be specified in each aspect.

We can apply *Chained advice* to the access control example. We use it to enforce that all aspects first check the access rule they implement before a resulting action is taken. We therefore make a distinction between aspects that check the access rules and the aspect that will eventually grant or deny access. In List. 3.20 the abstract aspect from List. 3.17 is now split into three aspects.

```
public aspect AccessControl {
  declare precedence: AccessChecker, AccessGranter;
}

public aspect AccessChecker {
  pointcut accessAllowed(): ...
}

public aspect AccessGranter {
  pointcut accessDenied(): ...

  Object around(): accessDenied() {
    throw new SecurityException();
  }
}
```

Listing 3.20: Using *chained advice* to impose a certain order between different aspects

The aspect `AccessControl` is now only used to specify the precedence between the two other aspects, making sure that all access checks have occurred before access is definitely granted or denied.

`AccessChecker` uses the `accessAllowed` pointcut to indicate the intention of allowing this access. After all `AccessChecker` aspects have performed their task, `AccessGranter` finally takes action based on all the intentions. `AccessControl` and `AccessGranter` run at the same join points because the

join points captured by `accessAllowed` are a subset of those captured by `checkAccess`.

What is missing from the *chained advice* pattern is the information flow between the chain elements. E.g. in our access control example, how do the `AccessChecker` aspects provide the necessary information to `AccessGranter` so that it can take the appropriate action? The other two mediation patterns (*dynamic annotation introduction* in Sect. 3.2.4 and *mediation data introduction* in Sect. 3.2.4) can be used in combination with *chained advice* to accomplish this.

Implementation in other AOP technologies. Pointcuts can be shared in CaesarJ, Spring and JBoss AOP and also the precedence between different aspects can be declared, so there is no difference in implementation.

Related work and known uses. Hanenberg, Unland and Schmidmeier describe the *chained advice* pattern [51, 48].

Dynamic annotation introduction

Solution. Attach metadata to a certain control flow by using an annotated `around` advice that only calls `proceed`. The `cflow` primitive can then be used to check whether some property holds for a certain control flow.

Forces. Use *dynamic annotation introduction* when

- aspects interact at different join points;
- aspects cooperate within one message flow;
- different objects are used throughout the message flow.

Rationale. If we return to the example, we want to achieve that when one aspect decides access is allowed, the other aspect should no longer intervene. One way to do this is to annotate the control flow as being safe. We can now fully implement the aspect `AccessChecker` from List. 3.20 by adding an advice that annotates the control flow of join points in `accessAllowed` as being safe.

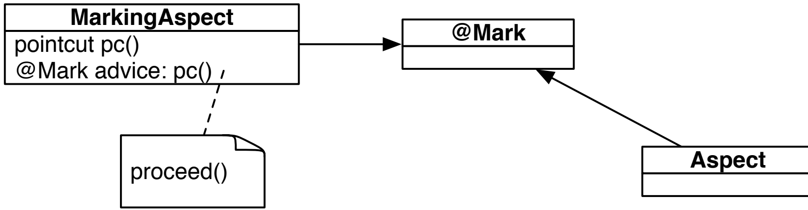

Figure 3.22: Structure of *dynamic annotation introduction*

```
public aspect AccessChecker {
  abstract pointcut accessAllowed();

  @Safe Object around(): accessAllowed(){
    proceed();
  }
}
public @interface Safe{}
```

Listing 3.21: *Dynamic annotation introduction*

The pointcut `accessDenied` from the aspect `AccessGranter` could then be implemented as follows (in combination with *template pointcut*, 3.2.3):

```
pointcut accessDenied(): checkAccess() && !marked();
abstract pointcut checkAccess();
pointcut marked(): cflowbelow(@annotation(Safe));
```

Implementation in other AOP technologies. There are no annotations in CaesarJ. A more general construction can be used: the combination of a hook method and `cflow`. Instead of annotating the advice, the advice will call the hook pointcut (i.e. a *structural convention*). JBoss AOP supports all the necessary mechanisms to implement *dynamic annotation introduction*. Due to the way weaving is implemented in Spring, *dynamic annotation introduction* cannot be implemented directly. Because aspects cannot be advised themselves, another indirection is needed to make things work.

Related work and known uses. De Fraine, Quiroga and Jonckers present control flow policies to statically verify aspect interactions [28]. *Dynamic annotation introduction* can be seen as one implementation technique to realize such policies at run-time.

Mediation data introduction

Solution. Introduce some mediation-specific data members into base objects and let each cooperating aspect manipulate this data. In the end a final action can be taken based on the final value of these data members.

Forces. Use *mediation data introduction* when

- aspects interact at different join points;
- the aspect cooperation spans multiple message flows;
- same object(s) is/are used by the cooperating aspects.

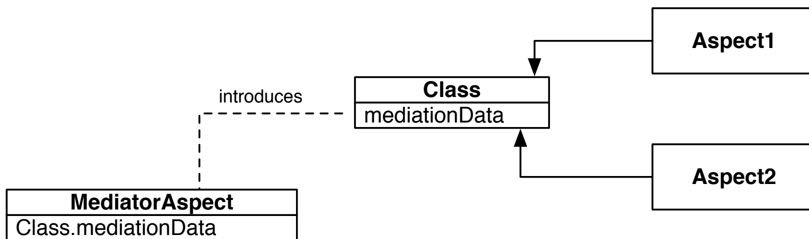

Figure 3.23: Structure of *mediation data introduction*

Rationale. Let's see how this pattern can be applied to resolve the conflict in the example. What we want to achieve is that if one aspect decides access is allowed, the other aspect should no longer intervene. *Mediation data introduction* solves this by attaching extra information to the base object that is being handled (in this case an Employee object). What happens is that when an aspect decides to allow access, it marks the employee as being allowed (e.g. setting a boolean flag to true). Now, before throwing an exception, the other aspect can inspect this information and act accordingly. Aspect `AccessChecker` would then be completed as follows:

```
public aspect AccessChecker {
  abstract pointcut accessAllowed(Employee e);

  boolean Employee.allowed = false;

  Object around(Employee e): accessAllowed(e){
    e.allowed = true;
    proceed(e);
```

```
    e.allowed = false;
  }
}
```

Listing 3.22: *Mediation data introduction*

The pointcut `accessDenied` from the aspect `AccessGranter` could then be implemented as follows (in combination with *template pointcut*, 3.2.3):

```
pointcut accessDenied(): checkAccess() && !marked(*);
abstract pointcut checkAccess();
pointcut marked(Employee e): this(e) && if(e.allowed);
```

Of course, if we want this aspect to be more reusable, we better replace the explicit reference to `Employee` with a marker interface (i.e. applying *container introduction*, 3.2.2) to reduce coupling.

Implementation in other AOP technologies. Because bindings in CaesarJ are unique for each aspect, the data that is introduced is not shared and can thus not be used for mediation. In JBoss AOP and Spring, this pattern can be implemented the same way as in AspectJ.

Related work and known uses. Example usage of *mediation data introduction* is the AspectOPTIMA framework [71] where some aspects introduce data to cooperate with other aspects (e.g. name, locks and checkpoints).

Summary mediation

To finish the description of Mediation, we give an overview of the main forces that drive the selection for the most appropriate idiom or combination thereof and briefly discuss their impact on ease-of-use.

Usage of associated programming idioms. Figure 3.24 presents the design fragment for *Mediation*. The figure links the different programming idioms with the forces related to mediation of aspects.

For the relatively simple coordination of aspects running at the same join point and having a fixed order, *chained advice* provides an easy solution. Otherwise, *mediation data introduction* or *dynamic annotation introduction* is needed, depending on whether the same objects are present throughout the interaction and whether the interaction spans multiple message flows. Since *mediation data introduction* works by injecting data into base objects it only works

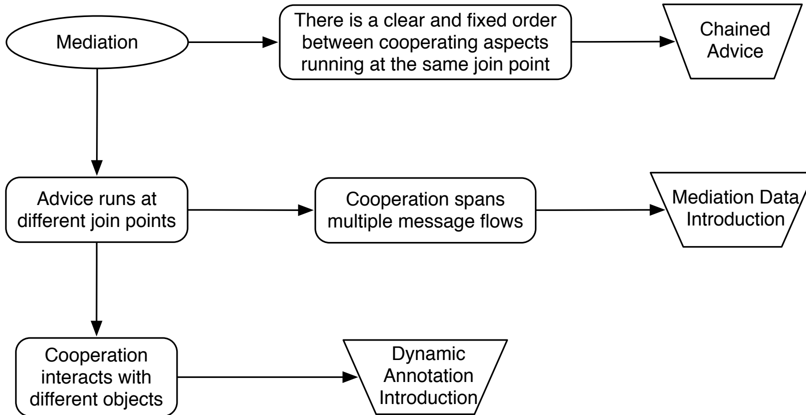

Figure 3.24: Design fragment mediation

when the same object(s) is/are used throughout the interaction. *Dynamic annotation introduction* works by annotating the message flow, which makes it inappropriate for interactions spanning multiple message flows.

For more complex interactions the current pattern definitions are probably not sufficient. E.g. it can become quite difficult to express the intention of an aspect without the aspect actually carrying it out. This probably requires a set of idioms on its own. Another issue that makes mediation a difficult design problem is the combination of determining execution order and resolution of conflicts between aspects. Conflict resolution is about determining which intentions of the aspects will take effect. Execution order is about which intention will take effect first. Currently, the mediation idioms don't handle this distinction well.

Ease-of-use. To have an easy-to-use aspect library, the designer specifies the cooperation of the aspect elements upfront as much as possible. Otherwise, precedence declarations can be used to define simple user policies. *Mediation data introduction* and *dynamic annotation introduction* are less adequate for letting users configure the aspect library.

3.3 Pattern generality

In this section we further discuss the generality of the pattern system with respect to different AOP approaches.

The pattern system is mainly focused on the pointcut-advice mechanism (PA). The programming idioms are targeted at AspectJ with variants available in other AOP languages or platforms for most of the idioms. The architectural pattern, however, presents a more general context that is also useful for AOP languages and platforms based on other AO mechanisms. In such a case, some of the concepts introduced by the architectural pattern become implicit. The language or platform enforces the use of the concept by integrating it in its decomposition style. Unfortunately, no AOP language or platform integrates all the concepts of our architectural pattern, which shows the general relevancy of our pattern system.

Event-based systems like Ptolemy [101], IIIA [116], Escala [41], IIT [100], EJP [57] provide the mechanism of explicit event announcement. As such they enforce aspect awareness of the base code while still enabling implicit and quantified composition. However, the other concepts remain relevant. Aspects (or in this case, event handlers) should abstract from the concrete events and concrete event context to be reusable across varying applications (Join point abstraction). Also, the interactions between the aspects are not simplified by using explicit event announcement (Mediation).

So-called symmetric AO formalisms like Multi-dimensional separation of concerns (Hyper/J [98]) and Subject-oriented programming [54, 97] make no distinction between aspect code and base code. In a way this is related to join point abstraction as each concern (think of a hyperslice or subject) will be defined separately without referring to the internals of other concerns. This context is similar to the oblivious case of our architectural pattern (see figure 3.2) with the difference that the connector contains all aspect-related functionality in the form of composition rules. As such, these connectors can become very complex and could benefit from concern modules that provide extension points similar to provided join point abstractions in Aspect awareness. The problem of aspect interaction becomes the more general problem of interaction between all modules as there is no difference between aspect modules and base modules. As a result the concepts of Mediation and Aspect awareness are alternative approaches to the same general problem of module interaction.

3.4 Example case: a reusable aspect for access control

This section presents a reusable implementation of an aspect for access control. This case has been used throughout this chapter as a running example. Here,

we will put together the pieces. The main goal is to give an example of how the different patterns and idioms can be combined. First we present the general framework of access control aspects (Sect. 3.4.1), which we will then use to apply access control to an oblivious base program (Sect. 3.4.2) and a base program that exposes sensitive operations by means of annotations (Sect. 3.4.3).

3.4.1 A reusable access control aspect

Based on the philosophy of stepwise refinement, the design of the aspect library consists of a common layer and two incremental layers that add new functionality, respectively, for managing more complex access control policies and for improving the ease-of-use of the overall aspect library.

Listing 3.23 shows the design of the common layer. As discussed in Sect. 3.2.4 on *chained advice*, we distinguish two distinct phases to allow multiple aspects to perform their checks before a decision is taken on granting the action. Therefore aspect `AccessControl` declares the precedence between these two phases. The precedence declaration uses the super-types `AbstractAccessChecker` and `AbstractAccessGranter` to make sure that every checking aspect executes before the aspect that grants the access. `AbstractAccessChecker` is an empty abstract aspect, used only to declare the precedence. `AbstractAccessGranter` contains the core functionality. It defines a template pointcut (Sect. 3.2.3) that specifies unauthorized accesses based on two abstract hook pointcuts. Furthermore it contains an abstract method that will be executed around each unauthorized access (by applying template advice, Sect. 3.2.3).

```
public abstract aspect AccessControl {
  declare precedence: AbstractAccessChecker+, AbstractAccessGranter+;
}

public abstract aspect AbstractAccessGranter {
  pointcut accessDenied(): checkAccess() && !accessAllowed();
  abstract pointcut checkAccess();
  abstract pointcut accessAllowed();                    Template pointcut

  Object around(): accessDenied() {
    handleUnauthorizedAccess();
  }
  public abstract void handleUnauthorizedAccess();      Template advice
}
public abstract aspect AbstractAccessChecker{}
```

Listing 3.23: Core functionality of a reusable aspect for access control

The next layer adds support for dynamic conditions. In List. 3.24 we therefore extend `AbstractAccessGranter` with `DynamicAccessGranter` to take into account the marks that will be set by `DynamicAccessChecker` (we included type parameters because we expect that dynamic conditions will be type specific, Sect. 3.2.2). Also a marker interface `AccessSubject` is introduced to abstract from all types that require access control. A boolean flag is added to this marker interface to enable marking the access decisions. This is achieved by combining mediation data introduction (Sect. 3.2.4) and marker interface with container introduction (Sect. 3.2.2).

```
public abstract aspect DynamicAccessGranter
  extends AbstractAccessGranter {
    pointcut accessAllowed(): marked(*);

    pointcut marked(AccessSubject subj):
      this(subj) && if(subj.allowed);                Mediation data introduction

    private boolean AccessSubject.allowed = false;

                                                     Container introduction
}

public interface AccessSubject{}                     Marker interface
```

Listing 3.24: Adding support for dynamic conditions

In List. 3.25 `DynamicAccessChecker` uses pointcut method (Sect. 3.2.3) to enable the users of the aspect to implement a dynamic condition (the default implementation always returns true, i.e. by default there is no dynamic condition). Based on this condition it marks the subject by applying mediation data introduction.

```
public abstract aspect
 DynamicAccessChecker<AllowedSubj, AllowedObj> extends
 AbstractAccessChecker {

    declare parents: AllowedSubj implements AccessSubject;

    abstract pointcut checkAllowed(AllowedSubj subj,
      AllowedObj obj);

    Object around(AllowedSubj subj, AllowedObj obj):
      checkAllowed(subj, obj) {
        if(condition(subj,obj)){
          ((AccessSubject)subj).allowed = true;
          proceed(subj, obj);                        Mediation data introduction
          ((AccessSubject)subj).allowed = false;
        }
        else proceed(subj, obj);                     Pointcut method
```

```
    }

    public boolean condition(AllowedSubj s, AllowedObj o){
      return true;
    }                                                Pointcut method
}
```

Listing 3.25: Implementing dynamic conditions using *mediation data introduction* and *pointcut method*

The final layer adds support for easy composition with oblivious base code. In List. 3.26 pointcut `checkAccess` is given a generic structure, easily made concrete by connecting both marker interfaces (another marker interface `AccessObject` is added). To support easy configuration of the access rules, the pointcut `checkAllowed` is given a generic structure based on three type parameters, describing the caller, the callee and the allowed actions respectively. `AccessChecker` uses a type parameter(Sect. 3.2.2) as an abstraction for method signatures.

```
public interface AccessObject{}                      Marker interface

public abstract aspect AccessGranter extends
 DynamicAccessGranter{
    protected pointcut checkAccess(): this(AccessSubject) &&
     call(* AccessObject+.*(..)) && target(AccessObject);
}

public abstract aspect
 AccessChecker<AllowedSubj, AllowedObj, AllowedAction> extends
 DynamicAccessChecker<AllowedSubj, AllowedObj> {

    declare parents: AllowedObj implements AllowedAction;
                                                     Type parameter

    pointcut checkAllowed(AllowedSubj s, AllowedObj o):
     this(s) && target(o) && call(* AllowedAction.*(..));
}
```

Listing 3.26: Adding support for easy composition of access control with an oblivious application

Note that the last two layers are optional. They can be skipped if not necessary for the application developer. For example the common layer can be directly used by refining the hook pointcuts of the *template pointcut* idiom. If dynamic conditions are needed the second layer can be used which adds a pointcut method. If ease-of-use is important, the bottom layer is appropriate. By deciding on a structure for the join points, the library can be easily configured by specifying the type parameters.

To summarize the design of the reusable aspect for access control, we now give an overview of the idioms that were used to implement each design pattern.

Join point abstraction. To implement the join point abstraction design pattern, we used a combination of *abstract pointcut* (in the common layer) and *marker interface* and *type parameter* (in the bottom layer). Marker interface and type parameter require a certain structure of the join points to interact with. To ensure versatility of the aspect library, abstract pointcut was used to enable connection to applications that do not satisfy this structure. We have chosen for type parameters to be able to capture the combination of caller and callee types, which is important for access control decisions. Marker interfaces were chosen to define the total scope of access control. Annotations would have been a suitable choice as well, since no callbacks are needed on the target join points. Additionally, a type parameter is also used to enable method-level access control instead of type-level access control.

Decomposition. Both pointcuts and advice were decomposed in this example. We used *template advice* to make abstraction of the action to be taken on unauthorized accesses. *Template pointcut* was used to split the concept of unauthorized accesses in two easier to define concepts (total scope and authorized accesses respectively). Furthermore, we separated the concept of a dynamic condition for authorized accesses using *pointcut method*.

Mediation. In this example we used a combination of *chained advice* and *mediation data introduction*. Two clear, distinct phases can be distinguished for access control. First, each rule determines its intention and second, a collective decision is taken and made effective in the system. *Chained advice* takes care of keeping the two phases separated, while *mediation data introduction* makes it possible to specify intentions without having an effect in the system. In this example, *dynamic annotation introduction* could be used as well for this purpose.

3.4.2 Composition with oblivious base program

Listing 3.27 shows how the reusable aspect described above can be connected to a base program that is not specifically prepared. First, we configure the total scope of access control by connecting the marker interfaces `AccessSubject` and `AccessObject` and activate the access granter by adding a non-abstract sub-aspect. Next, we specify the concrete access rules by providing empty aspects

that only give concrete types for the type parameters of `AccessChecker` and adding a dynamic condition if needed by overriding the `condition()` method.

```
public aspect AccessAllowed {

  declare parents: Employee implements AccessSubject;
  declare parents: Resource || Calendar
   implements AccessObject;                                   Marker interface

  static aspect ActivateGranter extends AccessGranter{}

  interface ResourceOperations {
    void book(Entry e);
    void cancel(Entry e);
  }

  interface ShowEntries {
    Entry[] showEntries();
  }

  interface SecretaryCalendar {
    void newEntry(TimeInterval t, Resource[] r);
    void editEntry(Entry e);
  }

  interface OwnerCalendar extends SecretaryCalendar {
    void newContinualEntry(TimeInterval first, Date last, Resource[]
         resources);
    void deleteEntry(Entry e);
  }

  static aspect Rule1 extends AccessChecker
   <Secretary,Resource,ResourceOperations>{}
  static aspect Rule2 extends AccessChecker
   <Employee,Calendar,ShowEntries>{}
  static aspect Rule3 extends AccessChecker
   <Secretary,Calendar,SecretaryCalendar>{}
  static aspect Rule4 extends AccessChecker
   <CalendarOwner,Calendar,OwnerCalendar>{
    public boolean condition(CalendarOwner co, Calendar c){
      return c.owner==co;
    }
  }
}
```

Listing 3.27: Example connector for an oblivious application to add access control

3.4.3 Composition with aspect-aware base program

How to compose access control with an aspect-aware base program depends strongly on the techniques used. Listing 3.28 shows how to connect the access control aspect with a base program that provides annotations to determine validity of accesses. Therefore we need to implement the hook pointcuts

`checkAccess` (which specifies all accesses that need to be checked) and `accessAllowed` (which specifies all accesses that need to be granted). We also need a helper pointcut (`correctAnnotation`) to be able to use variables that were not introduced by `accessAllowed`.

```
public aspect AnnotationGranter extends AbstractAccessGranter{
  pointcut checkAccess(): call(@Sensitive * *(..)) && @target(Sensitive
      );
  pointcut accessAllowed(): correctAnnotation(*,*);
  pointcut correctAnnotation(Sensitive s, Allowed role):
    @target(s) && @this(role) &&
    if(role.code().equals(s.allowed()));
}
```

Listing 3.28: Connecting an aspect-aware application using a sub-aspect leveraging annotations

3.4.4 Revisiting the requirements

We conclude the presentation of the integrated example with a brief discussion on how this implementation meets the requirements.

Versatility of the aspect library is achieved by defining the aspects in terms of join point abstractions. Abstract pointcuts in the common layer make it easy to connect the library to any base application by giving concrete definitions of the hook pointcuts. E.g. when the base application exposes stable abstractions using *Aspect awareness* these hook pointcuts can be easily defined. For easy configuration of an oblivious base application, the library also defines more structured join point abstractions. By deciding on a structure for the join points, the library can be easily configured by specifying the type parameters. Stability of the design is a result of having the choice between different styles of connecting, most suitable to a concrete context. Internal stability is achieved by decomposing the library in small aspects with small pointcuts and advice.

With respect to implementation, we see that a combination of idioms is used to realize each design pattern. We expect that this will be common for complex aspect libraries that should meet the requirement of versatility, stability and ease-of-use.

3.5 Conclusion

In this chapter we have presented a catalog of patterns and idioms aimed at the development of reusable aspects. Our focus was on aspects applicable to

various applications and application domains and the ease of configuring the aspect for a concrete context. Our pattern catalog consists of three layers: an architectural pattern, four categories (Aspect awareness, Join point abstraction, Decomposition and Mediation) and associated programming idioms. Each category focuses on a specific sub-problem, presents an abstract design and groups a set of programming idioms that can be used for implementing the abstract design. Each category additionally presents in general terms the forces and variation points that guide the application developer in choosing the appropriate idiom for implementing the abstract design depending on the specific development context.

We have described the patterns using a standard format and illustrated them by means of a running example regarding access control. We gave a holistic view on the running example which shows the use of the different patterns and idioms and the interplay between them. We also included the use of these patterns in academic and industry contexts.

Chapter 4

A sequence of patterns for reusable aspect libraries with easy configuration

The previous chapter showed that using well-known AspectJ idioms increases the reusability of aspect libraries. Availability of such reusable libraries is an important motivating factor to drive the further adoption of AspectJ in industry and aspect-oriented-programming in general. Existing work, however, mostly presents the existing AspectJ idioms as relatively independent solutions. As experience grows in using these idioms, it is possible to increasingly combine related idioms to form patterns and subsequently, pattern languages. A pattern language provides a structured process on how to apply a family of related patterns and idioms that cover a particular domain or discipline. This chapter presents a first step towards a pattern language for building reusable aspect libraries in the form of a sequence of aspect-oriented patterns that each combine a set of idioms to achieve (i) a configurable core design of the aspect library, (ii) library-controlled mediation of the interactions between the different aspects in the library and (iii) flexible configuration by providing multiple alternative modes for binding the aspect library to an application.

4.1 Pattern sequence overview

The availability of qualitative aspect libraries is an important driver to improve mainstream adoption of aspect-oriented (AO) technology [130]. Concrete examples of reusable aspect libraries exist [73, 27, 103], typically providing stable functionality with regard to a specific concern (transactions, persistence, etc.). Also typical is that configuration of such libraries can become complex and requires use of AO constructs.

In chapter 3 we have discussed the positive impact of applying AspectJ idioms on the versatility and stability of aspect libraries. As we will discuss further in chapter 5, the use of these idioms, however, has not significantly reduced the configuration complexity of aspect libraries. We believe that the underlying reason for this lack of improvement is the absence of the necessary guidance on how to combine the different idioms. In existing work, aspect-oriented idioms are mostly presented as relatively independent solutions. However, as experience grows in using these idioms, it is possible to increasingly combine related idioms to form patterns and subsequently, pattern languages.

This chapter explores the benefits of defining a pattern language that will provide the necessary guidance on how to combine the different idioms in order to achieve easy configuration while preserving versatility and stability of aspect libraries. In practice, configuration of aspect libraries is often achieved using annotations [114]. However, because of their inherent limitations, annotations alone are insufficient to realize configuration of aspect libraries in general. Annotations perform well in identifying the relevant join points, but are troublesome for specifying behavior and handling of interactions between different aspects.

The goal of this chapter is to present a first step towards a pattern language by describing by example a sequence of patterns for building reusable aspect libraries. The resulting libraries should be applicable to a wide range of applications without the need for complex configuration. This chapter presents the creation of the aspect-oriented architecture and design of a pricing library in terms of the pattern sequence. Although the patterns are valuable for AO composition in general, we focus on AspectJ for the example and its implementation. The pattern sequence was discovered while building several libraries. Although the specific application requirements in these libraries were different, the reusability and usability challenges were similar. This pattern sequence therefore embodies design expertise that can be reused in the domain of aspect-oriented libraries.

Our pattern sequence contributes by promoting a step-wise approach to develop reusable aspect libraries that employs existing idioms. In subsequent

steps it results in an extensible and configurable core design with library-controlled mediation of the internal aspect interactions and by providing multiple alternative configuration modes.

The pattern sequence consists of three patterns in the following order:

1. **Specify the core of the library.** The core consists of a behavioral part (what the aspect does) and a binding part (when it does it). This step ensures versatility and makes explicit the core abstractions to increase extensibility.

2. **Specify the mediation between resulting aspects.** In this step we control the internal interactions of the different aspects in the library through aspect mediation.

3. **Enable flexible composition.** To reduce complexity of configuration, we provide multiple alternatives to bind the library. Each alternative partially specifies the relevant join points using more structured abstractions like annotations or type parameters.

Each pattern in the sequence employs a combination of idioms from the catalog in chapter 3.

In the rest of this chapter we present the running example of a pricing library (Sect. 4.2), apply the pattern sequence to the pricing library (Sect. 4.3) and conclude in Sect. 4.5.

4.2 Case study pricing library

This section first presents the design of a pricing library as it resulted from a master thesis. The students were given a collection of patterns and idioms [19], but without any guidance on how to use them. Secondly, this section analyses the shortcomings of the resulting design with respect to the reusability qualities.

Core functionality of the library is to realize pricing rules according to certain promotions, taxes, service costs, etc. The library will apply modifications to the price of certain items as defined by the pricing rules. Additionally, the library enables conflict resolution between different pricing rules, e.g. a general promotion should not be applied to book items when a more specific promotion for books is applicable.

The case study is meant as a pedagogical example as pricing is understandable, but still sufficiently complex to illustrate the problems we wish to address. Also,

pricing is an interesting case for reusable libraries as it is relevant in multiple domains. For instance, in the master thesis the pricing library is also used as part of a real-time strategy game, where it is responsible for movement of units. Speed of movement depends on multiple factors, such as terrain type, proximity of enemy units, etc. Pricing (or business rules in general) has been used before as a target concern for AOSD [25, 29].

4.2.1 Original design of the pricing library

Figure 4.1 shows the design. Central to the design is `AbstractPriceAspect`,

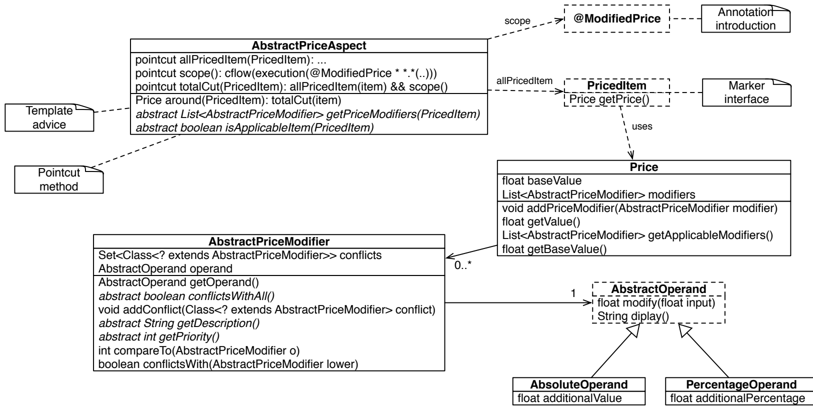

Figure 4.1: Design of the pricing library based pattern collection.

giving an abstract definition of pricing rules. It advises executions of `getPrice` on objects of type `PricedItem` (pointcut `allPricedItem`) in the control flow of a `@ModifiedPrice` annotation (pointcut `scope`) by means of the idioms *annotation introduction* and *marker interface*. Also *template advice* is used to abstract from the modifications (instances of `AbstractPriceModifier`) that need to occur (`getPriceModifiers`). The method `isApplicableItem` is a *pointcut method* to check if the pricing rule is applicable to the intercepted item.

The advice also employs a variant of *mediation data introduction* to control the combined behavior of the pricing aspects[1]. `AbstractPriceAspect` will add the `AbstractPriceModifier` instances from `getPriceModifiers` to the returned price object. Only when the resulting price is needed (by calling `getValue`)

[1] Instead of using the base object itself to store mediation data it uses a separate and specific Price object.

the different price modifications will take effect, taking priorities and conflicts into account (`getPriority` and `conflictsWith`).

To connect the pricing library to a concrete application, the user needs to do the following

1. provide a sub-aspect of `AbstractPriceAspect`
   (a) connect the marker interface `PricedItem` to base code elements
   (b) override `isApplicableItem`
   (c) override `getPriceModifiers`
      i. define a price modifier (sub-class of `AbstractPriceModifier`)
      ii. specify conflicts between certain price modifiers
      iii. specify priorities

An example configuration of the pricing library for a small web-shop example is depicted in Fig. 4.2. It defines three price modifications: a general promotion of 10%, a specific promotion for books of 20% and a fixed shipping cost. It also connects the type `Item` from the base code to the `PricedItem` type of the pricing library and connects the `@ModifiedPrice` annotation to activate the price modification in the necessary scope. In the next subsection we discuss the difficulties of this configuration and list the underlying problems of the pricing library

4.2.2 Limitations of the original design

Despite the fact that the library is designed with an extensive list of well-documented idioms for reusable aspects, there are still important limitations. The design is not optimal in terms of versatility and easy configuration. The main problems that lead to difficult configuration are as follows:

- Binding of the aspects to the relevant join points is difficult. The design suffers from the *callback mismatch* problem [20].
  - Using a marker interface with a signature complicates its binding (a combination of an inheritance declaration, an introduction and a redirecting around advice is needed, see yellow elements in Fig. 4.2)
  - Using only a marker interface prohibits diversification of intercepted items (all sub-aspects of `AbstractPriceAspect` will interact with the same items, leading to an extensive need of pointcut methods, see green elements in Fig. 4.2)

  - Suboptimal use of annotations (to bind annotation `@ModifiedPrice` to a constructor, a workaround is needed, see orange elements in Fig. 4.2).

- Mediation is cumbersome and error-prone

  - Resolving conflicts and handling priorities between pricing rules is scattered over the different pricing rules (see red and blue elements in Fig. 4.2)

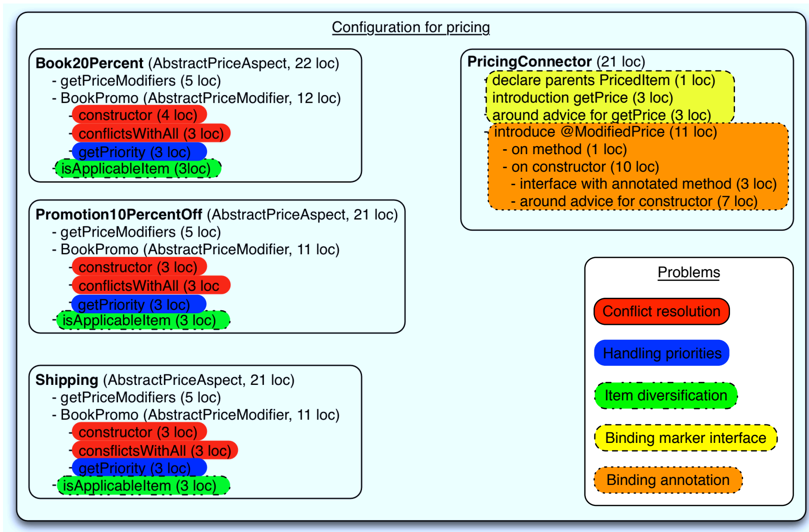

Figure 4.2: Example configuration for the original design of the pricing library.

4.3 Applying the pattern sequence to build reusable aspect libraries with easy configuration

In this section we apply the pattern sequence to the pricing library to improve its versatility and reduce complexity of configuration.

4.3.1 Core

Context. The design of a versatile and extensible library that provides crosscutting functionality.

Problem. How to define the core abstractions without tight coupling with the base code?

Solution. Specify the behavior (what the aspect does) and the binding (when the aspect does it) in abstract terms. In AspectJ this can be achieved using a combination of *abstract pointcut*, *template advice* and *pointcut method*.

Figure 4.3 depicts the core design of the pricing library using these idioms.

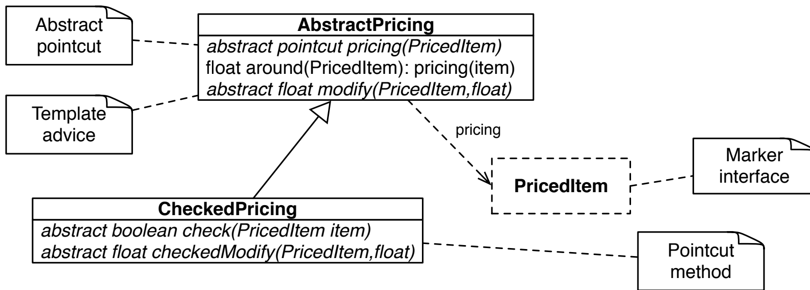

Figure 4.3: Core design of the pricing library.

It provides the core functionality in the most abstract way. `AbstractPricing` defines an abstract method (`modify`) that executes at the join points selected by an abstract pointcut, `pricing`, using the *template advice* idiom. `CheckedPricing` extends this basic functionality by introducing a *pointcut method*. Different sub-aspects of either `AbstractPricing` and `CheckedPricing` represent different price modifications.

To connect the core of the pricing library, the user needs to

- provide a sub-aspect of either `AbstractPricing` or `CheckedPricing` for each pricing rule
- give a definition for the `pricing` pointcut
- implement method `modify` (or `checkedModify`)

- optionally implement pointcut method `check`

The core design of the pricing library is extensible and versatile, but as a consequence, the user needs to define his own pointcuts, which shifts some of the complexity to the configuration. Also, it is not easy to control the interaction between the different price modifications. It can be done through the pointcut methods, but that solution doesn't scale to non-trivial interactions. Therefore, the next step deals with support for mediation between the sub-aspects.

4.3.2 Mediation

Context. Design of a reusable aspect library consisting of a group of interacting aspects.

Problem. It is difficult to control the resulting behavior of this group of aspects.

Solution. Separate the effect an aspect has from its intention. Instead of each aspect having an effect immediately, we encapsulate its effect (similar to the *Command* pattern [37]) and control the combined behavior through a mediator. Figure 4.4 presents the resulting design, showing new elements in grey. Each pricing aspect will add itself as a modification to the base object (`modifications` in `PricedItem`). It encapsulates its particular change in the price as an effect to be executed later. The `Mediator` aspect will introduce mediation data (`MediationData`, related to conflicts between different price factors) to the individual pricing aspects at the appropriate time, which, in this case, is when a price modifier that overrules another price modifier or might be overruled, is added to a `PricedItem` object (pointcuts `conflictAdded` and `conflictedAdded`). For pricing the mediation meta-data is concerned with possible conflicts. After all pricing aspects have made their intentions clear, the mediator can apply the correct changes based on mediation meta-data (advice on pointcut `finalPrice` and `conflictExecution`). In this case the mediator will skip the effect of pricing modifications that contain a conflict with another modification in this price calculation.

The aspect user can now specify pricing interactions as follows:

- define pointcuts `conflictAdded` and `conflictedAdded`, e.g. `execution(void PricedItem.addModification(..)) && args(conflict) && args(Book20Percent)` for `conflictAdded`.

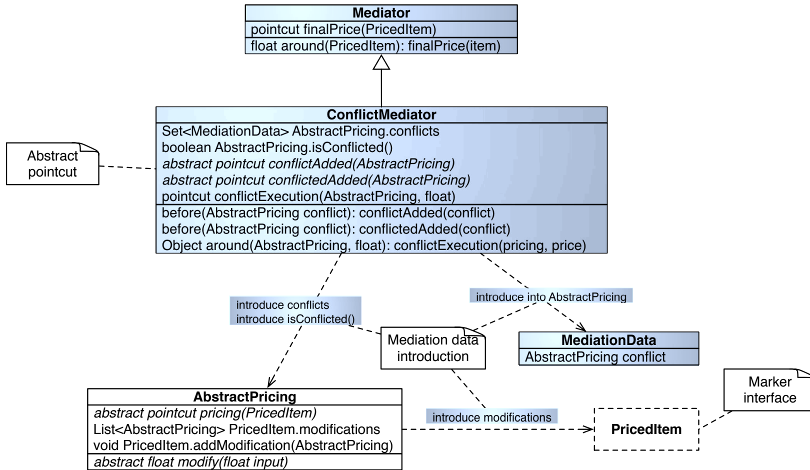

Figure 4.4: Design of the pricing library with support for mediation.

- declare precedence between the different price modifiers to control the order in which they will be added to an item and also will be executed

Complex mediation is now feasible in the library, but still configuration requires complex (aspect) definitions. We deal with this in the third step.

4.3.3 Flexible composition

Context. An aspect is versatile through the use of abstract pointcuts.

Problem. Binding the aspect means defining the pointcut descriptors, introducing complex and risky constructions in the configuration. How to ease aspect configuration?

Solution. Decompose abstract pointcuts to more fine-grained pointcuts and provide partial definitions.

In AspectJ this can be achieved with repeated use of *template pointcut* complemented with the use of structured abstractions like annotations, type parameters or marker interfaces. As a result, the aspect library remains flexible in how it can be configured. Figure 4.5 presents the complete design for

the pricing library[2]. *Template pointcut* is not necessary here, the pointcuts

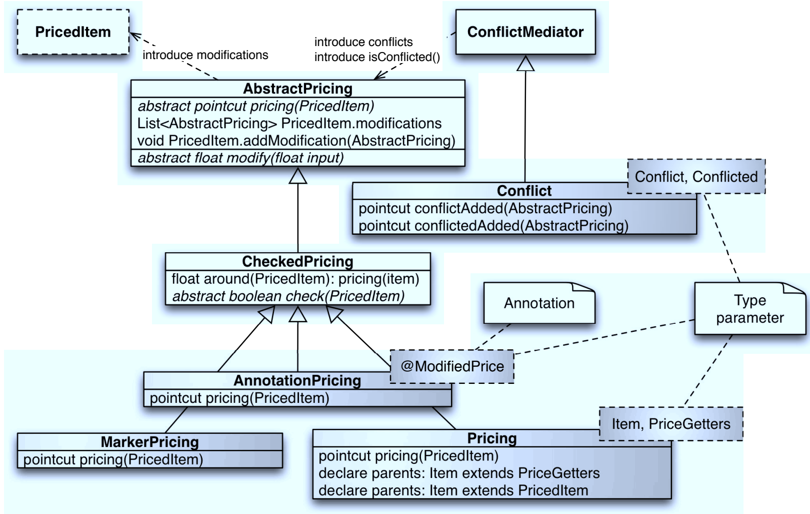

Figure 4.5: Design of the pricing library with flexible configuration.

`pricing`, `conflictAdded` and `conflictedAdded` are sufficiently fine-grained and can be represented directly using structured abstractions. Conflicts between price modifiers only really need the types behind the two conflicting price rules, so `conflictAdded` and `conflictedAdded` are implemented with *type parameters* in sub-aspect `Conflict`. For the `pricing` pointcut, more options are provided: via a marker interface (`MarkerPricing`), via annotations (`AnnotationPricing`) and via type parameters (`Pricing`).

To connect the resulting design of the pricing library, the aspect user needs to do the following:

- provide a sub-aspect of `Conflict` for each conflict between price modifiers
- provide a sub-aspect of either `Pricing`, `AnnotationPricing` or `MarkerPricing` for each price modifier
- provide a precedence declaration to control the order of price modifications

[2]Some details already represented in Fig. 4.4 are left out for clarity.

4.3.4 Connecting the aspect library to an application

Now that we have our complete design of the pricing library, we can have a deeper look at the configuration. Listing 4.1 gives the complete configuration of the pricing library for the same small web-shop example – general promotion (lines 15-19), specific promotion for books (lines 10-14), fixed shipping cost (lines 20-24). It also connects the type `Item` from the base code to the `PricedItem` type of the pricing library (line 2). One conflict is specified: when the promotion for books is activated, the general promotion is not valid (line 5). The precedence declaration specifies that shipping costs are added after promotions are dealt with (line 3). This configuration is very concise compared

```
1   public aspect PricingConfiguration {
2     declare parents: Item implements PricedItem;
3     declare precedence: Promotion10PercentOff, Book20Percent, Shipping;
4   }
5   aspect BookConflict extends Conflict<Book20Percent,Promotion10PercentOff>{}
6
7   public interface GetPrice {
8     public float getPrice();
9   }
10  public aspect Book20Percent extends Pricing<Book,GetPrice>{
11    public float modify(float input) {
12      return input*0.8f;
13    }
14  }
15  public aspect Promotion10PercentOff extends Pricing<Item,GetPrice> {
16    public float modify(float input) {
17      return input*0.9f;
18    }
19  }
20  public aspect Shipping extends Pricing<Item,GetPrice> {
21    public float modify(float input) {
22      return input+10f;
23    }
24  }
```

Listing 4.1: Example configuration of the complete pricing library

to the necessary code to configure the original design (less than half in terms of lines of code). If we compare this configuration with Fig. 4.2, we can note that handling of conflicts is now centralized; specification of priorities is achieved with 1 precedence declaration; binding of the marker interface is done with 1 inheritance declaration; and price items are diversified through type parameters. Chapter 5 presents a more proper evaluation of the benefits of the patterns and our pattern sequence.

4.4 Related work

Many sources exist that present good practices, idioms or patterns for AOP. Far less common are integrated solutions consisting of multiple elements that define a process towards a specific goal. Santos and Koskimies describe an

actual pattern language that employs AO patterns and idioms for the reuse of OO frameworks [108].

Other AO languages deal with the problems in this chapter by providing alternative means to bind aspects [101, 105]. Although these alternatives cause different problems with respect to the qualities discussed, we believe they would also benefit from our patterns on an abstract level. Also related are techniques that increase aspect awareness of base code. We gave an overview of such techniques in earlier work [19].

The complexity of aspect mediation has been discussed before. Schmidmeier [109] describes two core patterns for aspects interacting and cooperating with other aspects. Marot and Wuyts introduce some new AO language constructs to more easily compose aspects with other aspects [87].

The limitations we discussed with respect to the original design of the pricing library are related to code smells that give rise to refactorings [92] and implicit design assumptions [131] that limit the reusability of aspect code.

D'Hondt and Cibran have described how AO technology can be used to handle business rules (e.g. pricing) [25, 29]. A comparison with their results is future work.

4.5 Conclusions

In this chapter we have presented a sequence of patterns aimed at the development of reusable aspect libraries with easy configuration. Our contribution is the presentation of a sequence of patterns that each combines a set of idioms to achieve (i) a configurable core design of the aspect library, (ii) library-controlled mediation of the internal aspect interactions and (iii) flexible configuration by providing multiple alternative modes for binding the aspect library to an application.

The pattern sequence we present are based AspectJ idioms. Other AO languages, and also more advanced OO composition mechanisms, need to be studied in order to evaluate their potential to provide reusable solutions for the separation of crosscutting concerns. Candidate programming constructs are virtual classes, closures, mixins, implicit composition, ...

By adding more knowledge on the relations between the patterns and idioms, the definition of a real pattern language is a logical next step. Additionally, other qualities can be taken into account requiring the need of other pattern

sequences. New patterns and idioms will become relevant and other forces will drive pattern selection.

Chapter 5

Pattern evaluation: measuring impact on aspect reusability

In this chapter we evaluate the benefits of applying patterns for the design of reusable aspect libraries. This evaluation is based on the reusability qualities that formed the basis for the patterns in previous chapters: versatility, stability and easy configuration. We first present the approach and objective for the evaluation in section 5.1. Section 5.2 describes our practical experiences of applying the patterns to the implementation of aspects relevant in multiple real-world applications. Subsequently, we present a more rigorous metric-based evaluation of the benefits of using the patterns in two parts. Section 5.3 measures intrinsic properties of some of the implemented aspect libraries and uses them as predictors for the reusability qualities. Section 5.4 measures the qualities more directly in the context of some of the studied applications. We end the chapter with the conclusions (section 5.5) and limitations (section 5.6) of our evaluation.

5.1 Evaluation approach and objective

We perform a qualitative and quantitative study of the patterns. In the qualitative evaluation (section 5.2) we study a number real-world applications and identify recurring crosscutting concerns. These concerns lead to the implementation of reusable aspect libraries. We describe our experiences of implementing these libraries, with or without using the patterns from our catalog or in different combinations.

The objective of the quantitative evaluation (sections 5.3 and 5.4) is to measure to what extent the use of patterns and pattern sequence improves reusability qualities. More specifically, we want to study the following hypotheses:

- Using patterns and pattern sequence increases the versatility of aspect libraries.
- Using patterns and pattern sequence increases the design stability of aspect libraries.
- Using patterns and pattern sequence eases configuration of aspect libraries.

Our evaluation uses a selected subset of the applications and aspects from the qualitative study and is based on the assessment framework of Sant'Anna et al. [107]. Following their assessment framework, our quantitative evaluation consists of two parts. First, in section 5.3, we perform a predictive measurement by applying internal metrics, specific to each reuse quality, to different implementations of an aspect library. Internal metrics measure intrinsic properties of a system as opposed to metrics that measure properties of the system when put in context. In our study internal metrics only take the design of the library into account, while the direct metrics look at the library when applied to a particular application. The library implementations differ by their use of patterns and pattern sequence. The results give us a first impression on the impact of the use of patterns on reusability. In the second part, we use the libraries in the context of a set of applications to perform a more direct measurement of the reuse qualities (e.g. what proportion of the library can be reused by the application, how much effort does it take).

In both parts of the study, we analyze four different implementations of a reusable aspect library: (i) an AO implementation without explicit knowledge of AO patterns, (ii) two different AO implementations based on our collection of patterns and idioms and (iii) an AO implementation based on our pattern sequence.

5.2 Applying patterns: practical experiences

In this section we discuss our experiences with applying the patterns to the implementation of aspects relevant in multiple real-world applications. We first briefly overview the studied applications and secondly discuss the implemented aspects and their use of the pattern catalog. The study of the applications and the development of the aspect libraries were part of a master thesis of two students.

5.2.1 Studied applications

JadaSite. JadaSite is an e-commerce and content management application [58]. The application was developed in Java, has 5500 lines of code and uses a web-based interface. The following crosscutting concerns are identified: persistence, validation of user input, conversion of data to another data format and protection of all operations against unauthorized use.

Manage-My-Sales. MMS is an operational commercial web application. The application provides functionality to manage the current and future customer prospects and the associated deadlines and meta-information. Multiple users can access the system: the business managemer, sales representative, technical monitoring personnel, etc. Consequently all the information is centralized and online[1]. MMS uses Java EE, JSP and Hibernate. It is a medium sized application with 90 classes and 5000 lines of code. The identified concerns include persistence, logging, authorization and emailing.

Cities of Faith. Cities of Faith (COF) is a multiplayer online game developed in Java SE[2] This extensive project has 800 classes and 40,000 lines of code. Besides the presence of the Hibernate persistence framework and Swing GUI framework, most parts are customized. Persistence, authorization and input validation are the identified crosscutting concerns.

Cyber Conference Chair System CCCS assists with the administration and organization of conferences. A similar commercial application is Easychair [31]. The most important concern for CCCS is authorization.

[1] More detailed information is available at `www.managemysales.be`.
[2] The game is available at `www.cities-of-faith.com`.

USell. USell is a prototype e-commerce application. It provides services for the customer, the cashier and the manager. The identified concerns include persistence, input validation and flexible pricing strategies.

Overview. Table 5.1 gives an overview of the studied applications.

	JadaSite	MMS	COF	USell	CCCS
Language	Java	Java	Java	Java	Java
Type	open source	commercial	prototype	commercial	prototype
Size (LOC)	middle (5500)	middle (5000)	large (40000)	small (1500)	middle (7000)

Table 5.1: Overview of studied applications

5.2.2 Implemented aspects

Table 5.2 gives an overview of the implemented aspects. These aspects were

Argument validation	Checks the arguments when calling a method to make sure they satisfy the preconditions.
Emailing	Sending emails to interested users when relevant events occur.
Field injection	A practical alternative to singleton classes.
Persistence	A typical crosscutting concern that provides permanent storage of the state of run-time objects.
Authorization	This aspect controls the actions of certain objects for particular users.
Pricing	Pricing enable flexible pricing strategies with multiple factors like V.A.T. and transportation costs.

Table 5.2: Overview of implemented aspects

implemented by two master students as part of their thesis that aimed at the evaluation of the applicability of the patterns and idioms from chapter 3.

Argument validation. Validation of the actual parameter values is a widely recurring concern in many applications. For instance, checking that a string value is not null and not the empty string or checking that an integer is not negative. Aspects can provide an elegant solution for this problem. It increases consistency and comprehensibility of the code. A performance consideration is

that these checks are now replaced with a separate method call which can have an impact if the aspect compiler is not able to do the necessary optimizations.

```
aspect ArgumentController {
    pointcut controlArgument(Object o): execution(* * (@NotNull (*),
        ..)) && args(o,..);

    before(Object o): controlArgument(o){
      if(o==null) throw new IllegalArgumentException();
    }
}
```

Listing 5.1: AO implementation of argument validation

The aspect implementation (listing 5.1) uses *annotation convention* to specify declaratively in the base code the kind of check that is necessary. An extended implementation (listing 5.2) applies *template pointcut* and *template advice* to make the aspect reusable. These idioms make explicit the variation points of the `ArgumentController` aspect: the annotation to look for and the associated check. The aspect is not limited to the use of annotations (pointcut `annotationCut` is left abstract and can thus be given any definition), but they make most sense as they allow to select parameters on an individual basis.

```
aspect ArgumentController {
    pointcut controlArgument(Object o): annotationCut() && args(o,..);
    abstract pointcut annotationCut();                    Template pointcut

    before(Object o): controlArgument(o){
      checkArgument(o);
    }                                                     Template advice

    abstract void checkArgument(Object o);
}
```

Listing 5.2: Reusable AO implementation of argument validation

Following the guidance from figure 3.20 *template pointcut* and *template advice* are the right choices as both the pointcut and the advice are key elements subject to reuse and refinement (they are the variation points) and a dynamic check is not necessary here. Following the guidance from figure 3.15 *annotation introduction* is the appropriate idiom for its ability to capture specific join points (in this case individual parameters). However, due to AspectJ specifics, it is not possible to give a partial definition of the pointcut based on an annotation abstraction. *Abstract pointcut* is the best effort in this case.

Listing 5.3 shows two example configurations for the reusable argument validation aspect. The first implements the same functionality as listing 5.1

and the second checks whether integer arguments are positive based on the `@Positive` annotation.

```
aspect NotNullController extends ArgumentController {
  pointcut annotationCut(): execution(* * (@NotNull (*), ..));
  void checkArgument(Object o){//... }
}

aspect PositiveController extends ArgumentController {
  pointcut annotationCut(): execution(* * (@Positive (*), ..));
  void checkArgument(Object o){//... }
}
```

Listing 5.3: Example configuration of argument validation

Emailing. Emailing functionality is part of the MMS application and is scattered over different modules. The AO implementation (listing 5.4) combines *abstract pointcut*, *template advice* and *type parameter* to enable separation of this heterogeneous concern. The recipient, subject and body of the email vary with the kind of event that occurred. The use of *type parameter* allows to abstract from this heterogeneous context in combination with *template advice*.

```
abstract aspect EmailAspect<C>{
  abstract pointcut captureMailEvent(C context);                Abstract pointcut

  after(C context): captureMailEvent(context){
    sendEmail(context);
  }                                                             Template advice

  private void sendEmail(C context){
    String subject = extractSubject(context);
    String message = extractMessage(context);
    MailSender.get().addEmail(new Mail(subject,message));
  }
  protected abstract String extractSubject(C context);
  protected abstract String extractMessage(C context);
}
```

Listing 5.4: Reusable AO implementation of emailing

Following the guidance from figure 3.15, two forces are important here. First, we cannot make assumptions about the kind of join points that are relevant for emailing, this leads to *abstract pointcut*. Secondly, the context that is available at these join points will not have a uniform structure, leading to

type parameter. This aspect combines the two join point abstraction idioms in pointcut `captureEmailEvent`.

```
aspect NewProspectEmail extends EmailAspect<ProspectForm>{
  pointcut captureEmailEvent(Form f): execution(protected *
      ProspectForm.executeNew()) && this(f);

  protected String extractSubject(ProspectForm form){
    Prospect prospect = form.getNewProspect();
    String subject = "New prospect added - "+prospect.getCompany().
        getName();
    return subject;
  }
  protected String extractMessage(Form form){ //... }
}
```

Listing 5.5: Example configuration of the email aspect

Aspect `NewProspectEmail` shows the configuration of the emailing aspect in the MMS application for the event of new prospects. The context is defined as type `ProspectForm` and the hook method implementations define how to extract the needed data from this context.

Field injection. Aspects enable the injection of dependencies between objects. The implementation uses *annotation convention* for specifying that a certain dependency needs to be injected by the aspect.

```
public aspect Injector {

  private static HashMap<Class<?>, Object> values = new HashMap<Class
      <?>, Object>();

  public static void addInjectValue(@NotNull Object value) {
    values.put(value.getClass(), value);
  }

  private pointcut injectCut(Object o): get(@Inject * *.*) && target(o)
      ;

  before(Object o) : injectCut(o) {
    // if field == null set field with values.get(field.getType())
  }
}
```

Listing 5.6: AO implementation of field injection

Listing 5.7 shows how the user of the field injection library can easily use by adding values to the aspect and annotating dependencies with `@Inject` elsewhere.

```
public class SomeClass {
  @Inject private Database db;

  public void foo(){
```

```
    db.executeQuery();
  }
}
public class MainClass {
  public static void main(String[] args){
    Injector.addInjectValue(new Database());
    new SomeClass();
  }
}
```

Listing 5.7: Example use of field injection

Reusability for this simple aspect does not require the use of idioms besides *Annotation convention*. It is the most appropriate choice, because it allows identifying individual fields (not possible using other join point abstraction idioms as depicted in figure 3.15) and improves code readability.

Persistence. The persistence aspect enables the user to control persistent storage of objects using annotations. Both method annotations (for updating, `@PersistMe`, and deleting, `@DeleteMe`) and class annotations (for inserting, `@Entity`) are available. Additionally, annotations for explicitly enabling (`@EnablePersistence`) or disabling (`@DisablePersistence`) persistence are provided. A straightforward implementation (listing 5.8) using *annotation convention* is extended with *template pointcut*, *template advice* and *marker interface* to improve reusability (listing 5.9).

```
public abstract aspect PersistenceAspect {
  pointcut persistNew(Object newObject): execution(*.new(..)) && target
      (newObject) && within(@Entity *);
  pointcut persistChange(Object changedObject): call(@PersistMe * *(..)
      ) && target(changedObject);
  pointcut persistDelete(Object deletedObject): call(@DeleteMe * *(..))
       && this(deletedObject);

  public pointcut positiveScope(Object scope): execution(
      @EnablePersistence * * (..)) && target(scope);

  public pointcut negativeScope(): execution(@DisablePersistence * *
      (..));
  public pointcut totalScope(Object scope): cflow(positiveScope(scope))
       && !cflow(negativeScope());

  private pointcut executePersistNew(Object scope,Object newObject):
      totalScope(scope) && persistNew(newObject);
  private pointcut executePersistChange(Object scope,Object
      changedObject): totalScope(scope) && persistChange(changedObject)
      ;
  private pointcut executePersistDelete(Object scope,Object
      deletedObject): totalScope(scope) && persistDelete(deletedObject)
      ;

  after(Object scope, Object newObject):executePersistNew(scope,
      newObject){
    // implementation for persisting an INSERT
  }
```

```
  after(Object scope, Object changedObject):executePersistChange(scope,
       changedObject){
    // implementation for persisting an UPDATE
  }

  after(Object scope, Object deletedObject):executePersistDelete(scope,
       deletedObject){
    // implementation for persisting a DELETE
  }
}
```

Listing 5.8: AO implementation of persistence

The extended implementation makes explicit the variation points: when and what to persist (abstract hook pointcuts) and how to persist (abstract hook methods). The aspect is now independent from the specific annotations used in listing 5.8. Additionally, this implementation uses a marker interface to enable persistence for all methods of a type. As this is a homogeneous type abstraction, according to figure 3.15 both *marker interface* and *annotation introduction* are appropriate.

```
public abstract aspect AbstractPersistenceAspect {
  public abstract pointcut persistNew(Object newObject);

  public abstract pointcut persistChange(Object changedObject);

  public abstract pointcut persistDelete(Object deletedObject);
                                                    Abstract pointcut

  public pointcut positiveScope(Object scope):
    (execution(* PersistenceContext+.*(..)) ||      Marker interface
    execution(@EnablePersistence * *(..)) && target(scope);

  public pointcut negativeScope(): // same as before
  public pointcut totalScope(Object scope): //same as before

  private pointcut executePersistNew(Object scope,Object newObject): //
       same as before
  private pointcut executePersistChange(Object scope,Object
      changedObject): // same as before
  private pointcut executePersistDelete(Object scope,Object
      deletedObject): // same as before

  after(Object scope, Object newObject):executePersistNew(scope,
      newObject){
    performPersistNew(scope,newObject);
  }

  after(Object scope, Object changedObject):executePersistChange(scope,
       changedObject){
    performPersistChange(scope,changedObject);
  }

  after(Object scope, Object deletedObject):executePersistDelete(scope,
       deletedObject){
```

```
    performPersistDelete(scope, deletedObject);          Template advice
  }

  protected abstract void performPersistNew(Object scope, Object
      newObject);
  protected abstract void performPersistChange(Object scope, Object
      changedObject);
  protected abstract void performPersistDelete(Object scope, Object
      deletedObject);
}
```

Listing 5.9: Reusable implementation of persistence aspect

Listing 5.10 shows how to configure the reusable persistence aspect to obtain the same behavior as in listing 5.8.

```
aspect PersistenceConfig extends AbstractPersistenceAspect {
  pointcut persistNew(Object newObject): execution(*.new(..)) && target
      (newObject) && within(@Entity *);
  pointcut persistChange(Object changedObject): call(@PersistMe * *(..)
      ) && target(changedObject);
  pointcut persistDelete(Object deletedObject): call(@DeleteMe * *(..))
       && this(deletedObject);

  protected void performPersistNew(Object scope, Object newObject){
    // implementation for persisting an INSERT
  }
  protected void performPersistChange(Object scope, Object
      changedObject){
    // implementation for persisting an UPDATE
  }
  protected void performPersistDelete(Object scope, Object
      deletedObject){
    // implementation for persisting an DELETE
  }
}
```

Listing 5.10: Example configuration for reusable persistence aspect

Authorization. The authorization aspect controls whether a user has the necessary credentials to execute certain services (authentication is not included). Two different versions of the authorization aspect are implemented: one using *participant connection* (listing 5.11) and one using *annotation convention* (listing 5.12).

```
public abstract aspect ParticipantSecurityAspect {
  abstract pointcut accessedObject(Object accessedObject); Abstract pointcut

  public pointcut positiveScope(Object scope): (execution(
      @EnableSecurity * * (..))) && target(scope);
  public pointcut negativeScope(): execution(@DisableSecurity * * (..))
      ;
  public pointcut totalScope(Object scope): cflow(positiveScope(scope))
       && !cflow(negativeScope());
```

```
  private pointcut securityCut(Object scope,Object accessedObject):
      totalScope(scope) && accessedObject(accessedObject);

  before(Object scope, Object accessedObject):securityCut(scope,
      accessedObject){
    checkAccess(scope,accessedObject);
  }
  protected abstract void checkAccess(Object scope, Object
      accessedObject);
}
```

Listing 5.11: Authorization aspect using participant

Both implementations have their advantages. The implementation with participant uses *abstract pointcut* and allows the aspect user to define different authorization actions for different sub-systems (the force of heterogeneous connections in figure 3.20). The implementation based on annotations provides a more homogeneous approach with easy configuration, but requires annotating the base system (the force of homogeneous method-level join points in figure 3.15).

```
public aspect ParticipantSecurityAspect {

  public pointcut accessedObject(Object accessedObject): (execution(
      @SecurityCheck * * (..)) execution(@SecurityCheck new(..))) &&
      target(accessedObject);

  // same as other implementation
}
```

Listing 5.12: Authorization aspect using annotations

Pricing. The developed aspect library is the same library that illustrates the pattern sequence in chapter 4. Three different implementations are developed. We call these implementations $Pricing_{AO}$, $Pricing_{Pat}$ and $Pricing_{Pat_2}$. $Pricing_{AO}$ was implemented first without explicit knowledge of the patterns and idioms presented in chapter 3. $Pricing_{Pat}$ and $Pricing_{Pat_2}$ were both implemented after studying this collection, but use a different combination of idioms. Both use *pointcut method, template advice, annotation convention, marker interface* and *mediation data introduction*, but $\mathrm{Pricing}_{\mathrm{Pat}_2}$ adds *template pointcut* and *type parameter*.

The implementation of $Pricing_{Pat}$ is already extensively discussed in chapter 4. Listing 5.13 presents the main aspect of $Pricing_{Pat_2}$. It adds type parameters to support more heterogeneity in the pricing rules. For instance, the hook methods `isApplicableItem` and `getPriceModifiers` can now depend on the actual type of the priced item and implement a more specific pricing strategy (the force of heterogeneous context in figure 3.15 drives the choice for *type*

parameter). Also, *template pointcut* is added to give the user more control about when pricing strategies are relevant (add variability, see figure 3.20), but also to bind the heterogeneous context.

```
public abstract aspect AbstractPricingAspect<P extends PricedItem, C> {

  public pointcut pricedItemCut(P item):
    execution(Price PricedItem+.getPrice(..)) &&
    domainPricedItem(item) && !within(AbstractPricingAspect+);
  public abstract pointcut domainPricedItem(P item);        Template pointcut
  public abstract pointcut contextCut(C context);           Type parameter
  public abstract pointcut scopeCut();
   public pointcut totalcut(P item,C context):
    pricedItemCut(item) && contextCut(context) && scopeCut();
                                                            Template pointcut

  Price around(P item, C context): totalcut(item,context){
    Price price = proceed(item,context);
    if (isApplicableItem(item,context)) {
      for(AbstractPriceModifier modifier: getPriceModifiers(item,
          context)){
        price.addPriceModifier(modifier);
      }
    }
    price.display();
    return price;
  }

  protected abstract boolean isApplicableItem(P item, C context);
  protected abstract List<AbstractPriceModifier> getPriceModifiers(
    P item,C context);                                      Type parameter

}
```

Listing 5.13: Main aspect of pricing library $Pricing_{Pat_2}$

Overview. The following table gives an overview of which aspects are applied to which applications.

Pricing	MMS, COF, USell
Persistence	JadaSite, MMS, COF, USell
Authorization	JadaSite, MMS, COF, CCCS
Input validation	JadaSite, MMS, COF, USell
Emailing	MMS

The presented applications and aspect implementations have given us a wide experience and extensive knowledge about applying our patterns. In the

following sections we present a quantitative evaluation of a specific sub-set of these applications and aspect libraries.

5.3 Predicting reusability

This section describes the first part of the evaluation. We measure internal properties related to versatility, stability and easy configuration for multiple implementations of the same aspect library. We first sketch the experimental setup and define the measurement model. Secondly, we discuss the results and conclude the predictive evaluation.

5.3.1 Experimental setup

We compare four different implementations of an aspect library. We use the pricing library because it is the most extensive of the aspects that we implemented: it uses most patterns and has a need for mediation. Besides the three implementations discussed in section 5.2 we also evaluate $Pricing_{PatSeq}$, an implementation of the same library by the author of this thesis and based on the pattern sequence from chapter 4.

The library implements pricing rules that represent modifications to the base price of particular items. The library supports the definition of the modification (how exactly does the rule change the price), definition of the context of the rule (when is it applicable) and controlling the combination of different rules (e.g. ordering and conflicts).

5.3.2 Measurement model

For each of the qualities we use metrics that have been used before. We now briefly introduce the metrics used per quality.

Versatility. For the library to be versatile, it must be adaptable to many contexts. We therefore count the number of variation points offered by the library (**#VariationPoints**). This metric is based on Component Variability (CV) [24].

To count the variation points for each implementation of the pricing library, we first need to decide what exactly are valid variation points. We have taken the following into account:

- marker interface (exact interpretation can be specified for each application)
- mutable state (some variables can be changed to adapt behavior)
- abstract aspect (controls whether an aspect is effective or not)
- hook method (method that can be overridden to adapt advice behavior, this includes pointcut method)
- annotation (similar to marker interface)
- construction parameter (similar to mutable state, a parameter can be used to create an object)
- type parameter (specify the concrete type for an aspect)
- abstract pointcut (pointcut that can be given a concrete definition for a specific application)

Stability. To evaluate design stability of the aspect library, we measure modularity ($\mathbf{Modularity_{Aspect}}$) and complexity ($\mathbf{Conciseness_{Aspect}}$) of the internal design. To measure modularity and complexity, we use the coupling, cohesion and size metrics used in the assessment framework of Sant'Anna et al. [107]. We briefly define them here. The metrics for modularity:

- **CBC** (Coupling between components): counts the dependencies of a class, including types of formal parameters, return types and field types. Types of parent types are not counted.
- **DIT** (Depth of inheritance tree): counts the number of super-types until type `Object` is reached.
- **LCOO** (Lack of cohesion over operations): counts the pairs of methods that don't share a field minus pairs that do, but also pairs of pointcut/advice that don't share pointcut minus pairs that do.

The cohesion metric (LCOO) is extended to better fit the AO context. It now also includes pointcuts to determine cohesion of an aspect module. The size metrics are:

- **VS** (Vocabulary size): total number of modules (classes, aspects, interfaces).

- **LOC** (Lines of code): total lines of code (blank lines and comments not included).
- **WOC** (Weighted operations per component): measures the complexity of a component in terms of its operations (methods, advice, pointcuts). The weight applied to each operation is its number of parameters.
- **NOA** (Number of attributes): total number of attributes

We extended this set of metrics with the following aspect-specific measures: **NOPc** (number of pointcuts), **NOAdv** (number of advices), **DPrec** (number of precedence declarations) and **DPar** (number of parent declarations). These metrics count the total number of some AO constructs to give a better view on what constructs are used by the different implementations. Also, the metrics WOC and VS are extended to take, respectively, pointcuts and aspects into account.

Easy configuration. To achieve easy configuration, the library should provide a clean and clear interface that is easy to understand. We therefore measure the conciseness of the interface ($\mathbf{Conciseness_{Interface}}$) with the same metrics for conciseness as before, but now only applied to elements part of the interface of the library (elements the user needs to know to configure the library).

5.3.3 Predictive measurement

Versatility

As we can see in Table 5.3, the use of patterns increases the total number of variation points. The use of the pattern sequence doesn't increase the total number of variation points any further. It leads mainly to the use of other, more aspect-oriented variation points, especially abstract aspects and type parameters.

Stability

To predict stability, we measure modularity and conciseness of the library.

In Table 5.4 we see that metric DIT is not significantly affected by the method of development. We see that the use of patterns moderately decreases the coupling. The results for cohesion are not as useful as anticipated. The reason for this is that the aspects are mostly small and stateless, with no need for

	Marker interface	Mutable state	Abstract aspect	Hook method	Annotation	Construction parameter	Type parameter	Abstract pointcut	Total
$\text{Pricing}_{\text{AO}}$	1	3	4	1	0	0	0	0	9
$\text{Pricing}_{\text{Pat}}$	1	2	1	7	1	4	0	0	16
$\text{Pricing}_{\text{Pat}_2}$	1	3	1	7	0	5	2	3	22
$\text{Pricing}_{\text{PatSeq}}$	2	2	5	3	1	0	6	1	20

Table 5.3: Variation points of the pricing libraries

Pricing version	CBC	DIT	LCOO
$\text{Pricing}_{\text{AO}}$	13 (1.44)	13 (1.44)	0
$\text{Pricing}_{\text{Pat}}$	8 (1)	11 (1.38)	9 (1.12)
$\text{Pricing}_{\text{Pat2}}$	10 (1)	14 (1.4)	8 (0.8)
$\text{Pricing}_{\text{PatSeq}}$	11 (1.1)	12 (1.2)	2 (0.2)

Table 5.4: Modularity of the pricing libraries (total results and average per module)

pointcut sharing or methods sharing state, while the metric used for cohesion is based on the shared state between different methods (and the sharing of pointcuts in aspects).

The cohesion result for $Pricing_{Pat}$ and $Pricing_{Pat_2}$ can be clarified by one class that has considerable state and that is not present in the other implementations.

Pricing version	VS	LoC	NOA	NOM	NOPc	NOAdv	WOC	DPrec	Dpar
$\text{Pricing}_{\text{AO}}$	9	52	3	2	1	4	6	3	0
$\text{Pricing}_{\text{Pat}}$	8	175	6	23	3	1	17	0	0
$\text{Pricing}_{\text{Pat2}}$	10	218	8	28	7	1	26	0	0
$\text{Pricing}_{\text{PatSeq}}$	11	151	4	13	7	4	19	0	2

Table 5.5: Conciseness of the pricing library

Table 5.5 shows that the use of patterns leads to an increase in all conciseness metrics except for VS, NOAdv (different pieces of advice are generalized into one) and DPrec (a more implicit mechanism for mediation was used). With the pattern sequence the number of modules slightly increases, while the LoC

remains more or less the same. The use of typical OO constructs (fields and methods) seems to be replaced with pointcuts and advice.

Easy configuration

For ease-of-use we analyze the configuration interface of the aspect. With configuration interface we mean those modules that the aspect user needs to understand to configure the library.

We apply the same metrics for conciseness as in the stability measurement, but now only applied to elements on the configuration interface.

Pricing version	VS	LoC	NOA	NOM	NOPc	NOAdv	WOC	DPrec	Dpar
$\text{Pricing}_{\text{AO}}$	6	31	3	2	1	4	3	3	0
$\text{Pricing}_{\text{Pat}}$	7	113	4	17	3	1	16	0	0
$\text{Pricing}_{\text{Pat2}}$	9	152	6	21	7	1	24	0	0
$\text{Pricing}_{\text{PatSeq}}$	7	73	2	5	6	4	14	0	1

Table 5.6: Conciseness of the configuration interface

Most noticeable in Table 5.6 is the decrease in size metrics (LoC, WOC) and OO constructs (NOA,NOM) when using the pattern sequence. This not the case for AO constructs (NOPc,NOAdv,DPrec,Dpar). The relation of the results for using patterns or not is similar to the conciseness results for the aspects as a whole.

The predictive measurement gives us a first impression of the impact of using patterns on the reuse qualities of aspect libraries. Our conclusions are presented in section 5.5 together with those from the more direct measurement in the next section.

5.4 Assessing reusability

This section describes the second part of the evaluation. We measure versatility, stability and easy configuration for multiple implementations of the same aspect library in the context of multiple applications. We first sketch the experimental setup and define the measurement model. Secondly, we discuss the results and conclude the reuse evaluation.

5.4.1 Experimental setup

The setup for our validation consists of three applications: USell, a prototype sales application; Cities Of Faith (COF), an online multiplayer game and Manage My Sales (MMS), an operational commercial web application. The aspects studied in these applications are pricing, authorization and argument validation. Most attention is given to the pricing aspect library. It provides support for adding extra price factors like, V.A.T., transportation costs, promotions, etc. Each implementation of the aspect is applied to each application, without changing the aspect library (black-box reuse). We also have simulated a number of change requests (CR) that impose new requirements on the applications with respect to the different aspectual functionalities. For the USell application the change requests are as follows:

1. CR0 (base): default integration of pricing; it requires one tax modifier and a general promotion for all products.
2. CR1: it should be possible to get an overview of all the separate price factors that contributed to the final price.
3. CR2: ability to add specific promotions that are only active for selected products.
4. CR3: ability to distinguish business clients; their overview doesn't contain taxes, although these taxes contribute to the final price.
5. CR4: taxes should depend on client location.
6. CR5: ability to introduce service costs; service costs depend on the product price and are susceptible to promotions in the same way as products.
7. CR6: ability to add couple promotions; the cheapest of two media products is now free.
8. CR7: deactivate promotions on items that are involved in a couple promotion

The change requests for MMS and COF are similar. As they are only used indirectly for versatility, we briefly present them here[3]:

- MMS application: summer-time discounts, promotion for premium business customers

[3] A complete description is available in appendix.

- COF application[4]: paying players can perform more moves, moving a group of units together is less expensive

We used a somewhat different setup for evaluating the patterns than for evaluating the pattern sequence. In the setup for the pattern evaluation, each change request triggers changes in both the configuration of the aspect library as the base system. As one of the key motivating factors for the pattern sequence is easy configuration, we changed the setup so that each change request only triggers changes in the configuration and the base system remains untouched. This allows a more direct comparison between the different configuration efforts of the pricing library implementations.

5.4.2 Measurement model

For each of the qualities we use metrics that have been used before. We now briefly introduce the metrics used per quality.

Versatility. To evaluate versatility in the context of an application, we measure how much functionality of the library that is used (*%LibUse* based on Component Reuse Level [24]). The degree of library use is defined as the lines of code (LOC) that are actually used by an application divided by total LOC of the library. We also evaluate how much functionality of the library is re-implemented in the connector (*CoDu*). We use the DuDe tool to count the amount of duplicated text blocks across the application source code[5].

Stability. To evaluate stability, we count the added and changed pointcuts for each change request.

Easy configuration. To evaluate easy configuration, we measure the effort to implement each change request in terms of lines of code (LOC).

[4] In the context of COF, pricing is used to determine the cost of moving units (soldiers, ships) across the board. E.g. moving units through enemy territory is more expensive.

[5] Dude (Duplication Detector) is a tool that uses textual comparison at the level of line of code in order to detect fragments of duplicated code. Its powerful detection engine can also cover various adaptations of the duplicated code (such as variables renaming or statement insertion/removal) – http://loose.upt.ro/iplasma/dude.html.

5.4.3 Reuse measurement

Stability

To give an idea of the impact of the patterns on stability we compared the number of changed and added pointcuts for each change request of the USell application[6]. In counting the pointcuts we also take into account anonymous pointcut definitions that are part of an advice definition, unless they refer to a single named pointcut.

	Base	CR1	CR2	CR3	CR4	CR5	CR6	CR7	Total
$\text{Pricing}_{\text{AO}}$	0	9	5	3	0	18	3	0	38
$\text{Pricing}_{\text{Pat}}$	0	6	0	2	0	4	4	0	16
$\text{Pricing}_{\text{Pat}_2}$	3	8	0	0	0	5	5	0	21

Table 5.7: Added and changed pointcuts for each change request in the USell application

On average, the implementations using patterns require less pointcuts to be added or changed. This is especially noticeable in CR5, where new pointcuts related to service costs are introduced and existing ones are refactored. Also note that $\text{Pricing}_{\text{Pat}_2}$ needs three pointcuts right from the beginning. The reason is that this implementation uses a generic aspect with abstract pointcuts that need an implementation before it can become active.

To give an idea of the impact of using the patterns sequence we repeated the above experiment.

Figure 5.1 shows that using our pattern sequence, less pointcuts are needed and thus also less pointcuts need to be changed. This is the result of using structured abstractions, taking away the need for the user to define pointcuts. As a result, the configuration is more robust[7].

Versatility

To give an idea of the impact of the patterns on versatility we have compared the degree of library use for the 3 alternative implementations of the pricing

[6] The numbers for the other applications were not meaningful as they made extensive use of the *pointcut method* idiom.

[7] The results for $\text{Pricing}_{\text{AO}}$ and $\text{Pricing}_{\text{Pat}}$ are different from the first experiment because a different configuration methodology was used. In the first experiment, configuration includes changes to the base code, while the second experiment reuses the base application as is.

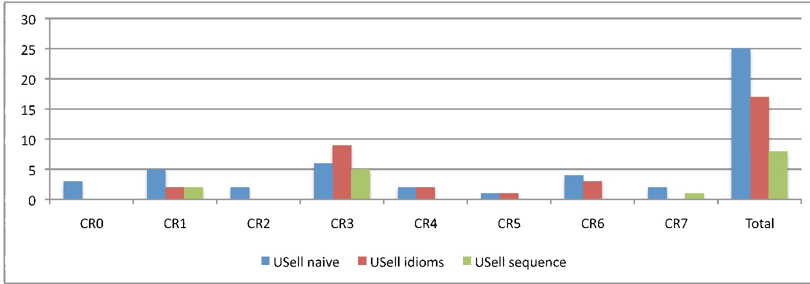

Figure 5.1: Added and changed pointcuts for each change request in the USell application.

library. We show the trend in library use by giving the value after the first change request and the value after all change requests. We would expect that as more change requests have been performed, more functionality of the aspect library is used. The results are as follows:

Degree of library use (%)	USell	MMS	COF
$\text{Pricing}_{\text{AO}}$	57→23	50→23	40→17
$\text{Pricing}_{\text{Pat}}$	89→99	99→99	89→99
$\text{Pricing}_{\text{Pat}_2}$	77→85	84→85	76→84

Table 5.8: Degree of library use for the pricing aspect after 1 change request and after all change requests

Not only do we see that the degree of library use for the patterned versions is higher, also their trend is upward, while that of the version without patterns is downward. As the requirements for pricing become more complex with more change requests, more functionality of the pricing library is needed. As the implementation without patterns does not provide the necessary abstractions, that functionality cannot be reused.

Degree of library use (%)	USell
$\text{Pricing}_{\text{AO}}$	63→35
$\text{Pricing}_{\text{Pat}}$	89→99
$\text{Pricing}_{\text{PatSeq}}$	62→95

Table 5.9: Degree of library use for the pricing aspect after 1 change request and after all change requests.

Table 5.9 shows the result for the experiment with the pattern sequence[8]. It shows that both the use of idioms and the pattern sequence lead to an increased reuse of library functionality. Initial reuse of the library developed using the pattern sequence is lower because not all modules need to be configured for trivial cases.

The decreasing trend of library use can also be considered from the perspective of the amount of code duplication on top of the library. If an application cannot reuse the functionality from the library, we expect that the application will need to define this functionality itself, leading to a duplication of code. To confirm this expectation, we have measured the amount of duplicated code in the USell application before and after integrating the pricing library. The following table gives the amount of blocks of duplicated code after integrating $\text{Pricing}_{\text{AO}}$, $\text{Pricing}_{\text{Pat}}$ and $\text{Pricing}_{\text{Pat}_2}$ (we expect similar results for $\text{Pricing}_{\text{PatSeq}}$). The USell application without the pricing library contains 11 blocks of duplicated code.

Amount of duplicated code blocks (%)	USell
$\text{Pricing}_{\text{AO}}$	15→37
$\text{Pricing}_{\text{Pat}}$	13→27
$\text{Pricing}_{\text{Pat}_2}$	14→21

Table 5.10: Amount of duplicated code in USell application before and after integration of the pricing aspect

We see indeed that the pricing implementation without patterns triggers more duplication in the USell application code than the other versions.

Easy configuration

To measure the effort of integrating the aspect library with a certain application we analyze the configuration for each change request in terms of conciseness (lines of code).

Figure 5.2 shows significant improvement. Providing structured abstractions really helps in managing the complexity of the configuration by encapsulating this complexity in the library as much as possible.

Furthermore, we compare the different used programming idioms to each other with respect to their impact on configuration. We perform this comparison

[8] Also here, different numbers because of the different configuration methodology.

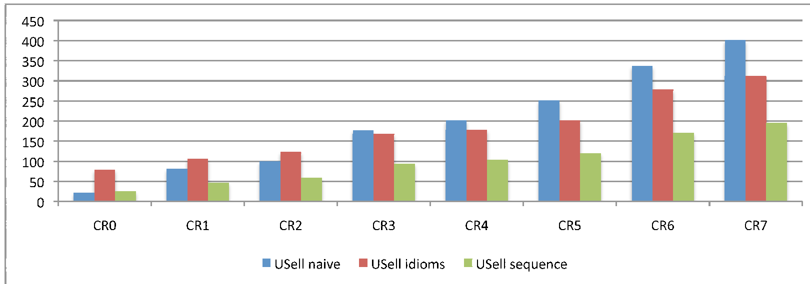

Figure 5.2: Conciseness of the configuration for each change request in the USell application (in LOC).

by measuring the total LOC of the application before and after integrating different aspects. Table 5.11 shows the results.

Total LOC	USell	MMS	COF
Without pricing	1518	4828	38700
$\text{Pricing}_{\text{Pat}}$	2213	5103	38888
$\text{Pricing}_{\text{Pat}_2}$	2197	5109	38882
Authorization (participant)	n/a	5012	39203
Authorization (annotations)	n/a	4930	38799
Argument validation (OO)	1463	4922	39156
Argument validation (annotations)	1396	4813	38700

Table 5.11: Total LOC for different implementations of the aspects

For authorization, we compare the use of annotations with the use of participant connection. For argument validation, we compare annotation-based pointcuts with a traditional object-oriented implementation. As can be expected we see that the use of annotations requires a significantly lower LOC (and thus on ease-of-use). Annotations allow the developer to ignore boilerplate code for authorization and argument validation by providing metadata with the method signature or on the arguments themselves.

For the two implementations of the pricing aspect there are no significant differences in LOC. From these numbers no conclusion can be made with respect to the impact on ease-of-use for these different combinations of idioms.

5.5 Study conclusions

Now that we have discussed the results for the predictive and direct measurements, we make our conclusions that can be used as hypotheses for further validation in a more extensive empirical study.

Versatility. The main increase in variation points was achieved by applying patterns. Using the pattern sequence had more impact on the kind of variation points than on the total number of them. This will probably improve easy configuration more than it does versatility. In the second part, we indeed see an increased level of reuse due to the use of patterns. The use of the pattern sequence had no further impact. We therefore make the following conclusion with respect to versatility:

Using patterns increases the versatility of aspect libraries, i.e.

> *Applications can reuse more functionality of libraries that are implemented using patterns.*
>
> *Applications need to re-implement less functionality of libraries that are implemented using patterns.*

Stability. No clear conclusions can be made on the impact of using patterns with respect to the internal modularity of the library, although coupling seems to decrease. We see that using patterns increases the code size of the library, but also that this is mitigated by applying the pattern sequence. In the second part we evaluated pointcut stability and we could see that the use of patterns leads to less pointcut changes. A result that is further improved by using the pattern sequence. We therefore make the following conclusions with respect to stability:

Using patterns and the pattern sequence increases the stability of aspect libraries, i.e.

> *Over multiple change requests, libraries implemented using patterns and the pattern sequence suffer less change impact on pointcuts.*

Easy configuration. As with stability, patterns increase the complexity of the configuration interface, the price of improving versatility, which, subsequently is mitigated by using the pattern sequence. The impact of using patterns on configuration complexity depends on the complexity of the change request. For relatively easy configurations (early change requests) using patterns is not cost effective; this is only the case for more complex configurations (later change

requests). Using the pattern sequence on the other hand almost immediately returns on the investment. We therefore make the following conclusion with respect to easy configuration:

Using the pattern sequence decreases the configuration cost of aspect libraries, i.e.

Aspect libraries implemented using the pattern sequence require less LOC to configure.

5.6 Study limitations

Although this evaluation is not meant as a conclusive empirical study, we believe that these results give a good impression on the applicability of the patterns and the benefits of applying them. To assess the real value of these results we analyze the threats to the validity. This in combination with the fact that the patterns or idioms are used in real-world applications should give them sufficient credibility as worthwhile solutions to the design problem of reusable aspect libraries.

Construct validity. Stability, versatility and ease-of-configuration are attributes that are difficult to measure. The metrics used for reuse [36] and stability [43] have been used before. The metric used for ease-of-use is less established, but size is always an important indicator for effort [84].

Internal validity. The most important threats to internal validity are the bias of the students and the author towards promoting the patterns and the learning effect as the implementations without patterns were implemented first. Also the difference in experience in using AOP between the master students and the author has an effect on the difference in quality of the implementations.

External validity. The size and number of the applications and aspects used is not sufficient to make general conclusions about the benefits of the patterns in a real-world context. Also, all code in the study was written in AspectJ, which limits the portability of the results to other AOP technologies.

5.7 Conclusions

In this chapter, we presented an initial validation with respect to the reuse qualities. Initial evaluation shows improved reusability and configurability in the context of a pricing example. The results give a first impression on the applicability of the pattern sequence and the benefits of applying it and shows that the way different patterns and idioms are combined has a big impact on the reusability properties of an aspect library. We also showed that versatility and stability improve with the use of patterns and that annotations are more easy to use than other idioms (a result that can be expected).

The results give a good impression on the applicability of the patterns and the benefits of applying them. In combination with the fact that the patterns and idioms are used in real-world applications should give them sufficient credibility as worthwhile solutions to the design problem of reusable aspect libraries.

Chapter 6

Conclusion

The availability of reusable aspect libraries is an important factor in the mainstream adoption of AOSD. Although software reuse has always been a challenge, there are some specific difficulties for developers of reusable aspect libraries due to the crosscutting nature of aspect-oriented composition. Aspects are defined in terms of characteristics of the base program. As a consequence, there is a tight coupling between aspects and base. The challenge thus is designing aspects in terms of abstractions that are not coupled to any specific application. At the same time, an aspect library should hide the complexity associated with the aspects, in order to ease configuration of the library by the user.

In this thesis we have incepted, described, classified and integrated idioms and patterns for the architecture, design and implementation of reusable aspect libraries. The main contribution and outcome of our work is a structured catalog of patterns and a step-wise approach for developing reusable aspect libraries with easy configuration.

6.1 Revisiting requirements and contributions

Throughout this thesis we have focused on three qualities for reusability: stability, versatility and easy configuration. We now give a retrospective overview of our achievements with respect to these qualities.

Stability. The nature of the coupling between aspects and base is different than that of the coupling between objects. Because of the quantifying abilities of AO composition there is a need for a stable, but flexible, definition of the relation between aspect and base, in order to make this relation resistant against modification of a related module. Stability of aspects reduce the effort and risk of maintenance operations that have an impact on the relation between aspect and base. It is the pointcut that mainly embodies the aspect coupling. In chapter 2, we have analyzed the fragility of these pointcuts, a well-known threat to the stability of AO systems [117, 63].

In our catalog of patterns (chapter 3), we have presented *Aspect awareness* and *Join point abstraction*, two categories of idioms that deal with the binding between aspect and base. The use of these idioms lead to an interface that decouples aspects from the concrete join points of an application. *Aspect awareness* realizes this from the point of view of the base code by exposing potentially relevant events as extension points that can be used by aspects. *Join point abstraction* is similar but from the point of view of the aspect. The fragile pointcut problem is not solved per se, but it will be less severe as its impact is now localized and encapsulated by the connection between aspect and base.

In chapter 4 we have presented a pattern sequence[1] that takes this one step further by looking at the structure of the interface between aspect and base. The last step in the sequence, *Flexible composition*, describes how to decompose join point abstractions to multiple fine-grained and easy-to-configure variation points. As a result no, or at least less complex pointcuts need to be defined to configure an aspect library. Consequently, the fragility of the binding between aspect and base is reduced.

Our evaluation in chapter 5 shows indeed that the use of patterns induces less pointcut changes in an evolution scenario involving multiple change requests. The evaluation also shows that the use of the pattern sequence further improves this result.

Versatility. Versatility means that the aspect library can be reused across a range of different applications. The assumptions an aspect makes about the structure or behavior of the base code it interacts with should not conflict with the requirements of the application. Consequently, the challenge for aspects is to define the relevant join points without referring to specific details of the base program. In a sense, similar difficulties to achieve stability need to be solved.

[1] A pattern sequence represents a process made up of smaller processes —its constituent patterns— arranged in an order that achieves a particular architecture or design in response to a specific situation [11].

Additionally, the quantification challenge, described in chapter 2, affects the ability of the aspect to capture the correct join points in terms of abstractions.

In our catalog of patterns (chapter 3), we have presented *Join point abstraction* and *Decomposition*. *Join point abstraction* idioms enable the developer of the aspect library to define the relevant join points in terms of abstractions, while *Decomposition* idioms improved the internal design of the library. The combination of a clean aspect design with an interface that specifies the required join points increases the adaptability of the library.

In chapter 4 the sequence of patterns streamlines this approach by first making explicit the core abstractions of the library (*Core design*) and in a later step providing fine-grained join point abstractions (*Flexible composition*). The result of this process is that the important concepts of the library are represented explicitly and are sufficiently fine-grained to enable composition with various applications.

In chapter 5 we have implemented some common reusable aspects using the patterns and applied them to multiple applications. The experience taught us that the patterns helped making the aspects reusable. This result was confirmed more quantitatively by our study that shows an increased level of library reuse for library implementations using patterns.

Easy configuration. Configuration of aspects involves identifying the relevant join points, specifying the crosscutting functionality and fine-tuning how different aspects should work together. Without proper design, the configuration itself will consist of relatively complex AO constructs, threatening the widespread use of the library. Libraries will typically consist of multiple different aspects, each acting independently. As a result it is necessary to specify configuration rules to regulate the interaction between the different aspects. In chapter 2 we have discussed the problem of aspect interaction in more detail. Also relevant for easy configuration is that the library should minimize quantification challenges for users of the library.

Chapter 3 presents *Mediation* idioms to deal with cooperating aspects. For each category of idioms, the catalog discusses the ease-of-use. The pattern sequence (chapter 4) generalizes and improves the approach towards mediation in a separate step of the sequence. Also, the final step (*Flexible composition*) shows how to combine *Decomposition* and *Join point abstraction* idioms to provide multiple modes of configuration to minimize the remaining quantification challenges the library user has to deal with.

The experience in chapter 5 shows that some patterns (e.g. *annotation convention*) ease configuration while others require more effort or AO expertise

(e.g. *participant connection*). The quantitative study shows indeed that using patterns not consistently eases configuration. Easy configuration is achieved using the pattern sequence that decreases the effort (in terms of code size) to configure the aspect library for multiple scenarios of integration.

Summary. In this thesis we have presented a catalog of patterns and idioms aimed at the development of reusable aspects. Our focus was on aspects applicable to various applications and application domains and the ease of configuring the aspect for a concrete context. The catalog consists of three layers: an architectural pattern, four categories, each focusing on a specific sub-problem, presenting an abstract solution design for that sub-problem and consisting of a set of programming idioms that can be used to implement the abstract design. Each category additionally presents in general terms the forces and variation points that guide the application developer in choosing the appropriate idiom depending on the specific development context. We have augmented this catalog with a step-wise approach to guide the development of reusable aspect libraries with easy configuration. It presents a sequence of patterns that each combines a set of idioms to achieve (i) a configurable core design of the aspect library, (ii) library-controlled mediation of the internal aspect interactions and (iii) flexible configuration by providing multiple alternative modes for binding the aspect library to an application.

We have gained wide experience in applying these patterns by using them to implement reusable aspects for a set of common recurring concerns and by applying these reusable aspects to multiple real-world applications like a web-shop and an online game platform. We have used a sub-set of this experience for a more quantitative study that provides a metric-based evaluation of the patterns and the pattern sequence. The results of this study confirmed the design knowledge we built from experience of using the patterns. These results provide an interesting set of hypotheses to test in an extensive empirical study.

6.2 Concluding thoughts

In this thesis we have used the notion of patterns to deal with the problems that limit the mainstream adoption of AOP. The presented patterns were based on the concepts of pointcut and advice, and some simple mechanisms for static crosscutting (like introductions and inheritance declarations). These are the key concepts of AspectJ and we believe they will remain the key concepts of any future mainstream AO technology. Our pattern system therefore presents a general solution for dealing with crosscutting concerns regardless of the

underlying composition style. For instance, current research gives considerable attention to making interesting events in the program explicit and available as extension points (EJP [57], Ptolemy [101], IIIA [116], EScala [41]). In our opinion our system of patterns still is relevant and valuable in such an evolution as, e.g., mediation of aspects and mapping of expected and provided join point abstractions remain pertinent design problems.

On the large scale, the presented patterns and idioms introduce techniques from component-based software composition. For instance, aspects and base will interact via interfaces that include required and provided join point abstractions. In such a setting, aspects take the role of crosscutting connectors [14]. With the use of the patterns, these connectors become more like components, having their own interfaces that need to be connected separately. As a result, they become reusable entities that encapsulate knowledge about the join point abstractions from a particular problem domain. For instance, in the domain of security, such a reusable connector would define the interesting security events and how these events relate to one another.

6.3 Future work

Some of the design patterns we describe in this thesis are well known and used on a large scale. For others we have built our own experience and evaluated them in a real-world context. Still, an extensive empirical study involving large-scale and complex case studies in a realistic development context will provide significant added value. The experience and evaluation in this thesis is a more than adequate starting point for such a study.

Our architectural pattern and the categories with their associated abstract design provide general solutions for the reusable modularization of crosscutting concerns. The idioms we present are based on the constructs available in AspectJ. It is interesting to explore other language mechanisms, both AO and OO, to describe other programming idioms and potentially new or updated idiom categories. For instance, mixin composition to compose different instances of pointcuts and advice; more flexible inheritance to incrementally define pointcuts and advice and an advanced type system to define more semantic pointcuts (e.g. using structural types).

Two of our categories, namely Aspect awareness and mediation have explored a design space with more potential. Our catalog provides the elementary techniques to realize basic mediation. Subsequently, the pattern sequence provides further detail on how these techniques can be combined and presents the first step towards a general strategy for more complex mediation. More

study on how to prepare aspects to cooperate with other aspects could be part of a more general approach of aspect composition. For instance, preparing aspects to cooperate with other aspects is related to preparing base code to be composed with aspects (Aspect awareness). Can these two categories learn from one another or are they two instances of a more general composition pattern? Fleshing out this relationship promises to be interesting and valuable for the field of AOSD.

By adding more knowledge on the relations between the patterns and idioms, the definition of a real pattern language is a logical next step. Additionally, other qualities can be taken into account requiring the need of other idiom categories and pattern sequences. New patterns and idioms will become relevant and other forces will drive pattern selection.

Appendix A

Description of change requests

In this appendix we describe the change requests for the MMS and COF applications in more detail.

A.1 Change requests MMS

1. CR0 (base): default integration of pricing; easy calculation of monthly license fee for each user.
2. CR1: it should be possible to get an overview of all the separate price factors that contributed to the final price.
3. CR2: seasonal discounts on the monthly license fee.
4. CR3: premium business customers receive a 10 Euro discount on their monthly fee.
5. CR4: MMS goes international: taxes should depend on geographical location of the customer.
6. CR5: every 5th MMS application a customer has gets a reduced license fee.
7. CR6: discounts for premium business customers (CR3) cannot be combined with seasonal reductions (CR2).

A.2 Change requests COF

1. CR0 (base): in COF, the pricing library is used to assign a travel limit to units (e.g. soldiers, ships) when they move around the map. Movements through enemy territory are more expensive than others.

2. CR1: it should be possible to get an overview of all the separate price factors that contributed to the final price.

3. CR2: there is less load on the game server at night, therefore nightly movements are less expensive

4. CR3: paying customers (players) can perform more movements.

5. CR3: premium business customers receive a 10 Euro discount on their monthly fee.

6. CR4: units that travel together have a reduced joined cost of movement.

7. CR5: the benefits of a paying customer (CR3) are not valid when moving through enemy territory.

Bibliography

[1] Aksit, M., Rensink, A., and Staijen, T. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development* (New York, NY, USA, 2009), AOSD '09, ACM, pp. 39–50.

[2] Aldrich, J. Open modules: Modular reasoning about advice. In *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming* (2005), A. P. Black, Ed., vol. 3586 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 144–168.

[3] Alexander, C. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[4] Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. Adding trace matching with free variables to aspectj. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2005), OOPSLA '05, ACM, pp. 345–364.

[5] The aspectj project. `http://www.eclipse.org/aspectj`.

[6] Baniassad, E., and Clarke, S. Theme: An approach for aspect-oriented analysis and design, 2004.

[7] Bartsch, M., and Harrison, R. Towards an empirical validation of aspect-oriented coupling measures. In *Proc. Workshop Assessment of Aspect Techniques* (2007).

[8] Bodden, E. An aspectj library for fault tolerance. `http://www.bodden.de/tools/#aspectj-ft`.

[9] Bodkin, R., Schidmeier, A., and Bodden, E. The ajlib incubator project. `http://fisheye.codehaus.org/browse/ajlib-incubator`.

[10] Borba, P., and Chiba, S., Eds. *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD 2011, Porto de Galinhas, Brazil, March 21-25, 2011* (2011), ACM.

[11] Buschmann, F., Henney, K., and Schmidt, D. *Pattern Oriented Software Architecture: On Patterns and Pattern Languages (Wiley Software Patterns Series).* John Wiley & Sons, 2007.

[12] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley, Chichester, UK, 1996.

[13] Bynens, M., De Win, B., Joosen, W., and Theeten, B. Ontology-based discovery of data-driven services. In *Second IEEE International Symposium on Service-Oriented System Engineering* (2006), pp. 175–178.

[14] Bynens, M., and Joosen, W. On the benefits of using aspect technology in component-oriented architectures. In *Proceedings of the 11th International Workshop on Component-Oriented Programming* (2006), pp. 1–4.

[15] Bynens, M., and Joosen, W. Towards a pattern language for aspect-based design. In *PLATE '09: Proceedings of the 1st workshop on Linking aspect technology and evolution* (2009), ACM, pp. 13–15.

[16] Bynens, M., Joosen, W., Moors, A., and Truyen, E. On the feasability of feature based composition for programming in the large. In *Practical problems of programming in the large* (2005).

[17] Bynens, M., Lagaisse, B., Joosen, W., and Truyen, E. The elementary pointcut pattern. In *BPAOSD '07: Proceedings of the 2nd workshop on Best practices in applying aspect-oriented software development* (New York, NY, USA, 2007), ACM, p. 2.

[18] Bynens, M., Truyen, E., and Joosen, W. A sequence of patterns for reusable aspect libraries with easy configuration. In *Software Composition* (2011). Accepted.

[19] Bynens, M., Truyen, E., and Joosen, W. A system of patterns for reusable aspect libraries. *Transactions on Aspect-Oriented Software Development VIII* (2011). Accepted (http://distrinet.cs.kuleuven.be/software/aodesignpatterns).

[20] Bynens, M., Van Landuyt, D., Truyen, E., and Joosen, W. Towards reusable aspects: the callback mismatch problem. In *Proceedings of the 9th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '10)* (2010), B. Adams, M. Haupt, and D. Lohmann, Eds., no. 33 in Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam, Universitätsverlag Potsdam, pp. 17–20.

[21] Canal, J. A. Parametric aspects: A proposal. In *RAM-SE* (2004), pp. 91–99.

[22] Chakravarthy, V., Regehr, J., and Eide, E. Edicts: implementing features with flexible binding times. In *AOSD* (2008), pp. 108–119.

[23] Chitchyan, R., Greenwood, P., Sampaio, A., Rashid, A., Garcia, A. F., and da Silva, L. F. Semantic vs. syntactic compositions in aspect-oriented requirements engineering: an empirical study. In *AOSD* (2009), K. J. Sullivan, Ed., ACM, pp. 149–160.

[24] Cho, E. S., Kim, M. S., and Kim, S. D. Component metrics to measure component quality. In *APSEC* (2001), IEEE Computer Society, pp. 419–426.

[25] Cibrán, M. A., and D'Hondt, M. A slice of mde with aop: Transforming high-level business rules to aspects. In *MoDELS* (2006), pp. 170–184.

[26] Clifton, C., Leavens, G. T., and Noble, J. Mao: Ownership and effects for more effective reasoning about aspects. In *ECOOP* (2007), E. Ernst, Ed., vol. 4609 of *Lecture Notes in Computer Science*, Springer, pp. 451–475.

[27] Cunha, C. A., Sobral, J. L., and Monteiro, M. P. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *AOSD* (New York, NY, USA, 2006), ACM, pp. 134–145.

[28] De Fraine, B., Quiroga, P. D., and Jonckers, V. Management of aspect interactions using statically-verified control-flow relations. In *International Workshop on Aspects, Dependencies and Interactions* (2008), pp. 5–14.

[29] D'Hondt, M., and Jonckers, V. Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In *AOSD* (2004), pp. 132–140.

[30] Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N., and Aho, A. V. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering 34*, 4 (2008), 497–515.

[31] Easychair conference system. `http://www.easychair.org`.

[32] Eclipse jdt weaving service. `http://wiki.eclipse.org/JDT_weaving_features`.

[33] Filman, R., and Friedman, D. Aspect-oriented programming is quantification and obliviousness, 2000.

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

[35] Fraine, B. D., Ernst, E., and Südholt, M. Essential aop: The a calculus. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings* (2010), ECOOP, Springer, pp. 101–125.

[36] Frakes, W. B., and Terry, C. Software reuse: Metrics and models. *ACM Comput. Surv. 28*, 2 (1996), 415–435.

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

[38] Garcia, A., Kulesza, U., Sardinha, J., Lucena, C., and Milidiú, R. The mobility aspect pattern. In *Proceedings of the 4th Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP'04* (2004).

[39] Garcia, A. F., Sant'Anna, C., Figueiredo, E., Kulesza, U., de Lucena, C. J. P., and von Staa, A. Modularizing design patterns with aspects: a quantitative study. In *AOSD* (2005), pp. 3–14.

[40] Garlan, D., Allen, R., and Ockerbloom, J. Architectural mismatch or why it's hard to build systems out of existing parts. *Software Engineering, International Conference on 0* (1995), 179.

[41] Gasiunas, V., Satabin, L., Mezini, M., Núñez, A., and Noyé, J. Escala: modular event-driven object interactions in scala. In *Proceedings of the tenth international conference on Aspect-oriented software development* (New York, NY, USA, 2011), AOSD '11, ACM, pp. 227–240.

[42] Ghezzi, C., Jazayeri, M., and Mandrioli, D. *Fundamentals of Software Engineering.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[43] Greenwood, P., Bartolomei, T. T., Figueiredo, E., Dósea, M., Garcia, A. F., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U., and Rashid, A. On the impact of aspectual decompositions on design stability: An empirical study. In *ECOOP* (2007), pp. 176–200.

[44] Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., and Rajan, H. Modular software design with crosscutting interfaces. *IEEE Softw. 23*, 1 (2006), 51–60.

[45] Group, T. H. Home of design patterns library and host of the plop conferences. `http://hillside.net`.

[46] Gudmundson, S., and Kiczales, G. Addressing practical software development issues in aspectj with a pointcut interface. In *In Advanced Separation of Concerns* (2001).

[47] Hanenberg, S., and Costanza, P. Connecting Aspects in AspectJ: Strategies vs. Patterns. First Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD, 2002.

[48] Hanenberg, S., and Schmidmeier, A. Idioms for building software frameworks in aspectj. AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, 2003.

[49] Hanenberg, S., and Unland, R. Using and Reusing Aspects in AspectJ. Workshop on Advanced Separation of Concerns, OOPSLA, 2001.

[50] Hanenberg, S., and Unland, R. Parametric introductions. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development* (New York, NY, USA, 2003), ACM, pp. 80–89.

[51] Hanenberg, S., Unland, R., and Schmidmeier, A. AspectJ Idioms for Aspect-Oriented Software Construction. In *Proceedings of EuroPLoP* (2003), pp. 617–644.

[52] Hannemann, J., and Kiczales, G. Design pattern implementation in java and aspectj. In *OOPSLA* (2002), pp. 161–173.

[53] Harbulot, B., and Gurd, J. R. A join point for loops in aspectj. In *Proceedings of the 5th international conference on Aspect-oriented*

software development (New York, NY, USA, 2006), AOSD '06, ACM, pp. 63–74.

[54] Harrison, W., and Ossher, H. Subject-oriented programming: a critique of pure objects. *SIGPLAN Not. 28*, 10 (1993), 411–428.

[55] Havinga, W., Nagy, I., and Bergmans, L. Introduction and derivation of annotations in aop: Applying expressive pointcut languages to introductions. In *First European Interactive Workshop on Aspects in Software* (2005).

[56] Hoffman, K., and Eugster, P. Bridging java and aspectj through explicit join points. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java* (New York, NY, USA, 2007), ACM, pp. 63–72.

[57] Hoffman, K., and Eugster, P. Towards reusable components with aspects: an empirical study on modularity and obliviousness. In *ICSE* (New York, NY, USA, 2008), ACM, pp. 91–100.

[58] Jadasite. `http://www.jadasite.com`.

[59] Jboss aop. `http://www.jboss.org/jbossaop`.

[60] Jhotdraw as open-source project. `http://jhotdraw.org/`.

[61] JUnit. Resources for test driven development. `http://www.junit.org/`.

[62] Kastner, C., Apel, S., and Batory, D. A case study implementing features using aspectj. *Software Product Line Conference, International 0* (2007), 223–232.

[63] Kellens, A., Mens, K., Brichau, J., and Gybels, K. Managing the evolution of aspect-oriented software with model-based pointcuts. In *ECOOP* (2006), Springer, pp. 501–525.

[64] Kerievsky, J. *Refactoring to Patterns.* Addison-Wesley, 2004.

[65] Khatchadourian, R., Greenwood, P., Rashid, A., and Xu, G. Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software. In *International Conference on Automated Software Engineering* (2009), IEEE/ACM, pp. 575–579.

[66] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming* (London, UK, 2001), Springer-Verlag, pp. 327–353.

[67] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds., vol. 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 1997, pp. 220–242.

[68] Kiczales, G., and Mezini, M. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ICSE '05, ACM, pp. 49–58.

[69] Kiczales, G., and Mezini, M. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ACM, pp. 49–58.

[70] Kiczales, G., and Mezini, M. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP* (2005), pp. 195–213.

[71] Kienzle, J. AspectOPTIMA. `http://www.cs.mcgill.ca/~joerg/AspectOPTIMA/AspectOPTIMA.html`.

[72] Kienzle, J., Duala-ekoko, E., and Gélineau, S. AspectOPTIMA: A case study on aspect dependencies and interactions. *Transactions on Aspect-Oriented Software Development 5* (2009), 187–234.

[73] Kienzle, J., and Gélineau, S. Ao challenge - implementing the acid properties for transactional objects. In *AOSD* (New York, NY, USA, 2006), ACM, pp. 202–213.

[74] Kniesel, G. Detection and resolution of weaving interactions. In *Transactions on Aspect-Oriented Software Development V*, A. Rashid and H. Ossher, Eds., vol. 5490 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 135–186.

[75] Kniesel, G., Rho, T., and Hanenberg, S. Evolvable pattern implementations need generic aspects. In *RAM-SE* (2004), pp. 111–126.

[76] Krueger, C. W. Software reuse. *ACM Comput. Surv. 24* (June 1992), 131–183.

[77] Kuhlemann, M., and Kästner, C. Reducing the complexity of AspectJ mechanisms for recurring extensions. In *Proc. GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE)* (2007).

[78] Kulesza, U., Alves, V., Garcia, A. F., de Lucena, C. J. P., and Borba, P. Improving extensibility of object-oriented frameworks with aspect-oriented programming. In *ICSR* (2006), M. Morisio, Ed., vol. 4039 of *Lecture Notes in Computer Science*, Springer, pp. 231–245.

[79] Laddad, R. AOP and Metadata: A perfect match, part 1. IBM developerworks: AOPWork, http://www-128.ibm.com/developerworks/java/library/j-aopwork3/.

[80] Laddad, R. AOP and Metadata: A perfect match, part 2. IBM developerworks: AOPWork, http://www-128.ibm.com/developerworks/java/library/j-aopwork4/.

[81] Laddad, R. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[82] Lagaisse, B., and Joosen, W. Decomposition into elementary pointcuts: A design principle for improved aspect reusability. In *SPLAT* (2006).

[83] Lagaisse, B., Joosen, W., and Win, B. D. Managing semantic interference with aspect integration contracts. In *International Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT)* (2004).

[84] Li, W., and Henry, S. M. Object-oriented metrics that predict maintainability. *Journal of Systems and Software 23*, 2 (1993), 111–122.

[85] Loughran, N., and Rashid, A. Framed aspects: Supporting variability and configurability for aop. In *ICSR* (2004), pp. 127–140.

[86] Marin, M. Ajhotdraw. http://swerl.tudelft.nl/bin/view/AMR/AJHotDraw.

[87] Marot, A., and Wuyts, R. Composing aspects with aspects. In *AOSD* (New York, NY, USA, 2010), ACM, pp. 157–168.

[88] Meszaros, G., and Doble, J. Metapatterns: A pattern language for pattern writing. In *In 3rd Pattern Languages of Programming conference* (1996).

[89] Microsystems, S. Enterprise javabeans technology. http://java.sun.com/products/ejb/index.jsp.

[90] Miles, R. *AspectJ Cookbook*. O'Reilly Media, Inc., 2004.

[91] Monteiro, M. P., and Fernandes, J. M. Towards a catalog of aspect-oriented refactorings. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development* (New York, NY, USA, 2005), ACM, pp. 111–122.

[92] Monteiro, M. P., and Fernandes, J. M. Towards a catalogue of refactorings and code smells for aspectj. *T. Aspect-Oriented Software Development 1* (2006), 214–258.

[93] Murphy, G. C., Lai, A., Walker, R. J., and Robillard, M. P. Separating features in source code: An exploratory study. In *ICSE* (2001), pp. 275–284.

[94] Noble, J., Schmidmeier, A., Pearce, D. J., and Black, A. P. Patterns of aspect-oriented design. In *In Proceedings of European Conference on Pattern Languages of Programs* (2007).

[95] Op de beeck, S., Truyen, E., Boucké, N., Sanen, F., Bynens, M., and Joosen, W. A study of aspect-oriented design approaches. CW Reports CW435, Department of Computer Science, K.U.Leuven, Leuven, Belgium, February 2006.

[96] Op de beeck, S., Van Landuyt, D., Truyen, E., and Joosen, W. A domain-specific middleware layer using AOSD: next-generation digital news publishing. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion,* (2008), ACM, pp. 78–81.

[97] Ossher, H., Kaplan, M., Harrison, W. H., Katz, A., and Kruskal, V. J. Subject-oriented composition rules. In *OOPSLA* (1995), pp. 235–250.

[98] Ossher, H., and Tarr, P. Hyper/j: multi-dimensional separation of concerns for java. In *Proceedings of the 23rd International Conference on Software Engineering* (Washington, DC, USA, 2001), ICSE '01, IEEE Computer Society, pp. 821–822.

[99] Ostermann, K., Mezini, M., and Bockisch, C. Expressive pointcuts for increased modularity. In *ECOOP* (2005), A. P. Black, Ed., vol. 3586 of *Lecture Notes in Computer Science*, Springer, pp. 214–240.

[100] Pawlitzki, T., and Steimann, F. Implicit invocation of traits. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010* (2010), ACM, pp. 2085–2089.

[101] Rajan, H., and Leavens, G. T. Ptolemy: A language with quantified, typed events. In *ECOOP* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 155–179.

[102] Rajan, H., and Sullivan, K. Aspect language features for concern coverage profiling. In *Proceedings of the 4th international conference on Aspect-oriented software development* (New York, NY, USA, 2005), AOSD '05, ACM, pp. 181–191.

[103] Rashid, A., and Chitchyan, R. Persistence as an aspect. In *AOSD* (New York, NY, USA, 2003), ACM, pp. 120–129.

[104] Rashid, A., Cottenier, T., Greenwood, P., Chitchyan, R., Meunier, R., Coelho, R., Südholt, M., and Joosen, W. Aspect-oriented software development in practice: Tales from aosd-europe. *Computer 43*, 2 (2010), 19–26.

[105] Rho, T., Kniesel, G., and Appeltauer, M. Fine-grained generic aspects. In *Foundations of Aspect-Oriented Languages* (2006).

[106] Sanen, F., Truyen, E., Joosen, W., Jackson, A., Nedos, A., Clarke, S., Loughran, N., and Rashid, A. Classifying and documenting aspect interactions. In *Proceedings of the Fifth AOSD Workshop on Aspect, Components, and Patterns for Infrastructure Software* (2006), pp. 23–26.

[107] Sant'anna, C., Garcia, A., Chavez, C., Lucena, C., and v. von Staa, A. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings XVII Brazilian Symposium on Software Engineering* (2003).

[108] Santos, A. L., and Koskimies, K. Modular hot spots: A pattern language for developing high-level framework reuse interfaces. In *In Proceedings of European Conference on Pattern Languages of Programs* (2008).

[109] Schmidmeier, A. Cooperating aspects. In *Proceedings of EuroPLoP* (2005).

[110] Schmidmeier, A. Aspect team. In *Proceedings of the 2nd workshop on Best practices in applying aspect-oriented software development* (New York, NY, USA, 2007), BPAOSD '07, ACM.

[111] Spring. `http://www.springsource.com/developer/spring`.

[112] Spring reference documentation. `http://www.springsource.org/documentation`.

[113] SpringSource. Spring insight. `http://www.springsource.com/products/tcserver/devedition`.

[114] SpringSource. Spring roo. "http://www.springsource.org/roo".

[115] Steimann, F. The paradoxical success of aspect-oriented programming. *SIGPLAN Not. 41*, 10 (2006), 481–497.

[116] Steimann, F., Pawlitzki, T., Apel, S., and Kästner, C. Types and modularity for implicit invocation with implicit announcement. *ACM Trans. Softw. Eng. Methodol. 20* (July 2010), 1:1–1:43.

[117] Stoerzer, M., and Graf, J. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 653–656.

[118] Sullivan, K. J., Griswold, W. G., Rajan, H., Song, Y., Cai, Y., Shonle, M., and Tewari, N. Modular aspect-oriented design with xpis. *ACM Trans. Softw. Eng. Methodol. 20*, 2 (2010).

[119] Tarr, P. L., Ossher, H., Harrison, W. H., and Jr., S. M. S. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE* (Los Alamitos, CA, USA, 1999), IEEE Computer Society, pp. 107–119.

[120] the AspectJ Team. The aspectj 5 development kit developer's notebook. http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html.

[121] Tourwe, T., Brichau, J., and Gybels, K. On the Existence of the AOSD-Evolution Paradox. SPLAT Workshop, Boston, AOSD, 2003.

[122] Van Landuyt, D., Op de beeck, S., Kemper, B., Truyen, E., and Joosen, W. Building a next-generation digital publishing platform using aosd. http://distrinet.cs.kuleuven.be/projects/digitalpublishing.

[123] Van Landuyt, D., Op de beeck, S., Truyen, E., and Joosen, W. Domain-driven discovery of stable abstractions for pointcut interfaces. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development* (New York, NY, USA, 2009), AOSD '09, ACM, pp. 75–86.

[124] Van Landuyt, D., Truyen, E., and Joosen, W. Discovery of stable domain abstractions for reusable pointcut interfaces: common case study for ao modeling. CW Reports CW560, Department of Computer Science, K.U.Leuven, Leuven, Belgium, Aug. 2009.

[125] Van Landuyt, D., Truyen, E., and Joosen, W. Discovery of stable abstractions for aspect-oriented composition in the car crash management domain. In *LNCS Transactions on Aspect-Oriented Software Development* (2010).

[126] Verhanneman, T., Piessens, F., Win, B. D., Truyen, E., and Joosen, W. Implementing a modular access control service to support application-specific policies in caesarj. In *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development* (New York, NY, USA, 2005), ACM.

[127] Walker, R. J., and Viggers, K. Implementing protocols via declarative event patterns. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering* (New York, NY, USA, 2004), SIGSOFT '04/FSE-12, ACM, pp. 159–169.

[128] Wampler, D. Noninvasiveness and Aspect-Oriented Design: Lessons from Object-Oriented Design Principles, 2004.

[129] Wampler, D. The Challenges of Writing Reusable and Portable Aspects in AspectJ: Lessons from Contract4J. In *International Conference on Aspect Oriented Software Development (AOSD 2006)–Industry Track Proceedings* (2006).

[130] Wiese, D., Meunier, R., and Hohenstein, U. How to convince industry of aop. AOSD, Industry Track, 2007.

[131] Zschaler, S., and Rashid, A. Aspect assumptions: a retrospective study of aspectj developers' assumptions about aspect usage. In *AOSD* (New York, NY, USA, 2011), ACM, pp. 93–104.

List of publications

- Bynens, M., Truyen, E., and Joosen, W. A sequence of patterns for reusable aspect libraries with easy configuration. In *Software Composition* (2011).[18]

- Bynens, M., Truyen, E., and Joosen, W. A system of patterns for reusable aspect libraries. *Transactions on Aspect-Oriented Software Development VIII* (2011).[19]

- Bynens, M., Van Landuyt, D., Truyen, E., and Joosen, W. Towards reusable aspects: the callback mismatch problem. In *Proceedings of the 9th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '10)* (2010), B. Adams, M. Haupt, and D. Lohmann, Eds., no. 33 in Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam, Universitätsverlag Potsdam, pp. 17–20.[20]

- Bynens, M., and Joosen, W. Towards a pattern language for aspect-based design. In *PLATE '09: Proceedings of the 1st workshop on Linking aspect technology and evolution* (2009), ACM, pp. 13–15.[15]

- Bynens, M., Lagaisse, B., Joosen, W., and Truyen, E. The elementary pointcut pattern. In *BPAOSD '07: Proceedings of the 2nd workshop on Best practices in applying aspect-oriented software development* (New York, NY, USA, 2007), ACM, p. 2.[17]

- Bynens, M., and Joosen, W. On the benefits of using aspect technology in component-oriented architectures. In *Proceedings of the 11th International Workshop on Component-Oriented Programming* (2006), pp. 1–4.[14]

- Op de beeck, S., Truyen, E., Boucké, N., Sanen, F., Bynens, M., and Joosen, W. A study of aspect-oriented design approaches. CW

Reports CW435, Department of Computer Science, K.U.Leuven, Leuven, Belgium, February 2006. [95]

- Bynens, M., De Win, B., Joosen, W., and Theeten, B. Ontology-based discovery of data-driven services. In *Second IEEE International Symposium on Service-Oriented System Engineering* (2006), pp. 175–178. [13]

- Bynens, M., Joosen, W., Moors, A., and Truyen, E. On the feasability of feature based composition for programming in the large. In *Practical problems of programming in the large* (2005).[16]

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science
DistriNet
Kasteelpark Arenberg 1 bus 2200
B-3001 Heverlee