Role-Based Refactoring of Crosscutting Concerns

by

Jan Hannemann

M.Sc., Georgia Institute of Technology, USA, 1999
M.Sc. (Diplom), Universität Osnabrück, Germany, 1999

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

November 2005

© Jan Hannemann 2005

Abstract

Improving the structure of code can help developers work with a software system more efficiently and more consistently. Aspect-oriented programming (AOP) offers additional ways to structure software by providing explicit means to modularize crosscutting concerns (CCCs) in modularity units called aspects. With the advent of AOP, a new kind of structural improvement of software needs to be considered: the refactoring of non-modularized CCCs into aspects. Refactorings have shown to be helpful for object-oriented software development and maintenance, but their application to aspect-oriented software is not yet well understood. In particular, since refactorings of non-modularized crosscutting concerns involve multiple program elements with potentially complicated relationships, they are considerably more complex than traditional refactorings; the lack of tool support to help plan, reason about and execute CCC refactorings impedes the improvement of code modularity.

The thesis of this research is that the refactoring of crosscutting concerns can be supported by a role-based concern model. In this model, crosscutting concerns are described in terms of abstract roles, and instructions for refactoring the concerns are written in terms of those roles. To apply a refactoring, a developer maps a subset of the roles to concrete program elements; a tool can then help complete the mapping of roles to the existing program. Refactoring instructions are then applied to manipulate and modularize the concrete elements corresponding to the crosscutting concern. The abstract nature of such a role-based concern model allows the definition of a refactoring description separately from concrete systems it may be applied to, and allows using a single description to refactor multiple instances of the same concern.

To aid developers in restructuring the implementation of crosscutting concerns using aspect-oriented programming, we introduce in this dissertation a refactoring approach and proof-of-concept tool founded on our role-based concern model. We show that abstract descriptions of crosscutting concerns can be applied to previously existing software and we describe the potential for expressing and executing a variety of new CCC refactorings.

Contents

Abstract	ii
Contents	iii
List of Tables	vi
List of Figures	vii
List of Abbreviations	ix
Acknowledgements	x
1 Introduction	1
1.1 Refactoring	4
1.1.1 Traditional Refactorings	4
1.1.2 Aspect-Oriented Refactorings	4
1.2 Role-Based Refactoring	8
1.3 Overview of the Dissertation	12
2 Design Pattern Implementations in Java and AspectJ	14
2.1 Established Challenges	15
2.2 Study Format	16
2.3 Results	16
2.3.1 Example: The Observer Pattern	16
2.3.2 Other GoF Patterns	23
2.4 Analysis	27
2.4.1 General Improvements	27
2.4.2 Specific Improvements	29
2.4.3 Crosscutting Structure of Design Patterns	31
2.5 Summary	33
3 Role-Based Refactoring	35
3.1 Workflow	37
3.2 Describing Crosscutting Concerns	37
3.3 Describing Refactorings	39
3.4 Mapping Role Elements	39
3.5 Impact Analysis	40
3.6 Executing Refactorings	41

3.7	Example	41
3.8	Summary	42
4	Tool Support for Role-Based Refactoring	44
4.1	The Refactoring Engine	44
4.1.1	Representing Crosscutting Concerns	46
4.1.2	Refactoring Primitives	47
4.1.3	Interference of Refactoring Primitives	49
4.1.4	Implementation of CCC Refactoring Instructions	49
4.1.5	Representing Aspect-Oriented Program Constructs	52
4.2	Auto-Mapping Support	52
4.3	Support for Impact Analyses	53
4.4	Example	55
4.4.1	Mapping	55
4.4.2	Impact Analysis	56
4.4.3	Executing the Changes	58
4.4.4	Remarks	58
4.5	Tool Calibration	59
4.6	Extending the Framework	59
4.7	Summary	62
5	Evaluation	63
5.1	Methodology	63
5.2	Study 1: Design Pattern Implementations in JHotDraw	63
5.2.1	Results	64
5.2.2	Other Patterns in JHotDraw	71
5.3	Study 2: Other Crosscutting Concerns	73
5.3.1	Logging	73
5.3.2	Laddad's AOP Refactoring Examples	74
5.4	Summary	79
6	Discussion	81
6.1	Benefits	81
6.2	Limitations	81
6.2.1	Approach	82
6.2.2	Supporting Technology	84
6.2.3	Tool	86
6.3	Future Work	88
6.3.1	Refactorings for other Crosscutting Concerns	88
6.3.2	Improvements of the Refactoring Tool	89
6.3.3	Automatic Generation of Refactoring Instructions	90
6.3.4	Exploratory Generation of Role-Based Refactorings	90
6.3.5	Extensions of the Role-Based Concern Model	91

7 Related Work	92
7.1 Refactoring	92
7.1.1 Composite Refactorings	92
7.1.2 Analyses for Ensuring the Behavior-Preservation of Object- Oriented Refactorings	93
7.1.3 Aspect-Oriented Refactorings	94
7.2 Concern Modeling	96
7.3 Aspects and Roles	97
7.4 Aspects and Design Patterns	97
8 Conclusions	99
A Aspect-Oriented Terminology	101
B Refactoring Primitives	102
B.1 Aspect Creation	102
B.2 Other Creational Primitives	102
B.3 Transformational Primitives	103
B.4 Destructive Refactorings	104
Bibliography	105

List of Tables

2.1	Design Patterns, desirable properties of their AspectJ implementations, and usage of roles	28
3.1	Role Elements and their Relationships	38
4.1	Conceptual refactoring steps and their implementation (OBSERVER) .	51
4.2	Sample weights for name fragments for role elements	54
5.1	Role mappings for the DrawingView — FigureSelectionListener instance of the OBSERVER pattern	65
5.2	Role mappings for the Drawing — DrawingChangeListener instance of the OBSERVER pattern	67
5.3	Role mappings for the Clipboard instance of the SINGLETON pattern .	68
5.4	Role mappings for the Iconkit instance of the SINGLETON pattern . .	68
5.5	Role mappings for the CompositeFigure instance of the COMPOSITE pattern	70
5.6	Role mappings for the AbstractFigure instance of the TEMPLATE METHOD pattern	71
5.7	Role mappings for the AbstractLineDecoration instance of the TEMPLATE METHOD pattern	72

List of Figures

1.1	Tangling and scattering of a concern	2
1.2	Refactoring of a non-modularized CCC into an aspect (schematic) . .	3
1.3	Source code fragment demonstrating logging of deductions in a bank- ing system.	5
1.4	The role-based refactoring approach (schematic)	9
1.5	The Chain of Responsibility design pattern structure according to [28], including methods for setting up the chain	10
1.6	The object structure for the sample system	10
1.7	Principal role elements comprising a single crosscutting concern . . .	11
2.1	A simple graphical figure element system that uses the OBSERVER pat- tern in Java. The assignment of <i>Subject</i> and <i>Observer</i> roles are indi- cated using UML stereotypes [73]. Underlined methods contain code necessary to implement this instance of the pattern.	17
2.2	The structure of an instance of the OBSERVER pattern in AspectJ. <i>Sub- ject</i> and <i>Observer</i> roles crosscut classes, and the changes of interest (the <code>subjectChange(...)</code> pointcut) crosscuts methods in various classes	22
2.3	Dependencies and code distribution in typical instances of the MEDI- ATOR pattern. The AspectJ implementation removes cyclic dependen- cies and localizes the pattern code	31
2.4	Crosscutting caused by pattern composition. A second pattern instance introduces additional crosscutting	32
3.1	The partial structure of an object-oriented instance of the OBSERVER pattern in the JHotDraw framework. The mapping of role elements to program elements is indicated using UML stereotypes [73].	36
3.2	Role elements in the OBSERVER CCC. Role types are represented as squares, role methods as rounded squares and role fields as hexagons. The structure reflects the <i>pull</i> variant of the pattern [28]	38
3.3	Based on an initial mapping (solid arrow), further mappings can be inferred (dashed arrows)	40
4.1	The CCC refactoring tool architecture. Existing software used is shown in grey	45
4.2	Concern structure description of the <i>Observer</i> CCC as used by the refactoring tool.	46

4.3	Pre-processing of buffer modifications	50
4.4	Structure of the OBSERVER CCC as presented by the refactoring tool.	55
4.5	Mapping the <i>Observer</i> role to the <i>Display</i> interface.	56
4.6	After mapping <i>Subject</i> to <i>Point</i> , the automapper determines candidates for mapping the <i>attach(Observer)</i> role method based on a structural and lexical analysis of methods of the <i>Point</i> class.	57
4.7	The completed mappings for the OBSERVER CCC instance in the figure package example.	57
4.8	An impact analysis detects that an existing aspect is advising a changing program element.	58
4.9	Concern structure description of a <i>Logging</i> CCC.	60
4.10	Refactoring instructions for the <i>Logging</i> CCC.	61
5.1	Concern structure of the SINGLETON CCC	67
5.2	Concern structure of the EXTRACT CONCERN IMPLEMENTATION CCC, applied to an instance of the COMPOSITE pattern	69
5.3	Concern structure of the TEMPLATE METHOD CCC	70
5.4	Concern structure of the LOGGING CCC	73
5.5	Concern structure of the EXTRACT INTERFACE IMPLEMENTATION CCC	75
5.6	The Extract Method Calls Refactoring as shown in [51].	76
5.7	Concern structure of the EXTRACT METHOD CALLS CCC	76
5.8	Argument trickle between <i>caller()</i> and <i>callee()</i> , taken from [51]. An argument gets passed through the call stack but is only used within the <i>callee()</i> method.	78
5.9	Concern structure of the REPLACE ARGUMENT TRICKLE WITH WORM-HOLE CCC	78
6.1	The programming models of Eclipse and AJDT and their relationships (schematic)	86

List of Abbreviations

AO	Aspect-Oriented
AOP	Aspect-Oriented Programming
AST	Abstract Syntax Tree
CCC	Crosscutting Concern
GoF	Gang-of-Four, the authors of [28]
IA	Impact Analysis
OO	Object-Oriented
OOP	Object-Oriented Programming

Acknowledgements

I would like to recognize the people who contributed to this dissertation in various ways:

First and foremost, I would like to express my deepest gratitude to my parents Marga and Jürgen Hannemann, who were a constant source of encouragement and motivation. Without their support, this work would not have been possible.

My co-supervisors, Gail Murphy and Gregor Kiczales, have provided me with invaluable technical and moral support. Our many thought-provoking conversations we have shaped my thinking about research and the various aspects of academic life. I am very grateful for Gregor's introduction to and guidance with the exciting area of aspect-oriented research, and I am deeply indebted to Gail for her consistently outstanding advice and encouragement.

Our Software Practices Lab provided a stimulating environment to conduct this research, and I would like to extend my thanks to all its members for the exciting discussions, useful feedback, and fun times. I am grateful to have had the opportunity to work with such outstanding people. In particular, I would like to thank Mik Kersten for his input on AspectJ- and AJDT-related questions, and Christopher Dutchyn, who never tired of my requests for clarifications on obscure aspects of programming language theory. My gratitude and deep appreciation go to Brian de Alwis who not only helped with the intricacies of realizing this document in LaTeX, but whose friendship and wisdom (and our shared appreciation for good beer and fine cheeses) have helped to make my stay in Vancouver so enjoyable.

Thank you all.

—Jan

Chapter 1

Introduction

Good modularity improves the quality of software. It facilitates independent development of components [19], improves comprehensibility [22], eases verification [85], and facilitates extensibility and evolution [69]. The value of modularity is even quantifiable [5].

Developing software requires working with a number of *concerns*, or considerations a developer might have about the implementation of a software system. For a secure online shopping site, for example, typical concerns would include the authentication of customers, adding an item to the shopping cart, verifying credit card information, as well as many others.

In software with good modularity, each concern is implemented in only one module, and each module implements only a small number of concerns. This structure helps the developer manage complexity. To deal with any one concern they only have to look at one module, and to understand any one module they only have to think about a small number of concerns.

Unfortunately, some concerns inherently do not fit into the structures that existing modularity mechanisms support. When a concern does not fit, its implementation cuts across the system's modules. The implementation of such *crosscutting* concerns (CCCs) is scattered and tangled, which is illustrated schematically in Figure 1.1. This SeeSoft-like view [24] shows the code in terms of its modularity units (such as classes), presented as rectangles. The lines of code within these units pertaining to the implementation of a concern are shown as grey areas.

Tangling occurs when the code pertaining to multiple concerns (shown in different shades of grey) is interleaved within a single modularity unit. *Scattering* occurs when code related to a single concern appears in multiple modularity units. These phenomena often occur together: a scattered implementation is usually tangled with other concerns in the affected modules. The authentication concern mentioned above, for instance, would likely be tangled with the main business logic of the application, and scattered across multiple modularity units. Non-modularized crosscutting concerns can affect multiple modules and result in code that is hard to understand and maintain. Other examples of crosscutting concerns are data persistence [70], quality of service [80], and policy enforcement [50]. Many applications also have a number of specific crosscutting concerns, including implementations of design patterns such as OBSERVER and COMPOSITE [28], synchronization strategies and resource sharing.

Crosscutting concerns can be modularized using aspect-oriented programming¹ (AOP) [46]. AOP is a relatively new programming paradigm that offers additional

¹For an overview of aspect-oriented terminology, please refer to Appendix A

Figure 1.1: Tangling and scattering of a concern

ways to structure software by providing mechanisms to explicitly modularize CCCs in modularity units called *aspects*.

To transform a non-modularized CCC into a modular aspect, developers need to systematically restructure their software. This can happen, for example, when adopting AOP in an object-oriented legacy system, or when CCCs were not apparent during the design of an AO system and were not properly modularized.

Systematic restructuring of software to improve its internal structure while preserving its external behavior is called *refactoring* [26, 32, 33, 67]. Refactorings in the traditional sense are small program transformations that change a single program element, such as a method or a field, or a single localized set of these elements, such as consecutive statements or code blocks. The transformation applied to support the refactoring includes updating references to these elements, for example adjusting call sites to a renamed method. Examples of traditional refactorings are *change variable name* or *move member variable to superclass* [67]. Multiple of these low-level refactorings can be combined into *composite refactorings*, such as *convert code segment to function* [67].

Refactorings of crosscutting concerns differ from traditional refactorings in that they target multiple scattered and tangled program elements. Conceptually, a CCC refactoring replaces the scattered and tangled implementation with an equivalent aspect, as shown schematically in Figure 1.2.

In this dissertation, we describe and evaluate the first semi-automated refactoring approach to help programmers transform scattered implementations of crosscutting

Figure 1.2: Refactoring of a non-modularized CCC into an aspect (schematic)

concerns into modular implementations in an AOP language. Integral to the approach is an abstraction of the target crosscutting concern.

Thesis: *A role-based concern model enables the semi-automatic refactoring of crosscutting concerns.*

A *role* in this context abstracts part of the functionality represented by the concern. For example, a role-based representation of a synchronization concern could have roles for acquiring and releasing a lock. In a software system utilizing synchronization, one or more concrete program elements (here: methods) can play the role of the *acquire-Lock()* role method. Our approach is based on the observation that roles can describe crosscutting behavior [35, 43, 53, 54, 79] and that CCCs that have been modularized conceptually by roles are amenable to modular implementation using an AOP language [38].

In our approach, a CCC refactoring consists of a description of the roles within a CCC and refactoring instructions in terms of these (abstract) roles. To invoke the refactoring, a developer maps the roles to concrete program elements in the scattered implementation. The refactoring instructions are then applied to the mapped elements to transform the program, converting it to a modular AOP implementation. Our approach focuses on preservation of *intent* over preservation of behavior, in the sense that the transformation preserves the overall requirements of the system, but may change the semantics of the program locally. Such cases are identified by impact analyses of the proposed changes and accepted or rejected by the developer.

In section 1.1 we describe in more detail the context of this work, providing a brief overview of traditional and aspect-oriented refactoring. In section 1.2 we present our role-based refactoring approach. Section 1.3 provides an overview of the organization of this dissertation and outlines how its parts will address the thesis claim.

1.1 Refactoring

Any useful software system requires constant evolution and change [8]. Often those changes require that the software be re-modularized, so that the system becomes easier to understand, extend, or maintain [32, 33]. To address this ongoing need, researchers and developers have developed the practice of *refactoring*. As described in [32, 33, 67, 71], refactorings are parameterized transformations of a system's source code intended to improve a system's structure with regards to informally expressed goals, such as maintainability, changeability, readability, performance, or memory demands. Traditional refactorings are generally behaviour preserving [67]. Modern software development environments include built-in support for semi-automated refactorings [20, 91].

To distinguish aspect-oriented refactorings from traditional refactorings, and to differentiate between individual approaches to aspect-oriented refactoring, we will use the simple AspectJ [45, 89] example shown in Figure 1.3. It shows part of a class for a banking system that focuses on handling deductions from an account. In the method `deduct(int)`, race conditions are prevented by acquiring a lock before the actual deduction is executed, which happens in method `doIt(int)`. After that, the lock is released. If the balance is not sufficiently high, an exception is raised. Further, an aspect logs all successful deductions.

1.1.1 Traditional Refactorings

We use the term *traditional refactorings* to refer to parameterized, behavior-preserving transformations of mostly object-oriented systems². They generally target single program elements and can often be automated [33]. Despite the limited number of elements targeted, they may cause program-wide changes. For example, a *rename method* refactoring [26] requires all call sites to the targeted method to be updated. Renaming the `deduct(int)` method in the example system represents a traditional refactoring.

More complex traditional refactorings are composite refactorings (or “big refactorings” [26]), which consist of a series of low-level refactorings. Typical examples perform operations such as simplifying conditionals or creating an abstract superclass. The scope of such refactorings is usually focused on single program elements or small sets of non-scattered program elements and their associated references.

1.1.2 Aspect-Oriented Refactorings

Aspect-oriented refactorings differ from traditional refactorings in that they involve AOP constructs, either as the targeted elements or in the resulting code. They can be divided into three distinct types:

1. **Aspect-aware OO Refactorings**, which are extensions of traditional object-oriented refactorings, ensuring that they do not break AO programming constructs.

² Although refactorings are not limited to OO systems, the majority of research targets object-oriented refactorings.

```
public class Account {
    ...
    public void deduct(int amount)
        throws InsufficientFundsException {
        acquireLock(this);
        dolt(amount);
        releaseLock(this);
    }
    private void dolt(int amount)
        throws InsufficientFundsException {
        if (amount > balance) {
            throw new InsufficientFundsException (...);
        } else {
            balance = balance - amount;
            transactions++;
        }
    }
}

public aspect TransactionLogger {
    ...
    pointcut deduction(Account acct, int amount):
        call(void Account.dolt(int)) &&
        target(acct) &&
        args(amount);

    after (Account acct, int amount) returning:
        deduction(acct, amount) {
        log(amount + " deducted from " + acct.getId());
    }
}
```

Figure 1.3: Source code fragment demonstrating logging of deductions in a banking system.

2. **New Aspect-Oriented Refactorings**, which target AOP constructs.
3. **Refactorings of Crosscutting Concerns**, which are complex refactorings of non-modularized CCCs into aspects.

The focus of this work is on refactorings of CCCs. To distinguish these from the other refactorings, we will briefly describe all three kinds of aspect-oriented refactorings.

Aspect-aware Object-Oriented Refactorings

Known OO refactorings must be adjusted to account for new AOP constructs. For instance, applying a Rename Method refactoring [26] to the (poorly named) `doIt(int)` method requires updates to references of that construct not only in OO code elements, but also in AOP constructs, such as the `deduction(..)`³ pointcut. Applying a refactoring to inline the `doIt(int)` method is another example of an aspect-oriented version of an OO refactoring, as the join points [40, 57] identified by the pointcut cease to exist. Making OO refactorings aspect-aware is the focus of several research projects, for example [34, 42, 86].

New Aspect-Oriented Refactorings

The new aspect-oriented programming constructs allow for a number of new refactorings. Many of these parallel existing OO refactorings, for example renaming or inlining a pointcut and advice body. With respect to the kind of refactorings that can be applied to them, pointcuts can be compared to methods and aspects to classes. It is straightforward to envision the meaning of refactorings such as *Add Parameter*, *Pull Up Method*, or *Push Down Method* [26] to a pointcut declaration. Similarly, equivalents of *Collapse Hierarchy*, *Extract Super/Subclass*, or *Move Class* may be conceived for aspects [62].

The new programming constructs in AOP languages also allow for a set of new refactorings that have no OO equivalents, such as the merging or splitting of advice and/or pointcuts. For example, imagine an additional pair of pointcut and advice in the banking system for logging deposits of money to an account. The pointcuts, the advice, or both could be merged with the ones for logging deductions. Several researchers have proposed new refactorings for AOP constructs, for example [42, 61, 62].

Finally, there are low-level object-to-aspect program transformations that move small OO code fragments to aspects by replacing them with AO equivalents. For example, the *Replace Implements with Declare Parents* refactoring [62] moves an `implements` declaration to an aspect by replacing it with the equivalent AspectJ construct `declare parents`. Another example would be to replace the explicit call to `acquireLock(..)` at the beginning of `deduct(int)` (see Figure 1.3) with a corresponding pointcut and advice pair.

All of these types of AO-refactorings are similar in scope to traditional refactorings and do not address the refactoring of CCCs directly. They usually comprise low-level transformation steps in a CCC refactoring.

³Analogous to AspectJ notation, we use“(..)” after a pointcut or method name to indicate that the construct has zero or more arguments.

Refactorings of Crosscutting Concerns

Refactorings of CCCs transform scattered implementations of crosscutting concerns into a modularized form (generally an aspect), which is schematically shown in Figure 1.2. For instance, replacing *all* code pertaining to logging account transactions in the example banking system with an equivalent, modular AO implementation would constitute a CCC refactoring.

Non-modularized CCCs consist of multiple related program elements. These relationships can be either explicit through code-level dependencies, such as method calls, subtyping, or *contains* relationships, or implicit, such as a developer's decision to include them in the concern. We will refer to the set of all program elements that make up a concern's implementation as the concern *extent*. These elements and their relationships form the concern *structure*. Refactorings of CCCs are challenging for a number of reasons.

First, given that program elements comprising a CCC may not be directly related programmatically, it can be difficult to describe the concern extent based on these properties. For example, in the case of serialization or file I/O in general, the code elements implementing read operations need not be directly linked to those for write operations.

Second, since non-modularized crosscutting concerns have scattered and tangled implementations, separating concern code from non-concern code can be hard. Currently, this is done by hand (for informal refactoring descriptions) or through aspect mining approaches, which are limited in their own ways [15]. If the concern extent is not correctly identified, the resulting refactoring will be incomplete.

Third, for CCC refactorings there is sometimes a trade-off between behavior preservation and preservation of intent. Consider, for instance, inlining of the `doIt(int)` method in the banking system shown in Figure 1.3. This refactoring would require changes to the `deduction(...)` pointcut since the pointcut references that method. The intent of the pointcut is to capture all deductions. To preserve the original behavior, we could define the logging to happen before the next program statement is executed (i.e., before the call to `releaseLock(...)`). Alternatively, we could decide that the logging should happen at the last statement of the original, inlined method, where the `transactions` field gets incremented. Both approaches fail to capture the original intent of the pointcut and make it less readable and self-explanatory. To maintain readability of our pointcut, we could instead specify that logging should take place after the `balance` field is modified in method `deduct(int)`. This would require accepting a minor behavior variation, which, while being against the fundamental principle of refactoring, would keep the pointcut readable, and the intent of the pointcut intact.

Finally, choices in the application of the refactoring and the space of possible aspect-oriented implementations raise the need for user interaction as part of the refactoring process. The proposed inlining of the `doIt(int)` method can have at least three possible resolutions with respect to the `deduction(...)` pointcut as shown above. The decision to choose one alternative over another is hardly automatable. Although it is conceivable that a tool can just pick one possibility and provide reasoning for later review (e.g., using Java metadata annotations), a complex refactoring may require multiple choices, each of which can influence subsequent refactoring steps.

Current research in this area can be divided into three approaches: informal de-

scriptions of CCC refactorings, approaches that combine aspect mining and low-level AOP refactorings, and approaches that employ concern models.

- **Informal descriptions** — Laddad investigated a number of CCC refactorings and described the required transformation steps that extract scattered implementations into aspects [49]. The descriptions are similarly structured as the refactoring instructions according to Fowler [26], but the refactorings are currently not supported by a tool and must be applied manually.
- **Combined mining and refactoring approaches** — Binkley et al. have proposed an approach to refactoring CCCs in which they incorporate an *aspect-mining*⁴ technique to recognize the scattered and tangled code comprising a CCC; once recognized, the concern code is modularized using a series of low-level aspect-oriented refactorings that are suggested by their refactoring tool [9]. Currently their approach relies on a manual identification of concern code by a developer, and their approach does not yet produce aspect code. The approach focuses on the specific problem of extracting code fragments into aspects, such as the *Extract Beginning* refactoring that replaces a code block at the start of methods with appropriate *before* advice [9]. In comparison to the approach described in this thesis, Binkley's approach does not consider the structure of the concern being transformed, and the refactoring is a series of individual code extractions. As a result, the aspect code they propose to generate has little structure.
- **Approaches using concern models** — Our earlier work is the only approach to date that uses a model for the concern to be refactored [37]. We employ *concern graphs* [72] to model the concerns and help plan refactorings, and discuss the role of developer interaction in providing tool support for refactoring scattered code to aspects. However, concern graphs do not include any abstraction to differentiate between individual concrete program elements, thus limiting the description of complex refactorings.

Work on CCC refactorings, including this dissertation, focuses on transformations of object-oriented concern implementation into aspects. This is due to the fact that AOP offers additional modularity mechanisms on top of OO, thus allowing for the modularization of formerly non-modularizable crosscutting concerns. A typical application of such OO-to-AO CCC refactorings is the adaption of a legacy OO system to utilize aspects.

1.2 Role-Based Refactoring

To help developers conceptualize, plan and execute refactorings of crosscutting concerns into modular aspects, we have developed a role-based refactoring approach [39]. In role-based refactoring, a crosscutting concern is described as a set of interrelated *role*

⁴Aspect-mining encompasses semi-automated analyses of software to identify candidate code fragments for extraction to aspects [74]

Figure 1.4: The role-based refactoring approach (schematic)

elements that abstract the functionality of the individual program elements comprising the concern. CCC refactorings are defined in terms of these role elements, effectively separating the CCC refactoring description from concrete implementations.

Figure 1.4 depicts the approach. The objective is to refactor a scattered and tangled implementation of a CCC (A) into a modular aspect (B). First, the developer chooses a CCC refactoring description from the library of CCC refactorings. This abstract description consists of two parts: a role-based concern description (C) and a set of refactoring instructions (D) in terms of these roles. Applying these instructions to the elements of the concern results in a modular implementation (E). To apply the refactoring to a concrete system, the role elements are mapped onto corresponding program elements in the target system (F). The (abstract) refactoring instructions are applied to the mapped program elements (G), resulting in the desired modularized implementation (B).

We illustrate the approach with a simple example. Consider a software system with a graphical user interface, for example a drawing program, that provides context-sensitive help for its widgets: whenever the F1 key is pressed, the widget under the current mouse position is asked to provide information about itself.

In order to avoid having to specify this information for every single widget, the developer used a CHAIN OF RESPONSIBILITY [28] pattern, a typical object-oriented solution. In this approach, widgets are organized in a logical chain (i.e., each widget

Figure 1.5: The Chain of Responsibility design pattern structure according to [28], including methods for setting up the chain

Figure 1.6: The object structure for the sample system

has a successor), and widgets can either choose to either answer a request for context-sensitive help, or pass the request on to its successor in the chain. For example, a **Button** could pass the request on to its containing widget, a **ButtonBar**, which could in turn pass it on to its successor, which might be the main application class itself.

The CHAIN OF RESPONSIBILITY pattern structure is shown in Figure 1.5 (plus methods for specifying successors, which are used for setting up the call chain).

For the application mentioned above, the structure might look like Figure 1.6. The implementation of the concern “context-sensitive-help” is non-modularized and cross-cuts various modules in the system. In an aspect-oriented implementation [38], it is possible to encapsulate the CHAIN OF RESPONSIBILITY logic into an aspect, modularizing the crosscutting structure. If a modular implementation of this concern is desired, a CCC refactoring is required to transform the original implementation into an aspect.

The concept of role-based refactoring is based on the observations that (1) cross-cutting concerns have structure to them, and that (2) this structure can be leveraged in the refactoring process. For example, all methods playing the role of `handleRequest(..)` would require the same set of low-level AO-refactorings. When considering a scattered implementation of a non-modularized concern (see Figure 1.1 (b)), it is important to note that its code fragments can be categorized according to which part of the concern’s principal functionality they implement, or, which role they play within the concern. Figure 1.7 shows a sample implementation of the object-oriented Chain of Responsibility pattern. For this particular concern the role elements are a role field

Figure 1.7: Principal role elements comprising a single crosscutting concern

successor, two role methods *handleRequest(..)* and *setSuccessor(..)*, and the role types *Handler* (not shown) and *Request*. Different shades of grey identify code segments corresponding to the role elements comprising the concern.

We differentiate three different kinds of role elements: role fields, like *successor*, role methods, such as *handleRequest(Request)*, and role types, like *Request*.

Role-based refactoring assumes that all code fragments playing a particular role need to be refactored in a similar fashion, allowing us to describe refactorings in terms of the role elements comprising the concern, and not in terms of concrete program elements. A role-based CCC refactoring description consists of the following elements:

1. **Concern description** — The CCC in question is described in terms of its role elements and structure. The structure of a CCC is determined by the relationships between its role elements, such as *contains* relationships (between types and methods and fields, for example) and method call relationships.
2. **Refactoring instructions** — Refactoring instructions are defined in terms of the role elements in the concern description. These instructions consist of a composite of traditional and aspect-oriented refactorings. For example, one step could entail replacing all methods playing the role of the *handleRequest(Request)* method with corresponding methods on an aspect.

The application of a CCC refactoring comprises the following three steps:

1. **Refactoring selection** — First, a CCC refactoring description is chosen from the library of CCC refactorings. Alternatively, a new CCC refactoring description is created and added to the library.

2. **Mapping of roles** — Next, the role elements are mapped onto concrete program elements in the target system to be refactored. Tool support can leverage the concern structure to identify the concern *extent*, i.e., the program elements pertaining to its implementation.
3. **Refactoring execution** — The program is transformed by applying the abstract refactoring instructions to the concrete program elements mapped to the role elements.

Since the refactoring description is based on the concern structure and not tied to a particular implementation, role-based CCC refactorings can be developed independently from concrete concern implementations.

1.3 Overview of the Dissertation

In the remainder of this dissertation we present our role-based approach and demonstrate how it can be used to support the refactoring of non-modularized crosscutting concerns into aspects.

Chapter 2 investigates the underlying assumption of this work that refactorings of non-modularized crosscutting concerns yield modularity benefits for at least a useful set of crosscutting concerns. The chapter presents a study comparing design pattern implementations (as sample concerns) in Java and AspectJ with respect to their modularity properties. It investigates the 23 Gang-of-Four (GoF) design patterns presented by Gamma et. al. [28]. The study shows that for most of the investigated CCCs an aspect-oriented implementation entails modularity benefits, and that OO and AO implementations of crosscutting concerns can have complex structural differences, rendering a transformation from OO to AO non-straightforward.

In Chapter 3, we employ the concept of roles to model crosscutting concerns and their refactorings and present the resulting role-based refactoring approach. The approach is general and language-independent. More specifically, the concern model and mapping mechanism are language independent, while refactoring instructions are loosely tied to the programming language used. The approach consists of three parts: first, a role-based concern model that allows specification of the principal elements of a crosscutting concern in terms of their roles within the concern, and the relationships between these elements. Second, a mapping between the role elements and the program elements implementing them. And finally, a model for describing aspect-oriented refactorings in terms of role elements identified in the concern description.

Chapter 4 presents a proof-of-concept implementation of our role-based approach that supports the semi-automated refactoring of crosscutting concerns in form of a refactoring tool. The tool is an instantiation of the approach for Java and AspectJ as representative object-oriented and aspect-oriented languages, respectively. It implements the role-based concern model, mapping facilities and the expression of role-based refactorings. For the description of CCC refactorings, we introduce a library of refactoring primitives. The tool itself is realized as a plugin for the Eclipse IDE [20, 91]. Its implementation and application to code examples with non-modularized crosscutting concerns constitute the basis for validating the thesis claim.

In Chapter 5 we evaluate our approach by applying the tool to refactor a variety of non-modularized crosscutting concerns into aspects, investigating how the tool performs on a non-trivial software system, and by studying the effort involved in developing new CCC refactorings. This chapter shows that our role-based concern model enables the semi-automated refactoring of crosscutting concerns.

Chapter 6 discusses open issues pertaining to the main facets of our approach and tool and outlines future work. In Chapter 7 we address related work, putting our work into perspective and highlighting its novelty. Finally, Chapter 8 reviews the thesis of this dissertation and summarizes our findings.

This dissertation makes four contributions to the field of software engineering. First, it demonstrates through a refactoring tool for Java and AspectJ that a role-based approach can semi-automate CCC refactorings. Second, the role-based model for describing crosscutting concerns may form the basis for other operations that need to capture or analyze CCCs, such as documenting a crosscutting concern or identifying inconsistencies in its implementation. Third, the thesis includes a description of a set of aspect-oriented refactoring primitives. Finally, we describe and evaluate aspect-oriented implementations of common design patterns, which developers may use independently of our refactoring approach.

Chapter 2

Design Pattern Implementations in Java and AspectJ

This dissertation is based on the underlying assumption that the transformation of non-modularized crosscutting concerns into aspects is useful. We explore this assumption in a study that investigates implementations for a particular — and often crosscutting — kind of concern: design pattern implementations.

Design patterns offer flexible solutions to common software development problems. Each pattern is comprised of a number of parts, including purpose/intent, applicability, solution structure, and sample implementations.

A number of the Gang-of-Four (GoF) patterns [28] involve crosscutting structures in the relationship between roles in the pattern and classes in an actual implementation [17, 18]. In the OBSERVER pattern, for example, all operations that change a *Subject* must trigger notifications of its *Observers*. In other words, the act of notification crosscuts one or more operations in each *Subject* in the pattern. In the CHAIN OF RESPONSIBILITY pattern, all *Handlers* need to be able to accept requests and to either handle them or forward them to the successor in the chain. This event handling mechanism crosscuts the classes playing the *Handler* role.

When the GoF patterns were first identified, the sample implementations were geared to the current state of the art in object-oriented languages. Other work [66, 78] has shown that implementation language affects pattern implementation, so it seems natural to explore the effect of aspect-oriented programming techniques [46] on the implementation of the GoF patterns.

In our study of the GoF patterns, we maintain the purpose, intent, and applicability of 23 well-known patterns, and allow only the solution structure and implementation to change. Thus, the focus of this study is not to discover new patterns, but simply to work out how implementations of the GoF patterns can be handled using a new implementation technique.

We compared object-oriented and aspect-oriented implementations of sample systems for each of the 23 GoF design patterns. Our results show that using AspectJ improves the modularity of implementations of many GoF patterns. In some cases this is reflected in a new solution structure with fewer or different participants, in other cases, the structure remains the same, only the implementation changes.

The degree of improvement in implementation modularity varies, with the greatest improvement coming when the pattern solution structure involves crosscutting of some form as indicated by the usage of roles, for example one object playing multiple roles, many objects playing one role, or an object playing roles in multiple pattern instances.

Patterns with crosscutting structure between roles and participant classes see the most improvement. The improvement comes primarily from modularizing the implementation of the pattern. This is directly reflected in the implementation being textually localized. An integral part of achieving this is to remove code-level dependencies between the participant classes and the implementation of the pattern.

Using AspectJ, we modularized the implementation of 17 of the patterns, while the other 6 can already be modularly expressed using object-oriented techniques. For 12 of the patterns, the aspect-oriented modularity enabled a core part of the implementation to be abstracted into reusable code. For 14, it enabled transparent composition of pattern instances, so that multiple patterns can have shared participants. For the 17 modularized patterns, all pattern code from some or all participants was moved into the pattern aspect, allowing those participants to be (un)pluggable with respect to the pattern.

These results — 74% of GoF patterns implemented in a more modular way, and 52% reusable — suggest that there is merit in replacing object-oriented design pattern implementations with aspect-oriented ones arising from improved modularity.

2.1 Established Challenges

Numerous authors have identified challenges that arise when patterns are concretized in a particular software system, as outlined below. The three most important challenges are related to implementation, documentation, and composition:

Design pattern implementation usually has a number of undesirable related effects. Because patterns influence the system structure and their implementations are influenced by it [25], pattern implementations are often tailored to the instance of use, which can lead to them “disappearing into the code” [25] and losing their modularity [76]. This makes it hard to distinguish between the pattern, the concrete instance and the object model involved [55]. Adding or removing a pattern to/from a system is often an invasive change that is difficult to reverse [14]. Consequently, while a design pattern description is reusable, its implementations usually are not [76].

The invasive nature of pattern code, and its scattering and tangling with other code creates documentation problems [76]. If multiple patterns are used in a system, it can become difficult to trace particular instances of a design pattern, especially if classes are involved in more than one pattern, as is the case with pattern overlay/composition [1].

Pattern composition causes more than just documentation problems. It is inherently difficult to reason about systems with multiple patterns involving the same classes, since the composition creates large clusters of mutually dependent classes [76]. This is an important topic because some design patterns explicitly use others patterns in their solution.

2.2 Study Format

The findings presented in this chapter are based on a comparative analysis of Java and AspectJ implementations of the GoF design patterns.

For each of the 23 GoF patterns we created a sample system that makes use of the pattern, and implemented the example in both Java and AspectJ. The Java implementations correspond to the sample C++ implementations in the GoF book, with minor adjustments to account for the differences between C++ and Java, such as the lack of multiple inheritance, etc. Most patterns have a number of implementation variants and alternatives. If a pattern offered more than one possible implementation, we picked the one that appeared to be the most generally applicable.

The AspectJ implementations were developed iteratively. The AspectJ constructs allowed a number of different implementations, usually with varying tradeoffs. Our goal was to investigate the design space of clearly defined patterns. We ended up creating a total of 57 different implementations, which ranged from 1 to 7 per pattern. Some of the tradeoffs and design decisions are discussed in Section 2.3.

2.3 Results

This section presents a comparison of the AspectJ and Java implementations of concrete instances of the GoF design patterns. Section 2.3.1 is a detailed discussion of the OBSERVER pattern. We use this discussion to present properties common to most of the AspectJ solutions. The remaining patterns are presented by building on the concepts developed in Section 2.3.1.

2.3.1 Example: The Observer Pattern

The intent of the OBSERVER pattern is to “define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically” [28]. Object-oriented implementations of the OBSERVER pattern, such as the sample code in the GoF book (p. 300–303), usually add a field to all potential *Subjects* that stores a list of *Observers* interested in that particular *Subject*. When a *Subject* wants to report a state change to its *Observers*, it calls its own *notify()* method, which in turn calls an *update(...)* method on all *Observers* in the list.

Consider a concrete example of the OBSERVER pattern in the context of a simple figure package as shown in Figure 2.1. In such a system the OBSERVER pattern would be used to cause mutating operations on figure elements to update the *Display*. As shown, code for implementing this pattern is spread across the classes.

All participants have to know about their role in the pattern and consequently have pattern code in them. Adding or removing a role from a class requires changes to that class. For example, changing the notification mechanism, such as switching between *push* and *pull* models [28], requires changes in all participating classes.

Figure 2.1: A simple graphical figure element system that uses the OBSERVER pattern in Java. The assignment of *Subject* and *Observer* roles are indicated using UML stereotypes [73]. Underlined methods contain code necessary to implement this instance of the pattern.

The Abstracted Observer Pattern

In the structure of the OBSERVER pattern, some parts are common to all potential instantiations of the pattern, and other parts are specific to each instantiation. The parts common to all instantiations are:

1. The existence of *Subject* and *Observer* roles, i.e., the fact that some classes act as *Observer* and some as *Subject*
2. Maintenance of a mapping from *Subjects* to *Observers*
3. The general update logic: *Subject* changes trigger *Observer* updates

The parts specific to each instantiation of the pattern are:

4. Which classes can be *Subjects* and which can be *Observers*
5. A set of changes of interest on the *Subjects* that trigger updates on the *Observers*
6. The specific means of updating each kind of *Observer* when the update logic requires it

Listing 2.1: The generalized ObserverProtocol aspect

```

1  public abstract aspect ObserverProtocol {
2
3      protected interface Subject { }
4      protected interface Observer { }
5
6      private WeakHashMap perSubjectObservers;
7
8      protected List getObservers(Subject s) {
9          if (perSubjectObservers == null) {
10             perSubjectObservers = new WeakHashMap();
11          }
12          List observers = (List) perSubjectObservers.get(s);
13          if (observers == null) {
14             observers = new LinkedList();
15             perSubjectObservers.put(s, observers);
16          }
17          return observers;
18      }
19
20      public void addObserver(Subject s, Observer o){
21          getObservers(s).add(o);
22      }
23
24      public void removeObserver(Subject s, Observer o){
25          getObservers(s).remove(o);
26      }
27
28      abstract protected pointcut subjectChange(Subject s);
29
30      abstract protected void updateObserver(Subject s, Observer o);
31
32      after(Subject s): subjectChange(s) {
33          Iterator iter = getObservers(s).iterator();
34          while ( iter.hasNext() ) {
35              updateObserver(s, ((Observer)iter.next()));
36          }
37      }
38 }

```

We developed AspectJ code⁵ that reflects this separation of reusable and instance-specific parts. An abstract aspect encapsulates the generalizable parts (1-3), while one concrete extension of the aspect for each instance of the pattern fills in the specific parts (4-6). The reusable ObserverProtocol aspect is shown in Listing 2.1.

The Subject and Observer roles The roles are realized as protected inner interfaces named Subject and Observer (Figure 2.1, line 3-4). Their main purpose is to allow correct typing of *Subjects* and *Observers* in the context of the pattern implementation, such as in methods like `addObserver( . . )`. Concrete extensions of the ObserverProtocol aspect assign the roles to particular classes.

These interfaces are protected because they will only be used by ObserverProtocol and its concrete extensions. No code outside the aspect and extensions needs to handle objects in terms of these roles.

⁵All examples and the code generated by our refactoring tool (Chapter 4) use AspectJ 1.2.1. The recent AspectJ 1.5 milestone builds (which support Java 5) would allow for a slightly more elegant implementation. Concretely, we could employ generic versions of the internally used abstract data types to avoid type casts.

These interfaces are empty because the pattern defines no methods on the *Subject* or *Observer* roles. The methods that would typically be defined on the *Subject* and *Observer* are instead defined on the aspect itself.

For patterns that were abstractable we had to decide where to put the role interfaces. Two locations are possible: Either as a protected interface inside the abstract aspect or as a separate public interface. We made this decision based on whether the role interface introduces client-accessed functionality, i.e., exposes functionality to clients (as for STRATEGY, ITERATOR, etc.) or not (as in the OBSERVER case). If the role has no client-accessible functionality, it will only be referenced from within pattern aspects. For that reason, we placed it in the abstract aspect. In the other case, we made the interface public.

The Subject-Observer Mapping Implementation of the mapping in the AspectJ code is localized to the `ObserverProtocol` aspect. It is realized using a weak hash map of linked lists to store the *Observers* for each *Subject* (line 6). As each pattern instance is represented by a concrete subaspect of `ObserverProtocol`, each instance will have its own mapping.

Changes to the Subject-Observer mappings can be realized via the public `addObserver(...)` and `removeObserver(...)` methods (line 20-26) that concrete subaspects inherit. To have a `Display` object *D* become the *Observer* of a `Point` *Subject* *P*, clients call these methods on the appropriate subaspect (e.g., `ColorObserver`):

```
ColorObserver.aspectOf().addObserver(P, D);
```

The `getObservers(...)` method is only used internally. It creates the proper secondary data structures (linked lists) on demand (line 8-18). Note that in this implementation the Subject-Observer mapping data structure is centralized in each concrete extension. All concrete aspects that subclass the abstract pattern aspect will automatically have an individual copy of the field, following the structure presented in [28]. This can cause bottlenecks in some situations. These can be fixed, on a per pattern-instance basis, by overriding `getObservers(...)` with a method that uses a more decentralized data structure.

Generally, whenever a pattern solution requires a mapping between participants, as in the case of the `successor` field of handlers in CHAIN OF RESPONSIBILITY, and the pattern implementation is abstractable, we can either define a field on the participant, or keep the mapping in a central data structure in the abstract aspect as in this example. Whichever approach is chosen, the point of access to the data structure is the instance-specific aspect, so that different instances of the pattern involving the same participants are possible and will not become confused.

The Update Logic In the reusable aspect, the update logic implements the general concept that *Subjects* can change in ways that require all their observing objects to be updated. This implementation does not define exactly what constitutes a change, or how *Observers* should be updated. The general update logic consists of three parts:

The changes of interest depict *conceptual operations*, a set of points in program execution, at which a *Subject* should update its *Observers* (to notify them of changes

to its state). In AspectJ, sets of such points are identified with pointcut constructs. In the reusable aspect, we only know there are modifications of interest, but since they are instance-specific, we do not know what they are. Therefore, we define an abstract pointcut named `subjectChange( . . )` that is to be concretized by instance-specific subaspects (line 28).

In the reusable part we only know that the *Observers* will have to be updated in the context of the pattern, but cannot predict how that is best achieved. We define an abstract update method `updateObserver(Subject)` that will be concretized for each pattern instance (line 30). That way, each instance of the OBSERVER pattern can choose its own update mechanism.

Finally, the reusable aspect implements the update logic in terms of the generalizable implementation parts mentioned above. This logic is contained in the after advice (line 32-37). This after advice says: whenever execution reaches a join point matched by the `subjectChange( . . )` pointcut, update all *Observers* of the appropriate *Subject* afterwards.

Pattern-Instance-Specific Concrete Aspects

Each concrete subaspect of `ObserverProtocol` defines one particular kind of observing relationship, in other words a single pattern instance. Within that kind of relationship, there can be any number of *Subjects*, each with any number of *Observers*. The subaspect defines three things:

- The classes that play the roles of *Subject* and *Observer*. This is done using the `declare parents` construct, which adds superclasses or super-interfaces to a class, to assign the roles defined in the abstract aspect.
- The conceptual operations on the *Subject* that require updating the *Observers*. This is done by concretizing the `subjectChange( . . )` pointcut.
- How to update the *Observers*. This is done by concretizing `updateObserver(Subject)`. The choice between push or pull model for updates is no longer necessary as we have access to both the *Subject* and the *Observer* at this point and can customize the updates.

The `declare parents` construct is part of the AspectJ open class mechanism that allows aspects to modify existing classes without changing their code. This open class mechanism can attach fields, methods, or — as in this case — interfaces to existing classes.

Listing 2.2 shows two different instances of the OBSERVER pattern involving the classes `Point`, `Line`, and `Display`. In both instances, `Point` and `Line` play the role of *Subject*, and `Display` plays the role of *Observer*. The first observes color changes, and the second observes coordinate changes.

Particular classes can play one or both of the *Subject* and *Observer* roles, either in the same pattern instance or separate pattern instances. Listing 2.3 shows a third pattern instance in which `Display` acts as *Subject* and *Observer* at the same time.

Listing 2.2: Two different Observer instances

```

1 public aspect ColorObserver extends ObserverProtocol {
2
3     declare parents: Point implements Subject;
4     declare parents: Line implements Subject;
5     declare parents: Display implements Observer;
6
7     protected pointcut subjectChange(Subject s):
8         (call(void Point.setColor(Color))
9          || call(void Line.setColor(Color)) ) && target(s);
10
11     protected void updateObserver(Subject s, Observer o) {
12         ((Display)o).display("Color change for "+s);
13     }
14 }
15
16 public aspect CoordinateObserver extends ObserverProtocol {
17
18     declare parents: Point implements Subject;
19     declare parents: Line implements Subject;
20     declare parents: Display implements Observer;
21
22     protected pointcut subjectChange(Subject s):
23         (call(void Point.setX(int))
24          || call(void Point.setY(int))
25          || call(void Line.setP1(Point))
26          || call(void Line.setP2(Point)) ) && target(s);
27
28     protected void updateObserver(Subject s, Observer o) {
29         ((Display)o).display("Coordinate change for "+s);
30     }
31 }

```

Listing 2.3: The same class can be *Subject* and *Observer*

```

1 public aspect DisplayObserver extends ObserverProtocol {
2
3     declare parents: Display implements Subject;
4     declare parents: Display implements Observer;
5
6     protected pointcut subjectChange(Subject s):
7         call(void Display.display(String)) && target(s);
8
9     protected void updateObserver(Subject s, Observer o) {
10         ((Display)o).refresh();
11         ((Display)o).log("Display updated by "+s);
12     }
13 }

```

Figure 2.2: The structure of an instance of the OBSERVER pattern in AspectJ. *Subject* and *Observer* roles crosscut classes, and the changes of interest (the `subjectChange()` pointcut) crosscuts methods in various classes

In the AspectJ version all code pertaining to the relationship between *Observers* and *Subjects* is moved into an aspect, which changes the dependencies between the modules. Figure 2.2 shows the structure for this case.

Properties of this Implementation

This implementation of the OBSERVER pattern has the following closely related modularity properties:

- **Locality** — All the code that implements the OBSERVER pattern is contained in one abstract and one concrete aspect, none of it is in the participant classes. The participant classes are entirely free of the pattern context, and as a consequence there is no coupling between the participants. Potential changes to each OBSERVER pattern instance are confined to one place.
- **Reusability** — The core pattern code is abstracted and reusable. The implementation of `ObserverProtocol` is generalizing the overall pattern behavior. The abstract aspect can be reused and shared across multiple OBSERVER pattern instances. For each pattern instance, we only need to define one concrete aspect.
- **Composition transparency** — Because a pattern participant's implementation is not coupled to the pattern, if a *Subject* or *Observer* takes part in multiple observing relationships their code does not become more complicated and the pattern instances are not confused. Each instance of the pattern can be reasoned about independently.
- **(Un)pluggability** — Because *Subjects* and *Observers* need not be aware of their role in any pattern instance, it is possible to switch between using a pattern and not using it in the system.

Note that the OO and the AO versions of the patterns very considerably structurally. Similar observations can be made for the other patterns as well, indicating that a potential CCC refactoring that replaces an OO implementation with an aspect can be complex.

2.3.2 Other GoF Patterns

In the following we describe the remaining 22 GoF patterns and how the AspectJ implementation is different from a pure Java version. The patterns are grouped by common features, either of the pattern structures or their AspectJ implementations.

Composite, Command, Mediator, Chain of Responsibility: Roles Only Used Within the Pattern Aspect

Similar to the OBSERVER pattern, these patterns introduce roles that need no client-accessible interface and are only used within the pattern. In AspectJ such roles are realized with empty (protected) interfaces. The types they introduce are used within the pattern protocol. One abstract aspect for each pattern defines the roles and attaches default implementations where possible. See Listing 2.4 for parts of the abstract `CompositionProtocol` aspect.

For patterns involving particular conceptual operations, the abstract pattern aspect introduces an abstract pointcut to be concretized for each instance of the pattern, which captures the join points that should trigger important events, such as the execution of a *Command* in the COMMAND pattern. As in the OBSERVER example, advice (after, before, or around) is responsible for calling the appropriate methods.

In the COMPOSITE case, to allow walking the tree structure inherent to the pattern, we define facilities to have a visitor traverse and/or change the structure. These visitors are defined in the concrete aspect. See Listing 2.5 for an example of how statistics can be collected from the composite structure. In this example we show an instance of the COMPOSITE pattern modeling a file system. Directories are *Composites*, and files are *Leafs*. The example shows how to calculate the disk space needed for the file system, assuming that `File` objects have a size field. Again, clients use a public method on the aspect to access the new functionality. Appropriate methods on the participants are introduced privately and are visible only by the aspect.

Singleton, Prototype, Memento, Iterator, Flyweight: Aspects as Object Factories

These patterns administrate access to specific object instances. All of them offer factory methods to clients and share a create-on-demand strategy. The patterns have abstracted, reusable implementations in AspectJ, with code for the factory in the aspect.

In the AspectJ implementations, the factory methods are either parameterized methods on the abstract aspect or methods attached to the participants. If the former approach is used, multiple instances of the pattern compose transparently, even if all factory methods have the same names. The SINGLETON case is special in that we can turn the original constructor into the factory method using around advice and returning the unique object on all constructor calls.

Listing 2.4: Part of the abstract Composite pattern implementation

```

1  public abstract aspect CompositionProtocol {
2
3      protected interface Component {}
4      protected interface Composite extends Component {}
5      protected interface Leaf extends Component {}
6
7      private WeakHashMap perComponentChildren = new WeakHashMap();
8
9      private Vector getChildren(Component s) {
10         Vector children;
11         children = (Vector)perComponentChildren.get(s);
12         if ( children == null ) {
13             children = new Vector();
14             perComponentChildren.put(s, children);
15         }
16         return children;
17     }
18
19     public void addChild(Composite composite, Component component) {
20         getChildren(composite).add(component);
21     }
22     public void removeChild(Composite composite, Component component) {
23         getChildren(composite).remove(component);
24     }
25
26     public Enumeration getAllChildren(Component c) {
27         return getChildren(c).elements();
28     }
29
30     protected interface FunctionVisitor {
31         public Object doIt(Component c);
32     }
33
34     protected static Enumeration recurseFunction(Component c,
35                                                    FunctionVisitor fv) {
36         Vector results = new Vector();
37         for (Enumeration enum = getAllChildren(c); enum.hasMoreElements(); ) {
38             Component child;
39             child = (Component)enum.nextElement();
40             results.add(fv.doIt(child));
41         }
42         return results.elements();
43     }
44 }

```

Listing 2.5: Part of a COMPOSITION pattern instance aspect

```

1  public aspect FileSystemComposite extends CompositeProtocol {
2
3      declare parents: Directory implements Composite;
4      declare parents: File implements Leaf;
5
6      public int sizeOnDisk(Component c) {
7          return c.sizeOnDisk();
8      }
9
10     private abstract int Component.sizeOnDisk();
11
12     private int Directory.sizeOnDisk() {
13         // Assume directories have no relevant size of their own
14         int diskSize = 0;
15
16         Enumeration enum = SampleComposite.aspectOf().getAllChildren(this);
17         while (enum.hasMoreElements()) {
18             diskSize += ((Component)enum.nextElement()).sizeOnDisk();
19         }
20         return diskSize;
21     }
22
23     private int File.sizeOnDisk() {
24         return size;
25     }
26 }

```

Parameterized factory methods can alternatively be implemented according to Nordberg's factory example [65]: the factory method is empty (returns null or a default object). Other return values are provided by around advice on that method. If the arguments are appropriate, the advice creates a new matching object; otherwise it just proceeds with the regular execution. This allows us extend the factory (in terms of new products) without changing its code. Participants no longer need to have pattern code in them; the otherwise close coupling between an original object and its representation or accessor (MEMENTO, ITERATOR) is removed from the participants.

Adapter, Decorator, Strategy, Visitor, Proxy: Language Constructs

Using AspectJ, the implementation of some patterns completely disappears, because AspectJ language constructs implement them directly. This applies to these patterns in varying degrees.

The ADAPTER and VISITOR patterns can be realized by extending the interface of the *Adaptee* (via AspectJ's open class mechanism). DECORATOR, STRATEGY and PROXY have alternate implementations based on attaching advice (as outlined for DECORATOR in [65]).

While simpler and more modular, the approaches have inherent limitations. The advice-based implementation of DECORATOR loses its dynamic manipulation properties (dynamic reordering of *Decorators*) and is thus less flexible. The interface augmentation for *Adapter* cannot be realized in this manner when we want to replace an existing method with another one that has the same name and arguments but a different return type.

Protection or delegation proxies can be implemented to be reusable using the above approach, but some applications of the PROXY pattern require the *Proxy* and the *Subject* to be two distinct objects (as is the case for remote and virtual proxies). In these cases the Java and AspectJ implementations are identical.

Abstract Factory, Factory Method, Template Method, Builder, Bridge: Multiple Inheritance

These patterns are structurally similar: inheritance is used to distinguish different, but related, implementations. As this is already nicely realized in OO, these patterns could not be given more reusable implementations. However, with AspectJ it is possible to replace the abstract classes mentioned in the GoF solution by interfaces without losing the ability to attach (default) implementations to their methods. With Java, we cannot use interfaces if we want to define a default implementation for methods that are part of the pattern code. In that respect, AspectJ's open class mechanism effectively provides a limited form of multiple inheritance.

Besides that, BUILDER and BRIDGE have the following additional implementation considerations. For BUILDER, an aspect can intercept calls to the creation methods and replace them with alternate implementations using around advice (see STRATEGY above). For BRIDGE, a decoupling of *Abstraction* and *Implementor* can be achieved by using polymorphic advice as suggested by Nordberg [88]. While this approach reduces the coupling between the participants, it is less flexible with respect to dynamically changing *Implementors*.

State, Interpreter: Scattering Contained

These patterns introduce tight coupling between their participants. In the AspectJ implementations, parts of the scattered code can be modularized.

In the STATE pattern, the crosscutting code for state transitions can be modularized in an aspect using (mainly) after advice. Although the INTERPRETER pattern implementation is not modularized, extensions to the pattern that augment or change the behavior of the system without can be modularly implemented by attaching methods to the participants using the open class mechanism.

Façade: no benefit from AspectJ implementation

For this pattern, the AspectJ approach is not structurally different from the Java implementation. FAÇADE provides a unified interface to a set of interfaces to a subsystem, to make the subsystem easier to use. This example mainly requires namespace management and good coding style. On the other hand, AspectJ's `declare warning` and `declare error` constructs provide simple means to impose a global policy for handling violations of the FAÇADE principle (accesses to the protected subsystem that bypass the *Façade* interface).

2.4 Analysis

In this section, we present an analysis of the previously observed benefits of implementing patterns with AspectJ. The analysis is broken into three parts:

- The general improvements observed in many pattern re-implementations.
- The specific improvements associated with particular patterns.
- The origins of crosscutting structure in patterns, and a demonstration that observed improvements correlate with the presence of crosscutting structure in the pattern.

2.4.1 General Improvements

For a number of patterns, the AspectJ implementations manifest several closely related modularity benefits: locality, reusability, dependency inversion, transparent composability, and (un)pluggability. Attempting to say which of these is primary is difficult, instead we simply describe them and discuss some of their interrelationships.

The AspectJ implementations of 17 of the 23 GoF patterns were localized. For 12 of these, the locality enables a core part of the implementation to be abstracted into reusable code. In 14 of the 17 we observed transparent composability of pattern instances, so that multiple patterns can have shared participants (see Table 2.1).

The improvements in the AspectJ implementations are primarily due to inverting dependencies, so that pattern code depends on participants, not the other way around. This is directly related to locality: all dependencies between patterns and participants are localized in the pattern code.

An object or class that is oblivious of its role in a pattern can be used in different contexts (such as outside the pattern) without modifications or redundant code, thereby increasing the reusability of participants. If participants do not need to have pattern-specific code, they can be readily removed from or added to a particular pattern instance, making the participants (un)pluggable. To benefit from this, the participants must have a meaning outside the pattern implementation. For example, the participants in a CHAIN OF RESPONSIBILITY pattern often have other responsibilities in the application they are in (as widgets in the GUI example in GoF), while *Strategy* objects in the STRATEGY pattern usually just encapsulate an algorithm.

The locality also means that existing classes can be incorporated into a pattern instance without the need to adapt them; all the changes are made in the pattern instance. This makes the pattern implementations themselves relatively (un)pluggable.

Since pattern locality implies that all code fragments pertaining to the implementation of a pattern are in one place, it also facilitates to impose global policies related to a design pattern, such as adding thread safety, logging facilities or performance optimizations. For example, in the CHAIN OF RESPONSIBILITY case, the policy that determines what happens to requests that reach the end of the chain without being answered is contained to a single point within the aspect instead of being spread across

Table 2.1: Design Patterns, desirable properties of their AspectJ implementations, and usage of roles

Pattern Name	Modularity Properties				Kinds of Roles	
	Locality ^a	Reusability	Composition Transparency	(Un)plug-gability	Defining ^b Roles	Superimposed Roles
Facade	Same implementation for Java and AspectJ				Facade	-
Abstract Factory	no	no	no	no	Factory Product	-
Bridge	no	no	no	no	Abstraction Implementor	-
Builder	no	no	no	no	Builder, (Director)	-
Factory Method	no	no	no	no	Product Creator	-
Interpreter	no	no	n/a	no	Context Expression	-
Template Method	(yes)	no	no	(yes)	(AbstractClass)	(ConcreteClass)
Adapter	yes	no	yes	yes	Target, Adapter	Adaptee
State	(yes)	no	n/a	(yes)	State	Context
Decorator	yes	no	yes	yes	Component Decorator	Concrete-Component
Proxy	(yes)	no	(yes)	(yes)	(Proxy)	(Proxy)
Visitor	(yes)	yes	yes	(yes)	Visitor	Element
Command	(yes)	yes	yes	yes	Command	Commanding Receiver
Composite	yes	yes	yes	(yes)	(Component)	Composite Leaf
Iterator	yes	yes	yes	yes	(Iterator)	Aggregate
Flyweight	yes	yes	yes	yes	Factory	Flyweight
Memento	yes	yes	yes	yes	Memento	Originator
Strategy	yes	yes	yes	yes	Strategy	Context
Mediator	yes	yes	yes	yes	-	(Mediator) Colleague
Chain of Respons.	yes	yes	yes	yes	-	Handler
Prototype	yes	yes	(yes)	yes	-	Prototype
Singleton	yes	yes	n/a	yes	-	Singleton
Observer	yes	yes	yes	yes	-	Subject Observer

^a Locality: "(yes)" means that the pattern is localized in terms of its superimposed roles but the implementation of the remaining defining role is still done using multiple classes (e.g., *States* for the *STATE* pattern). In general, (yes) for a desirable property means that some restrictions apply.

^b Defining and superimposed roles were not always easy to differentiate. If the distinction was not clear, the role names are shown in parentheses in either or both categories

the various *Handler* types. In the OBSERVER case, a conceivable performance optimization is to de-centralize the mapping between *Subjects* and *Observers* (see Section 2.3.1 above), which can also be achieved by changing solely the aspect.

In essence, we observe typical advantages generally associated with localized concerns with regards to future changes and program evolution. In particular, the problematic case of pattern composition and overlay [1, 25, 55, 76] becomes better structured (and easier to reason about) when pattern instances are defined in separate modular units.

In addition to code-level benefits, the modularity of the design pattern implementation also results in an inherent documentation benefit. As mentioned in [1, 76], the mere existence of classes that exclusively contain pattern code serve as records of what patterns are being used. In the AspectJ cases, we observe two additional improvements. First, all code related to a particular pattern instance is contained in a single module (which defines participants, assigns roles, etc.). This means that the entire description of a pattern instance is localized and does not “get lost” [76] or “degenerate” [25] in the system. Secondly, with the current AspectJ IDE support, all references, advised methods etc. are hyperlinks that allow a developer an overview of the assignment of roles and where the conceptual operations of interest are.

In 12 cases we were able to develop reusable pattern implementations. This happened by generalizing the roles, pattern code, communication protocols, and relevant conceptual operations in an abstract reusable aspect. For any concrete instance of the pattern, the developer defines the participants (assigns roles) and fills in instance-specific code. Changes to communication protocols or methods that are part of the abstract classes or interfaces involved do not require adjusting all participants.

If we can reuse generalized pattern code and localize the code for a particular pattern instance, multiple instances of the same pattern in one application are not easily confused (composition transparency). The same participating object or class can even assume different roles in different instances of the same pattern (see the OBSERVER example above). This solves a common problem with having multiple instances of a design pattern in one application.

2.4.2 Specific Improvements

The Singleton case

The AspectJ version of the pattern implementation opened up two new design options that are not possible in Java: First, is *Singleton* an inherited property, or do we have an inheritance anomaly? Second, do we want a devoted factory method to provide the *Singleton* instance, or do we want the constructor to return it whenever it is called?

We decided to implement the *Singleton* property as inherited, but provided facilities to exclude specific subclasses from the pattern’s constructor protection if desired.

For the second, we decided that using the constructor instead of a devoted factory method was beneficial. The factory, if desired, can then be implemented either directly in the class, or as a transparently composed aspect.

Multiple Inheritance and Java

As originally presented, some of the GoF patterns make use of multiple-inheritance in their proposed implementation, for example the class version of the ADAPTER pattern. For many patterns, the roles that participants play within the patterns are realized as abstract classes in Java. Participant classes inherit interfaces and default implementations from these abstract classes. But if the participant classes have functionality outside the pattern context (such as GUI widgets as *Subjects* or *Observers* in the OBSERVER pattern), they are usually already part of an inheritance hierarchy. Since Java lacks multiple inheritance, implementation in these cases can be somewhat awkward: In Java, if a participant has to inherit both its role and its other functionality, then one of the supertypes has to be realized as an interface. Unfortunately, interfaces in Java cannot contain code, making it impossible to attach default implementations of methods, for example.

The open class mechanism in AspectJ provides us with a more flexible way of implementing these patterns, as it allows to attach both interfaces and implementations (code) to existing classes.

Note that this does not solve the problems typically associated with multiple inheritance, such as the ambiguities that arise when names of members of multiple supertypes collide, or, more specifically, the “diamond” problem [13]. Should such naming collisions occur, they are detected by the AspectJ compiler and need to be resolved by the developer.

Breaking Cyclic Dependencies

Some design patterns regulate complex interactions between sets of objects. In object-oriented implementations these classes are tightly coupled and mutually dependent. One example of a design pattern that introduces cyclic dependencies is MEDIATOR, a variation of the OBSERVER pattern that is often used in UI programming. Here, changes to *Colleagues* (e.g., widgets) trigger updates in the *Mediator* object. The *Mediator*, on the other hand, might update some or all of the *Colleagues* as a reaction to this.

A typical structure for this pattern is shown in Figure 2.3 (a). Inheritance relationships (the *Mediator* and *Colleague* interface) are not shown. The pattern introduces cyclic dependencies between *Mediator* and *Colleagues* (denoted by arrows pointing in opposite directions). The pattern code (for updates, etc.) is distributed both over the *Mediator* and all *Colleagues*.

In the AspectJ implementation shown in Figure 2.3 (b), the indirection introduced by the *ConcreteMediator* aspect removes the cyclic dependencies. The aspect defines the participants, assigns the roles and identifies which points in the execution trigger updates. *Colleagues* do not have to have any pattern-related code in them, they are “freed” of the pattern. Changes to the pattern (for example, the notification interface) are limited to a single module: the aspect. Again, an abstract aspect (here: *MediatorProtocol*) implements generalizable parts of the pattern.

Figure 2.3: Dependencies and code distribution in typical instances of the MEDIATOR pattern. The AspectJ implementation removes cyclic dependencies and localizes the pattern code

2.4.3 Crosscutting Structure of Design Patterns

This section identifies the origins of crosscutting structure in the patterns, shows that the observed benefits of using AspectJ in a pattern implementation correlate with crosscutting in the pattern, and illustrates how the usage of roles within the pattern is indicative for their crosscutting nature.

Roles define the behavior and functionality of participants in a pattern. Examples of such roles are *Component*, *Leaf* and *Composite* for the COMPOSITE pattern, *Subject* and *Observer* for the OBSERVER pattern, or *Abstract-* and *ConcreteFactory* for the ABSTRACT FACTORY pattern. Crosscutting in pattern structure is caused by different kinds of roles and their interaction with participant classes.

In some patterns, the roles are *defining*: the participants have no functionality outside the pattern. That is, the roles define the participants completely. Objects that play the *Façade* role, for example, provide a unified interface to a subsystem and usually have no other behavior of their own. Defining roles often include a client-accessible interface.

In other patterns, the roles are *superimposed*: they are assigned to classes that have functionality and responsibility outside the pattern. In the OBSERVER pattern for example, the classes that play *Subject* and *Observer* roles do more than just fulfilling the pattern requirements. In a GUI context, *Subjects* could be widgets, for example. In other words, classes with superimposed roles have behavior outside the pattern context. The *Subject* role is thus only an augmentation of the already existing class. Superimposed roles usually do not have a client-accessible interface.

In object-oriented programming, defining roles are often realized by subclassing an abstract superclass to achieve different but related behaviors; superimposed roles are often interfaces that specify behavior and responsibilities.

Figure 2.4: Crosscutting caused by pattern composition. A second pattern instance introduces additional crosscutting

Roles and Crosscutting

Superimposed roles lead to three different kinds of crosscutting among patterns and participants:

- Roles can crosscut participant classes. That is, for one role, there can be n classes, and one class can have n roles; i.e., the *Subject* role as shown in Figure 2.4 (a).
- Conceptual operations of interest (operations that trigger an *Observer* update) can crosscut methods in one or more classes. That is, for one conceptual operation there can be n methods, and one method can be in n conceptual operations; i.e., the `subjectChange(...)` operations triggering an *Observer* update as shown in Figure 2.4 (b).
- Roles from multiple patterns can crosscut each other with respect to classes and/or methods. That is, two or more classes that pattern *A* sees as part of one role, pattern *B* may see as in more than one role, and vice versa. The same is true for conceptual operations; i.e., the *Subject* role and `subjectChange(...)` operations as shown in Figure 2.4 (a).

Table 2.1 shows that the types of roles a pattern introduces and the observed benefits of an AspectJ implementation are correlated. The design patterns can be divided into three groups: those with only defining roles, those with both kinds of roles and those with only superimposed roles. The table shows that while the AspectJ implementations of the patterns in first group show no improvements⁶, patterns from the last group show improvements in all modularity benefit categories we identified. For patterns that have both kinds of roles, the results are dependent on the particular pattern.

Given that AspectJ is intended to modularize crosscutting structure, this result should not be surprising. It says that patterns that involve primarily crosscutting structure are well modularized in an AspectJ implementation. (Note that AspectJ does not remove the crosscutting of the pattern, but rather provides mechanisms to modularize that structure.)

⁶Minor improvements as outlined in Section 2.3 are not shown.

Defining and Superimposed Roles

The tight correlation between pattern roles, the crosscutting a pattern introduces, and the observed benefits of an AspectJ implementation suggest that the types of roles used within a pattern indicate whether a pattern implementation is crosscutting or not.

With defining roles, each unit of abstraction (class) represents a single concept, meaning that the functionality of a class corresponds to its role in the pattern. Inheritance is used to distinguish between related but different implementations. In such a case, transparency and pluggability are not useful properties, as each participant is inherently useful only within one particular pattern instance.

With superimposed behavior, the situation is different. Participants have their own responsibilities and justification outside the pattern context. If we force one such class into the pattern context, we have — at the very least — two concerns represented by one module of abstraction (class): the original functionality and the pattern-specific behavior. The resulting tangling and oftentimes code duplication can cause problems as the modularity is compromised. For these patterns and their implementations, a clean modularization of the pattern functionality and the original functionalities of the participants is desirable. In an AspectJ implementation it is usually possible to modularize the abstracted pattern behavior and have one aspect per pattern instance assign roles, conceptual operations, and fill in instance-specific code. Since the participants do have a meaning outside the pattern context, they are not inherently restricted to a single role or even a single pattern instance.

This model appears to be accurate for those GoF patterns that have only defining or only superimposed roles. For others, the expected benefits seem to depend on the number of participants implementing a particular kind of role. Superimposed roles that map to multiple participants, such as *Element* in VISITOR, or *Composite* or *Leaf* in COMPOSITE, indicate potential for modularization, even if the pattern also includes defining roles.

2.5 Summary

Improvements from using AspectJ in pattern implementations are directly correlated to the presence of crosscutting structure in the patterns. This crosscutting structure arises in patterns that superimpose behavior on their participants as indicated by the usage of roles within the pattern. In such patterns the roles can crosscut participant classes, and conceptual operations can crosscut methods (and constructors). Multiple such patterns can also crosscut each other with respect to shared participants.

The improvements manifest themselves as a set of properties related to modularity. The pattern implementations are more localized, and in a number of cases are reusable. Because the AspectJ solutions better align dependencies in the code with dependencies in the solution structure, AspectJ implementations of the patterns are sometimes also composable.

Localizing pattern implementation provides inherent code comprehensibility benefits: the existence of a single named unit of pattern code makes the presence and structure of the pattern more explicit. In addition, it provides an anchor for improved

documentation of the code.

Our results suggest that replacing object-oriented CCC implementations with aspects is beneficial for a set of useful and widely used concerns, due to improved modularity in the AspectJ implementations. Structurally the OO and AO version often vary considerably (as in the OBSERVER case, for example), rendering such transformations non-trivial.

A subset of the OOP/AOP design pattern implementation pairs we developed provides the basis for an initial set of CCC refactorings we investigate. The next chapter introduces our role-based approach to CCC refactorings and illustrate how CCC refactorings, such as the transformation from an OO design pattern implementation to an aspect, are expressed using our role-based concern model.

Chapter 3

Role-Based Refactoring

We now present a role-based approach to refactoring crosscutting concerns (CCCs). Our approach helps a developer transform a scattered implementation of a CCC into an equivalent, but modular, AOP implementation.

In the previous chapter, roles were used to describe the behavior of types⁷ within a design pattern CCC. To model crosscutting concerns and their refactorings, we extend this concept to distinguish between the individual elements contributing to the implementation of a crosscutting concern. We differentiate between role types, role methods and role fields, which we collectively refer to as *role elements*. Role elements can be related to each other. We refer to these relations as *role relations*. Examples include *contains* relations between a role type and role methods, or a *calls* relation between two role methods.

Refactoring instructions are defined in terms of the role elements of the target concern. Together, a concern description and its refactoring instructions comprise a CCC refactoring description.

Role elements and relations abstractly define the structure of the CCC without tying it to a concrete implementation. Thus, the use of roles allows a CCC refactoring to be defined separately from the concrete systems to which it may be applied, allowing the collection and reuse of existing refactoring instructions in a CCC refactoring library. To apply a CCC refactoring, the developer maps role elements to concrete program elements. Then, refactoring instructions are applied, changing the mapped program elements. References to affected program elements, such as calls to changed methods, are updated accordingly, so that the refactoring can affect non-mapped program elements as well.

Our role-based approach is based on the observation that, for a given CCC, certain sets of program elements need to be refactored in the same principal way. For example, in a concrete instance of the OBSERVER pattern we can have multiple types playing the *Subject* role, all of which likely provide implementations for the *attach(Observer)* role method. Each of these concrete methods requires the same treatment in the transformation to an aspect-oriented implementation: being replaced with the *addObserver(...)* aspect method.

We use a concrete example of a CCC refactoring to illustrate the approach. In this example, an instance of the OBSERVER pattern in the open source graphical editing framework JHotDraw [92] is refactored into an aspect. In this particular instance, the class *DrawingView* class acts as a *Subject*, and *FigureSelectionListener* acts as an *Observer*. The structure of the concern implementation and the mapping of role elements is shown in Figure 3.1.

⁷We use the notion of *types* to collectively refer to both interfaces and classes.

Figure 3.1: The partial structure of an object-oriented instance of the OBSERVER pattern in the JHotDraw framework. The mapping of role elements to program elements is indicated using UML stereotypes [73].

3.1 Workflow

Applying a role-based refactoring to a CCC involves four steps:

1. **Selecting a CCC refactoring.** The developer chooses a refactoring from a library of CCC refactorings, or creates a new refactoring and adds it to the library. A CCC refactoring includes an abstract description of the concern it targets, and a set of refactoring instructions to produce a modular AOP implementation. Due to their abstract nature, such refactorings can be applied to multiple instances of the target concern. In the JHotDraw example, the developer chooses the OBSERVER design pattern CCC refactoring.
2. **Stating a mapping.** The role elements in the description of the CCC are mapped to the program elements of the scattered implementation. A CCC refactoring tool can help the developer map role elements. In the example, the developer might map the *Subject* role type to `DrawingView` and the *Observer* role type to `FigureSelectionListener`. Based on these initial mappings, the tool could then suggest further mappings, such as from `update(Subject)` to `selectionChanged(DrawingView)`, that the developer can accept or reject.
3. **Planning the refactoring.** The CCC refactoring tool analyzes the planned refactoring steps as they would apply to the target system to identify choices in the refactoring process, and to provide the developer with associated tradeoffs. For example, the tool would warn the developer if an introduced method is accidentally matched by an existing pointcut. The developer decides how to resolve these cases.
4. **Execution.** Once the refactoring has been planned, the tool transforms the code according to the refactoring instructions, incorporating the decisions made in the planning step. The execution of the refactoring may result in the creation of new program elements, such as a new aspect to contain the modularized OBSERVER pattern, as well as changes to the existing code, such as the replacement of object methods with aspect methods or the removal of obsolete interfaces.

3.2 Describing Crosscutting Concerns

Conceptually, we differentiate between a concrete CCC and its abstract description. A *concrete* CCC is the actual implementation of the concern, while an *abstract* CCC generalizes the structure of concrete CCCs. For example, in the JHotDraw framework, there are multiple cases where the OBSERVER pattern has been used. These are multiple concrete scattered OBSERVER CCCs. All of them have a similar structure, which is captured in the abstract OBSERVER CCC.

Each CCC refactoring in the library includes an abstract description of the concern it targets in terms of its role elements and role relations. Role elements are named to differentiate between program elements of the same kind (types, methods, or fields) that

Table 3.1: Role Elements and their Relationships

Role Element	Possible Relationships with other Role Elements		
	Type	Method	Field
Type	-	contains	contains
Method	returns, hasArgument	calls	-
Field	hasType	-	-

Figure 3.2: Role elements in the OBSERVER CCC. Role types are represented as squares, role methods as rounded squares and role fields as hexagons. The structure reflects the *pull* variant of the pattern [28]

fulfill different functions within the concern, for example between *attach(Observer)* and *notify()*, both of which are role methods in the OBSERVER CCC. Role relations are directed, binary relationships between role elements, such as the *calls* role relation between *notify()* and *update(Subject)*. Table 3.1 depicts relations between kinds of role elements we have found useful in the CCC refactorings we have considered. The kinds of role elements and role relations used in CCC refactorings may need to be extended as more CCCs are considered; for example, role relations describing that types extend types may be needed.

A crosscutting concern description is thus effectively a graph in which role elements are nodes and role relations are edges. For instance, the structure of the Observer CCC is shown in Figure 3.2. Note that the distinction between *Subject* as an interface and *Subject* as a concrete type — as suggested by the GoF design pattern description — is an implementation issue and not a conceptual one; hence it is not reflected in the CCC description.

Given the spectrum of implementation variants for CCCs, an important question is how to deal with potentially varying concern structures, such as known implementation variants for design patterns. Chapter 6 discusses this issue in detail.

3.3 Describing Refactorings

To execute a CCC refactoring, a set of transformation steps needs to be defined that transforms the code base and modularizes the scattered concern. These transformations are captured in the CCC refactoring instructions.

CCC refactoring instructions consist of an ordered set of *refactoring primitives*, which are composite refactorings that are parameterized in terms of role elements. For example, one step in the OBSERVER CCC refactoring instructions consists of applying the *replace object method with aspect method* refactoring primitive to the original *attach(Observer)* methods. This primitive is itself composed of multiple other refactoring primitives, including steps to update call sites to the original method and to adjust import statements of the involved types.

3.4 Mapping Role Elements

Applying a CCC refactoring requires a mapping from the abstract CCC description to program elements in a code base that comprise a concrete CCC implementation. The set of mappings is a partial binary relation associating each role element with zero or more program elements.

The process of mapping role elements to program elements is aided by the approach. Based on initial partial mapping information provided by the developer, a comparison of the structure of the abstract CCC and a static analysis of the type hierarchy and call graph structure of the target software can be used to suggest additional mappings.

For instance, if a developer has mapped the *Subject.attach(Observer)* role method onto *addFigureSelectionListener(...)* on *DrawingView*, a tool can straightforwardly infer a mapping of the *Subject* role type to *DrawingView* based on the *contains* role relation between *Subject* and *attach(Observer)*. Similarly, the tool can suggest mapping the *Observer* role type to *FigureSelectionListener* based on the *hasArgument* relation between *attach(Observer)* and *Observer*.

Figure 3.3 shows the initial and derived mappings. The mapping algorithm can be applied iteratively using derived mappings as the basis for further inferences.

A comparison of the structure in the abstract CCC and the program structure can also reveal potentially incorrect mappings, and suggest alternative mappings that match the abstract CCC structure more closely. For example, it can be detected whether the concrete methods that a role method is mapped to have the wrong argument or enclosing type, or whether the implementation is missing parts of the structure of the CCC. In both cases, the developer can be presented with the current mapping and the mapped elements that do not match the structure of the CCC description. Potential alternatives are evaluated based on the fit of the CCC description and the structure of the mapped elements. Structural information inherent in the CCC description provides an initial ranking (the mapping with the fewest deviations has the best fit; all role element relationships are considered equally important for this purpose). Note that the algorithms for completing and checking partial mappings are not tied to a specific concern, but rather work on any abstract CCCs expressed in terms of role elements and

Figure 3.3: Based on an initial mapping (solid arrow), further mappings can be inferred (dashed arrows)

their relationships.

To further facilitate the mapping process, it is possible to augment the abstract CCC description with non-structural information about the role elements, such as lexical or semantic information, as described in Chapter 4.

In summary, a comparison of the program's AST and the abstract concern structure can aid in finding scattered program elements directly corresponding to role elements. Program elements related to the concern, such as call sites for a mapped method, need not be explicitly identified and are automatically updated when the code is transformed (see Section 3.6).

3.5 Impact Analysis

Impact analyses are static program analyses that can be utilized to assess the impact of a planned change to the code base. Whenever an impact analysis discovers a decision to be made by the programmer, it can initiate a dialog with the developer, presenting the associated information to allow a decision to be reached. Four pieces of information can be presented to the developer: the situation, the alternatives, the associated tradeoffs, and a recommendation. This is analogous to the way refactorings support is provided by current IDEs such as Eclipse, except that the analyses need to consider AOP elements, and the involved tests tend to be more complex as multiple planned refactoring steps have to be considered together.

The aspect-oriented refactoring steps comprising a CCC refactoring can adversely affect other parts of the target system. For example, the dynamic behavior of a system using `instanceof` tests can change if a refactoring introduces additional interfaces via `declare parents` statements. Similarly, a changing or newly created program element can be matched by an existing pointcut, causing associated advice to no longer apply or to apply in the wrong circumstances. A partial list of the implications that changes to an aspect-oriented code base can have is presented in [36].

CCC refactorings add an additional level of complexity as they consist of a series of individual refactoring steps, which can potentially interfere with each other. For example, a refactoring step can modify or delete a program element that is referred to in a subsequent refactoring. Effectively, analyzing a CCC refactoring means analyzing the pre- and postconditions for each refactoring step sequentially, while taking into account how each step affects the target program.

Static analysis can usually conservatively determine whether a proposed transformation of an aspect-oriented system *may* have an adverse effect, but determining the precise effect on the behavior of the system is often difficult [36]. If we take into account that some changes in behavior are acceptable if the intent of the concern is preserved, and that automated analysis techniques fail to capture and test intent, presenting the case as a decision point to the developer is often the only tractable option.

Section 4.3 describes the impact analyses we implemented for our tool.

3.6 Executing Refactorings

Once the refactoring has been planned and all of the associated decision points in its application have been resolved, a supporting tool can perform the code manipulations. The tool follows the refactoring description, performing refactoring instructions on all concrete program elements which are mapped to role elements that the instructions refer to. Other, non-mapped program elements that reference changing portions of the program are updated accordingly.

3.7 Example

To illustrate the kinds of steps involved in refactoring a CCC, we use the example from the beginning of this chapter to describe the steps involved in transforming an object-oriented instance of the OBSERVER pattern in JHotDraw into an aspect-oriented version. The refactoring instructions we use for the example assume a library aspect that abstracts commonalities of the OBSERVER pattern from [38].

Listing 2.1 shows the structure of the abstract library aspect for the OBSERVER case. The library aspect contains most of the pattern functionality, including facilities for the addition and removal of observers per subject, and the notification logic that triggers observer updates when a change of interest occurs. As outlined in Chapter 2, for a concrete instance of the pattern, a developer only needs to subclass this aspect, specify which types act as *Subjects* and *Observers*, concretize the `subjectChange(...)` pointcut to specify what constitutes a change of interest, and override the `update(...)` method to define the update behavior.

The required refactoring steps for the general OBSERVER CCC refactoring are described below. For brevity, instructions for removal of program elements rendered obsolete by the modifications (empty interfaces, unused variables, etc.) are omitted. The aspect-oriented refactorings named are described in Appendix B.

1. Create a new aspect to modularize the CCC and assign it a name selected by the developer, extending the existing `ObserverProtocol` library aspect using

the *create subaspect* refactoring.

2. Add `declare parents` statements to the new aspect to programmatically assign the internal `Subject` and `Observer` interfaces. This is realized using the *add internal interface* refactoring with the role types *Subject* and *Observer* as parameters.
3. Replace the existing methods for adding, removing, and updating observers with the provided aspect methods. The refactoring *replace object method with aspect method* is applied to *attach(Observer)*, *detach(Observer)*, and *update(Subject)*.
4. Replace the update logic by using the *replace method call with pointcut and advice* refactoring applied to the *notify()* role method.

Applying the same refactoring to another instance of the `OBSERVER` CCC would not require changing the refactoring steps, but only the mappings of role elements to program elements. As long as the new instance of the pattern has the required structure, the same CCC refactoring can be applied. Varying names of concrete program elements comprising the concern implementation do not pose a problem due to the abstractness of the underlying role-based concern model.

3.8 Summary

Transforming a scattered implementation of a crosscutting concern into a modular AOP implementation requires many changes to the code. In this chapter we have introduced an approach for refactoring CCCs based on roles. The roles allow the abstract description of the CCC; without this abstraction, each time a similar scattered implementation of a CCC appears in the code for a system, a refactoring would have to be built from scratch. The applicability of a role-based refactoring is determined by the structure of the target concern implementation and not the names of the program elements comprising it.

Creating a concern description and associated refactoring instructions requires expert knowledge of the targeted concern. In Section 6.3 we consider approaches to create these descriptions automatically during program investigation, but this is future work. In the case of design pattern CCC refactorings, the concern structures generally match the design pattern structures as defined in [28], except for the lack of differentiation between interfaces and classes in the role-based model. Developing refactoring instructions is similarly straightforward if the desired aspect-oriented structure is known, as is the case for the investigated design pattern CCCs.

In describing the approach, we focused on a specific kind of CCC refactorings: refactorings of non-modularized crosscutting concerns into aspects. The main application we envision is the stepwise adaption of AOP for an OO system. In order for the approach to express refactorings of existing AOP implementations, the concern model would have to be extended to include aspect-oriented role elements, such as pointcuts and advice, as well as appropriate role relations. Such an extension of the role-based refactoring approach is future work.

In the following two chapters, we will show that the approach can be realized in a refactoring tool and we will use our tool to transform instances of different non-modularized CCCs into aspects to show that the approach is not limited to design pattern CCCs, and that it is viable for refactoring non-trivial code.

Chapter 4

Tool Support for Role-Based Refactoring

We have developed a tool for the role-based refactoring of crosscutting concerns and realized it as a plugin for the Eclipse IDE.

Our tool consists of three main modules: a refactoring engine that is responsible for performing code transformations, the automapper, which aids the developer in completing mappings of role elements to program elements, and the impact analysis module, which provides a series of refactoring checks to identify potential problems with a planned refactoring. Figure 4.1 shows an overview of the architecture of the tool.

The tool relies on different forms of representation of the target program, which are derived from the program source via existing tools, primarily the Eclipse [91] program models. Further, a database of program facts is used by the automapper module and two AspectJ Development Tools (AJDT) [87] program models are used in impact analyses involving aspect-oriented program elements (not shown).

The two Eclipse program models consist of an abstract syntax tree (AST) representation and the Java Development Tools (JDT) model. We mostly rely on the JDT model in our tool as it provides a suitable and easily accessible interface to obtain information about program elements. The AST representation is used for certain searches within the refactoring engine. A JavaDB⁸ database of program facts is employed by the automapper module. It contains information derived from the Eclipse AST that complements the Eclipse models. Solely for the purpose of impact analysis involving aspect-oriented program elements, we use the two program models generated by the AJDT. These two representations are analogous to Eclipse's AST and JDT model, but contain aspect-specific information.

4.1 The Refactoring Engine

The refactoring engine is responsible for planning and executing the CCC refactorings. For a given CCC refactoring, it maintains a representation of the concern structure and the mapping between role elements and program elements. Refactorings are executed — after consultation with the impact analysis module to identify and resolve potential problems with the planned refactoring — by executing the refactoring instructions on the mapped program elements to create the target concern structure. This includes

⁸JavaDB is part of the FEAT tool [72].

Figure 4.1: The CCC refactoring tool architecture. Existing software used is shown in grey

creating aspects, modifying or removing original program elements, and updating references to them.

These changes to the target system are of the form of buffer manipulations to the affected compilation units, in other words text changes to the Java files. We chose the realization as buffer manipulation over AST manipulations because several of the components we use, specifically AJDT and JDT, were in development during the development of our tool and their APIs were not stable. Future versions of our tool could utilize these models better and realize changes as transformations of the target program's AST instead (see Section 6.2.2), which would facilitate the generation of intermediate ASTs (i.e., between individual refactoring steps) and help manage the consistency of changes (see Section 4.1.3).

Individual program transformations steps are analyzed with respect to their impact on the rest of the system to detect potential deviations in system behavior caused by them. Since some behavioral changes are acceptable if the intent of the targeted concern is preserved, these cases are presented to and resolved by the developer, as outlined in Section 4.3.

In a few cases, the developer may have to adjust the created code slightly, for example when an object method is replaced with an aspect method, as this may affect the scoping of statements within the original method. Such cases are realized by providing appropriate in-code suggestions. In the example of the method replacement, the suggestion for the new method body would consist of the body of the original method in commented form. Similarly, post-refactoring review of the generated code is required

```

1  <ConcernDescription name="Observer CCC">
2    <RoleType>
3      <TypeName>Subject </TypeName>
4      <RoleMethod>
5        <MethodName>attach </MethodName>
6        <Argument>
7          <ArgumentName>Observer </ArgumentName>
8        </Argument>
9      </RoleMethod>
10     <RoleMethod>
11       <MethodName>detach </MethodName>
12       <Argument>
13         <ArgumentName>Observer </ArgumentName>
14       </Argument>
15     </RoleMethod>
16     <RoleMethod>
17       <MethodName>notify </MethodName>
18     </RoleMethod>
19     <RoleField>
20       <FieldName>observers </FieldName>
21     </RoleField>
22   </RoleType>
23   <RoleType>
24     <TypeName>Observer </TypeName>
25     <RoleMethod>
26       <MethodName>update </MethodName>
27       <Argument>
28         <ArgumentName>Subject </ArgumentName>
29       </Argument>
30     </RoleMethod>
31   </RoleType>
32 </ConcernDescription>

```

Figure 4.2: Concern structure description of the *Observer* CCC as used by the refactoring tool.

if a refactoring replaces multiple mapped program elements with a single construct, as is the case in the CHAIN OF RESPONSIBILITY CCC refactoring, where the various *handle(Request)* methods are aggregated in two aspect methods. In these cases, the tool suggests for the bodies of the aspect methods the bodies of all of the original handler methods in commented form.

If a post-refactoring review is required, the behavior of the original system is generally not preserved until the developer makes the proper adjustments. These adjustments usually have a granularity of statements rather than types or methods and are aided by code suggestions generated by the refactoring engine.

4.1.1 Representing Crosscutting Concerns

We express concern descriptions using an Extensible Markup Language (XML) [10] representation of the role elements and role relations. A typical description of a concern structure is shown in Figure 4.2. Conceptually, this description results in a structure as shown in Figure 3.2.

4.1.2 Refactoring Primitives

Instructions for CCCs refactorings consist of a series of calls to a library of refactoring primitives. These primitives are aspect-oriented composite refactorings. A complete list of implemented refactoring primitives can be found in Appendix B. All primitives are realized as subtypes of Eclipse's abstract `Refactoring` class. For our purposes, two methods are of interest: `checkInitialConditions(...)` and `createChange()`. The former checks that the supplied parameters (role elements) contain appropriate mappings and performs a refactoring-specific analysis of the targeted code to decide whether the refactoring can be applied as planned. The latter creates a `Change` object that contains the required program transformation steps.

We implemented several refactoring primitives, which can be divided into three categories: creational, transformational, and destructive program transformations.

Creational Primitives

These primitives create aspect-oriented programming constructs such as aspects, pointcuts, and advice. Primitives in this category are parameterized to reflect the wide variety of possible manifestations of the associated programming constructs.

Due to the complexities involved with creating these constructs, the actual implementations of these refactorings are broken down into various specialized cases. For example, the *Create Pointcut* refactoring encompasses cases such as creating a PC matching a set of method calls, creating joint PCs from two existing PCs, and creating PCs capturing executions of all methods in a particular type. The following list is an overview of the creational refactoring primitives.

- *Create Aspect* - creates a new stand-alone aspect to hold the modularized implementation of the target CCC.
- *Create Subaspect* - creates an aspect as a subaspect of an existing abstract library aspect.
- *Create Aspect from Library Aspect* - as above, but copies the library aspect structure and results in a stand-alone aspect.
- *Create Advice* - creates an advice for a given aspect.
- *Create Pointcut* - creates a pointcut for a given aspect.
- *Create Inter-Type Declaration* - creates a method or field ITD.

Aspect creation differs from other refactoring primitives in that it happens before other refactoring steps are even planned. This is vital to ensure that the program elements of the newly created aspect are available in the Eclipse AST and JST program representations. For example, the `OBSERVER` refactoring refers to the `change()` pointcut, which is generated as part of the aspect creation, in subsequent refactoring steps. By creating the aspect before other planning refactoring steps, it can be compiled and appears in the Eclipse program models we rely on.

Transformational Primitives

Refactorings in this category manipulate existing programming constructs. Typical examples include low-level refactorings such as moving members from classes to aspects (replacing them with inter-type declarations) or changing imports of a class or aspect, and high-level transformations, such as replacing an abstract class with an interface and aspect-supplied inter-type declarations.

- *Change Abstract Class to Interface* - replaces an abstract class with an interface and appropriate inter-type declarations.
- *Change Accessibility Modifier* - changes the accessibility modifier (public, private, etc.) of a member.
- *Change Variable Type* - changes the type of a variable.
- *Change Imports* - adjusts the imports of a type.
- *Move Object Member to Aspect* - replaces an object member with an equivalent inter-type declaration.
- *Remove Argument* - removes an argument from a method.
- *Replace Method* - replaces a method with another method and updates the call sites.
- *Replace Object Method with Aspect Method* - replaces an instance method with a (static) aspect method.

Destructive Refactorings

Destructive refactorings remove program elements such as methods, fields and types. They are mainly used to dispose of elements that the refactoring renders obsolete, for example the *observers* field on *Subjects* in the OBSERVER CCC that is not required in the aspect-oriented version of the pattern. These refactorings — as opposed to the ones mentioned before — have rather straightforward implementations.

- *Remove Type* - removes a type from the system.
- *Remove Member* - removes a member.
- *Remove Method Calls* - deletes all call sites to a particular method.
- *Remove Field References* - deletes all references to a given field.

4.1.3 Interference of Refactoring Primitives

All refactoring primitives, except for those that create aspects, are implemented as source code buffer modifications of existing types. The affected regions of the code are determined from querying the affected JDT program elements. Unfortunately, since the program models do not get updated during the refactoring, it is possible that one refactoring step invalidates subsequent refactorings of the same source/buffer. It is conceivable that a future version of the refactoring tool could compute the affected code area dynamically based on previous changes, thereby avoiding these invalidations. Currently however, changes can interfere with each other. This is the case when a refactoring step replaces a region of the buffer with a region of different length, and a subsequent step affects a region in the buffer with a higher offset. It is further possible that one change affects a region of code that lies with the region of another change. For example a *Remove Method* change can target a method that contains a field reference targeted by a *Remove Field Reference* refactoring. Clearly, only the former is necessary. If both of these overlapping changes are applied, the buffer becomes inconsistent.

We address these potential problems by analyzing the proposed changes before they are applied. During the sorting process, the tree-like structure of changes associate with a CCC refactoring is replaced with a flat list of refactoring steps, sorted in descending order by the affected compilation unit and the offset of the affected buffer region. This ensures that for each refactoring step, all subsequent steps affect either an untouched region of the same buffer or a different buffer altogether, eliminating the danger of destructive interference between refactoring steps. In all investigated cases, overlapping changes occur either when statements within a method are changed and the method is marked for deletion, or two or more changes affect the same region in the same way (duplicate changes). These cases can therefore be resolved by removing the contained changes. Figure 4.3 show this process schematically. The composite change is initially unsorted and contains overlapping changes (a). After processing the change, overlapping regions are resolved and the changes are sorted in ascending offset order to avoid inconsistent buffer modifications (b).

If changes were realized as direct AST mutations instead of buffer modifications, overlapping changes are also possible, but easier to resolve. In case of the method and field reference overlap as described above, the two changes would not create inconsistencies; the end result is the program AST without the subtree representing the method, regardless which refactoring is applied first.

4.1.4 Implementation of CCC Refactoring Instructions

Implementing CCC refactoring instructions requires translating the necessary refactoring steps into calls to the refactoring primitives provided by the tool. In essence, every conceptual refactoring step translates into a call to the appropriate parameterized primitive. Consider for example the refactoring of the Observer instance in JHotDraw from the previous chapter. Table 4.1 shows the principal refactoring steps identified in Section 3.7 (left column) and their implementation as a CCC refactoring in our tool (right). The right column shows the entire code for the CCC refactoring. Only consistency checks for potentially invalid mappings are omitted.

Figure 4.3: Pre-processing of buffer modifications

Table 4.1: Conceptual refactoring steps and their implementation (OBSERVER)

Refactoring Step	Implementation
Create new aspect	<code>IType concreteAspect = createSubaspect(newAspectName, findType(abstractAspectName), getTypes("Observer")[0].getPackageFragment(), new String[] {"Subject", "Observer"}, new String[] {"Subject", "Observer"});</code>
Specify role names for internal interfaces Specify corresponding internal interface names	<code>addChange(ReplaceObjectMethodsWithAspectMethods.create(getMethods("attach"), findMethod(concreteAspect, "addObserver"), concreteAspect)); addChange(ReplaceObjectMethodsWithAspectMethods.create(getMethods("detach"), findMethod(concreteAspect, "removeObserver"), concreteAspect)); addChange(SuggestMethodBody.create(getMethod("update"), findMethod(concreteAspect, "updateObserver")));</code>
Replace Object Methods with Aspect Methods	<code>addChange(CreatePointcutRefactoring.create(findCallsToMethods(getMethods("notify"), findMethodInType(concreteAspect, "subjectChange"), new IType[] findType("Subject"), new String[] {"s", null, null}));</code>
Create pointcut capturing calls to <i>notify()</i>	<code>addChange(RemoveMethodCallsRefactoring.create(getMethods("notify"))); addChange(MethodDeletionChange.create(getMethods("notify")));</code>
Remove <i>notify()</i> and calls to it	

Implementations of the CCC refactorings we investigated can be found in the appendix. We realized a total of 11 GoF design pattern (see Section 4.5) refactorings and 4 non-pattern refactorings (see Section 5.3). Our CCC refactoring instructions, including the instructions for non-pattern concerns (Chapter 5), range from three (EXTRACT INTERFACE, SINGLETON) to eight (WORMHOLE) calls to the tool's refactoring primitives, with an average of 5.4 calls.

4.1.5 Representing Aspect-Oriented Program Constructs

The refactoring engine relies in large parts on the Eclipse program models: the AST and JDT. Unfortunately, neither of them includes facilities to express aspect-oriented programming constructs. Since we did not want to restrict the tool to pure OO code, we needed to create a means to include these constructs in the AST and JDT models⁹.

We achieved this by *masking* all aspect-oriented program elements at the beginning of a refactoring using comments. After the refactoring is completed, these masked sections are translated back into aspect-oriented code. For example, the abstract change (. .) pointcut of the reusable aspect-oriented OBSERVER implementation is represented as follows:

```
// <ASPECT>protected abstract pointcut subjectChange(Subject s);</ASPECT>
// <CLASS>
protected abstract void subjectChange(Subject s);
// </CLASS>
```

The <CLASS> portion provides an object-oriented stand-in for the pointcut. The Eclipse parser parses it as a method and provides an appropriate representation to which the refactoring instructions can refer. After the refactoring, this section is replaced with the <ASPECT> portion:

```
protected abstract pointcut subjectChange(Subject s);
```

Masking always produces a valid Java program. This approach, while inelegant, works for the investigated refactorings. We expect that, as AJDT matures and opens up the interface for accessing its program representations, future implementations of our role-based refactoring approach will be able to use the AJDT exclusively. This would alleviate the need for masking AOP constructs.

4.2 Auto-Mapping Support

The AutoMapper module facilitates the mapping of role elements onto concrete program elements, according to the algorithm outlined in Section 3.4. Based on one or more initial mappings provided by the developer, the automapper attempts to match

⁹An alternative we considered was using AJDT program models exclusively instead of the Eclipse AST/JDT for our refactoring engine. This turned out to be problematic, due to the constant flux of their internal implementation and, at least in the beginning of the development of our tool, their incompleteness.

the CCC structure with the structure of the target program. The program structure used comprises hierarchical relationships derived from the ASTs and method call relationships as provided by the JavaDB program facts database. This structural analysis generates candidate program elements for unmapped roles.

Ambiguities can arise when multiple concrete program elements qualify for a mapping to a single role element. We will illustrate this on the example of the *OBSERVER* pattern instance. Assuming that an initial mapping exists for the role types *Subject* and *Observer*, we will consider the case of deriving mapping candidates for the role method *attach(Observer)* on *Subject*. Based on structural information alone, the algorithm will suggest all program elements $p \in P$ that fulfill the following requirements:

1. p is a method
2. p is a member of the type(s) *Subject*
3. p has an argument that matches the type *Observer*

In this case, two role elements have the same set of associated requirements: the *attach(Observer)* and *detach(Observer)* role methods on *Subject* are identical from the perspective of structural automapping. Consequently, the concrete program elements implementing the two methods will both appear as candidates for either role method mappings.

To rank multiple candidates for a single role, we use lexical information about the role elements stored in the abstract CCC descriptions. Lexical information constitutes a set of name fragments (currently substrings) that are likely to be used for concrete program elements playing a particular role. For example, it is more likely that a concrete implementation of the role method *attach(Observer)* has a name containing either “add” or “attach”, than “remove”, or “detach”. Some of the lexical rules used to automap this CCC are shown in Table 4.2. The weights of each matching rule are summed to determine the score of the mapping candidate. We assigned the weights based on our subjective expectations of the likelihood that the rule would be indicative. Table 4.2 shows weights for three role elements.

The auto-mapping process is iterative, so if the developer accepts one or more suggestions, subsequent steps take these new mappings into account.

Other kinds of support for automapping are conceivable, we will address two promising ideas as future work in Section 6.2.1.

4.3 Support for Impact Analyses

The Impact Analysis (IA) module aims to identify the effect of a planned refactoring on the rest of the system, and to resolve potential problems with the developer. The module generates appropriate dialogues with the developer when it (conservatively) determines that a planned change may affect the behavior of the target system. Resolving such potential problems can affect the refactoring workflow, usually by undoing a transformation step or aborting the refactoring if inconsistencies are detected.

Table 4.2: Sample weights for name fragments for role elements

Role Element	Name Fragment	Weight
<i>attach(Observer)</i>	"add"	2
<i>attach(Observer)</i>	"attach"	3
<i>attach(Observer)</i>	"remove"	-2
<i>attach(Observer)</i>	"detach"	-3
<i>attach(Observer)</i>	"observer"	1
<i>attach(Observer)</i>	"listener"	1
<i>notify()</i>	"notify"	3
<i>notify()</i>	"change"	1
<i>notify()</i>	"observer"	1
<i>notify()</i>	"listener"	1
<i>Observer</i>	"observer"	3
<i>Observer</i>	"listener"	3

To illustrate the concept of impact analyses, we implemented a number of sample analyses. It was not our goal to provide an exhaustive set of analyses for our tool. Instead, we aimed to explore how impact analyses fit into the CCC refactoring workflow. We differentiate between pre-condition checks that are specific to the individual refactorings (primitives or CCC refactorings) and global checks that can apply to multiple program transformations.

Pre-condition checks operate on the program elements passed to the refactoring object and ensure that the refactoring can be properly executed. If a pre-condition is not met, the refactoring will either not be executed or will require developer confirmation, depending on the severity of the problem. For example, in the *move object member to aspect* refactoring, static object members can not be replaced with an intertype declaration, and appropriate enabling refactorings are non-obvious, so the refactoring fails. If the chosen member was protected instead, the tool would offer the developer the choice to change the accessibility of the member (and related members in the inheritance hierarchy) to public. This would allow the refactoring to proceed, but as it affects the semantics of the program, developer confirmation is required. Note that in the case of composite refactorings, pre-condition checks can be redundant¹⁰.

Global impact analyses apply to multiple refactorings. These analyses are not part of the individual refactoring descriptions but are instead realized separately as aspects to avoid a scattered implementation, since the analyses need to collect information about the system and planned refactoring from various places in the workflow. One example for an impact analysis in this category determines if a method is manipulated in a way that might affect pointcuts that refer to it. Should this be the case, the transformation can affect the system's behavior and the developer can choose to abort the refactoring, or proceed if it does not affect the intent of the original implementation. The required information about pointcut matches is extracted from the AJDT program models. The implementation as aspects has the further benefit that costly impact anal-

¹⁰Kniesel presents a possible approach that could be utilized to address the redundancy of precondition checks of composite refactorings [47]

Figure 4.4: Structure of the OBSERVER CCC as presented by the refactoring tool.

yses can be easily removed from or added to the tool as desired.

We do currently not have a universal analysis framework that provides implementations of complex program analyses, such as those affecting the type structure (e.g., see [81]). The development of such a framework and additional impact analyses, in particular dependency analyses for individual refactoring steps, such as through *Conditional Transformations* [47], are future work.

4.4 Example

In the following, we illustrate the operation of our role-based refactoring tool by using it to refactor an instance of the OBSERVER pattern in the figure package described in 2.3.1. We assume that the developer using the tool is aware that changes to `FigureElements` trigger updates of the `Display`, and accordingly chooses the OBSERVER CCC refactoring. The OBSERVER CCC structure is shown in Figure 3.2.

4.4.1 Mapping

The tool presents a tree view of the role elements comprising the OBSERVER concern. The hierarchy reflects the *contains* relationships between role elements; other relationships, such as *calls* or *hasArgument* are not shown. Figure 4.4 depicts the structure of the OBSERVER concern as presented to the developer.

The developer provides an initial mapping of *Subject* to `FigureElement` and *Observer* to `Display`, based on his knowledge of the system. Mappings are realized by selecting a role element and assigning a concrete program element to it. Figure 4.5 illustrates this for the *Observer* $\mapsto$ `Display` mapping. Proposed mappings are checked by the tool to ensure mapped program elements are of the proper kind (i.e., a role type can only be mapped to interfaces and classes, not methods and fields).

After providing these two initial mappings, the automapping facilities of the refactoring tool are used to complete the mappings. First, subtypes of `FigureElement` are trivially identified as further candidates for *Subject*, and mapped manually by

Figure 4.5: Mapping the *Observer* role to the *Display* interface.

the developer. In the next automapping iteration initiated by the developer, candidates for the role methods and role fields are identified. For example, to determine candidates for the *attach(Observer)* method, the automapper identifies two methods with matching signature: the methods *addChangeObserver(...)* and *removeChangeObserver(...)* (both declared in the *FigureElement* interface, and defined in *Point* and *Line*) match the structural analysis. The automapper differentiates them based on the lexical analysis outlined in section 4.2. The analysis for the *attach(Observer)* mapping to methods on *Point*, for example, is shown in Figure 4.6.

The automapping facilities correctly ranks candidate method *addChangeObserver(...)* highest, and the developer selects the appropriate mapping by browsing to the suggested code element and mapping it manually. Mappings for *attach(Observer)* to methods on *FigureElement* are resolved the same way.

Candidates for *detach(Observer)*, *notify()* and *update(Subject)* are identified in a similar fashion, but for each of these only one candidate method exists, so a lexical analysis is not required. The *observers* role field is not identified automatically as the automapping facilities do not currently handle aggregate types. The completed mappings are shown in Figure 4.7.

4.4.2 Impact Analysis

To illustrate the concept of impact analysis, we use the example of an impact analysis warning the developer when a method scheduled for deletion is advised by an existing aspect in the system. To show this, we created an aspect that advises the *Point.addChangeObserver(...)* method. Since all methods *attach(Observer)* is mapped to are deleted in the course of an *OBSERVER* refactoring, the appropriate impact analysis produces a warning before the refactoring is executed, as shown in

```

1 Processing role element: attach(Observer): void
2 >>> Candidates after structural inference:
3   Candidate for attach: removeChangeObserver (lexical: 0)
4   Candidate for attach: addChangeObserver (lexical: 0)
5     Candidate is removeChangeObserver
6       Testing 6 lexical rules...
7       Contains: "remove", new score is -2
8       Contains: "observer", new score is -1
9     Candidate is addChangeObserver
10      Testing 6 lexical rules...
11      Contains: "add", new score is 2
12      Contains: "observer", new score is 3
13 >>> Candidates after lexical and semantic inference:
14   Candidate for attach: removeChangeObserver (lexical: -1)
15   Candidate for attach: addChangeObserver (lexical: 3)

```

Figure 4.6: After mapping *Subject* to *Point*, the automapper determines candidates for mapping the *attach(Observer)* role method based on a structural and lexical analysis of methods of the *Point* class.

Figure 4.7: The completed mappings for the OBSERVER CCC instance in the figure package example.

Figure 4.8: An impact analysis detects that an existing aspect is advising a changing program element.

Figure 4.8.

The developer can choose to abort the refactoring or proceed with its execution. In this particular case, we decide to proceed and ignore the impact on the behavior of the system.

4.4.3 Executing the Changes

The refactoring creates the aspect capturing the OBSERVER concern from a library aspect (see [38]) and performs the required changes to the original concern implementation.

If the developer wishes to review the changes, Eclipse's undo support can be utilized to review changes for each individual affected compilation unit.

4.4.4 Remarks

The purpose of our proof-of-concept tool is to show the feasibility of semi-automated support for CCC refactorings. User studies will be needed to identify how to structure the dialogues between tool and developer more efficiently. Some of these interactions have alternative realizations. As an example, an earlier version of our tool mapped program elements identified by the impact analysis automatically instead of involving the developer explicitly. We found that performing the mappings per hand according to the automapping suggestions helps the developer to understand the targeted concern implementation, but future versions of the tool may offer support for either or both automapping realizations.

Similarly, future versions can conceivably offer more options for the developer when dealing with the results of an impact analysis. Instead of just aborting or proceeding with the refactoring, the tool could offer the option to browse the affected code, or to mark it for later review.

4.5 Tool Calibration

To derive a set of useful refactoring primitives, we developed seven principal CCC refactorings for 11 of the design pattern CCCs identified in Chapter 2:

- Chain of Responsibility
- Composite
- Mediator
- Memento
- Observer
- Singleton
- Template Method, Abstract Factory, Factory Method, Builder, Bridge¹¹

The design patterns were chosen based on an estimation of the frequency of their usage in real-world software (Observer, Composite, Singleton, Abstract Factory, Factory Method, Template Method)¹² and on the quality of their aspect-oriented implementations (Chain of Responsibility, Composite, Mediator, Memento, Observer, Singleton) according to [38]. Note that some patterns fall into both categories. The similarity of the refactoring processes also led to the inclusion of Builder and Bridge.

The tool properly performed the CCC refactorings for all 11 CCC refactoring examples we investigated (one for each design pattern refactoring implemented; the systems are from our previous work [38]) and the automapping feature successfully identified correct candidate mappings given a seed mapping in all cases. Chapter 5 includes an analysis of how the calibrated tool performed during the refactoring of design pattern CCCs in a non-trivial software system.

4.6 Extending the Framework

Our tool provides a framework for defining and executing role-based refactorings. To calibrate and evaluate the approach, we created a library of CCC refactorings, most of which are design pattern refactorings. If developers need to create additional refactoring descriptions within this framework, they need to provide the following information:

1. **Concern description:** this is a description of the concern structure consisting of role elements and role relations in XML format similar to Listing 4.2. Optionally, this structural information can be augmented by lexical information as discussed in Section 4.2.

¹¹These design pattern CCCs required virtually the same refactoring instructions. While their AO implementations naturally differ, the same general ideas for the transformation of the concern structures apply for all of them [38].

¹²According to personal communication with John Vlissides and a survey on the Design Patterns Discussion mailing list at the University of Illinois at Urbana-Champaign: <http://lists.cs.uiuc.edu/mailman/listinfo/patterns-discussion>

```

1  <ConcernDescription name="Logging CCC">
2      <RoleType>
3          <TypeName>Logger</TypeName>
4          <RoleMethod>
5              <MethodName>logEntryMethod</MethodName>
6          </RoleMethod>
7          <RoleMethod>
8              <MethodName>logExitMethod</MethodName>
9          </RoleMethod>
10     </RoleType>
11     <RoleField>
12         <FieldName>logger</FieldName>
13         <FieldType>Logger</FieldType>
14     </RoleField>
15 </ConcernDescription>

```

Figure 4.9: Concern structure description of a *Logging CCC*.

2. **Refactoring instructions:** this involves creating a new subclass of our framework's extension point `CCCRefactoring`. This abstract class provides default implementations for CCC refactorings, such as facilities for accessing concrete program elements that a particular role element is mapped to. Developers need to overwrite its `createChange(..)` method. Within that method, library refactoring primitives are called.
3. **New elementary refactorings:** if the CCC refactoring requires program transformation steps not captured by existing elementary refactorings, the developer can create these by extending Eclipse's abstract class `Refactoring`.

As a concrete example, consider adding a refactoring for a simple logging CCC, which is described in detail in Section 5.3.1. First, we define the CCC structure, which consists of a *Logger* role type with two methods *logEntryMethod(..)* and *logExitMethod(..)* that are called by clients. Clients have a role field of type *Logger*. The concern definition is shown in Figure 4.9. Optional lexical information could define "log" as a likely name fragment for all role elements and additionally "before" and "start" for *logEntryMethod(..)* and "after" and "end" for *logExitMethod(..)*.

The next step is to define the refactoring instructions. This involves subclassing `CCCRefactoring` and defining the refactoring instructions in terms of refactoring primitives referring to role elements. In this particular case, we need the following refactoring primitives:

1. Create Aspect (to capture the modularized implementation)
2. Create Pointcut *P1* (based on call sites of *logEntryMethod(..)*)
3. Create Pointcut *P2* (based on call sites of *logExitMethod(..)*)
4. Create Advice (stating that before *P1*, *logEntryMethod()* is called)
5. Create Advice (stating that after *P2*, *logExitMethod()* is called)
6. Remove Method Calls (removing calls to methods to which *logEntryMethod()* is mapped)

```

1  public Change createChange(..) throws CoreException, ... {
2
3      // create empty aspect with two masked PCs
4      IType aspect = AspectFactory.create(
5          "MyLoggingAspect",
6          getType("Logger").getPackageFragment(),
7          new String[] {"methodEntry()", "methodExit()"}); // PC placeholder
8
9      addChange(CreatePointcutRefactoring.create(
10         findCallsToMethods(getMethods("logEntryMethod")),
11         getPC(aspect, "methodEntry"),
12         new IType[] {},
13         new String[] {},
14         null,
15         null));
16
17      addChange(CreatePointcutRefactoring.create(
18         findCallsToMethods(getMethods("logExitMethod")),
19         getPC(aspect, "methodExit"),
20         new IType[] {},
21         new String[] {},
22         null,
23         null));
24
25      addChange(CreateAdviceRefactoring.create(aspect,
26         "before",
27         getPC(aspect, "methodEntry"),
28         getMethodCallStatements(getMethods("logEntryMethod")), true));
29
30      addChange(CreateAdviceRefactoring.create(aspect,
31         "after",
32         getPC(aspect, "methodExit"),
33         getMethodCallStatements(getMethods("logExitMethod")), true));
34
35      addChange(RemoveMethodCallsRefactoring.create(
36         getMethods("logEntryMethod")));
37
38      addChange(RemoveMethodCallsRefactoring.create(
39         getMethods("logExitMethod")));
40
41      return change;
42  }

```

Figure 4.10: Refactoring instructions for the *Logging* CCC.

7. Remove Method Calls (removing calls to methods to which *logExitMethod()* is mapped)
8. Remove Member (removing the fields to which *logger* is mapped)

The resulting CCC refactoring instructions are shown in Figure 4.10.

No new elementary refactorings were needed. The creation of this particular refactoring took less than 30 minutes.

Chapter 5 includes a study providing additional examples of extending our framework with new CCC refactorings.

4.7 Summary

In this chapter we presented a proof-of-concept implementation of our role-based approach in the form of a refactoring tool for the Eclipse IDE. The tool supports the semi-automation of CCC refactorings.

In the following chapter, we will evaluate the approach and tool by applying the tool to refactor CCC instances in a non-trivial software system, and by extending the refactoring framework to capture additional, non-design-pattern CCC refactorings.

Chapter 5

Evaluation

This chapter describes two case studies we performed to evaluate our refactoring approach. The studies show that the role-based concern model enables the semi-automation of the refactoring of non-modularized crosscutting concerns.

In the first study, we apply our approach and tool to transform CCCs in an existing, non-trivial software system. This study was designed to show that role-based refactoring descriptions may be applied in systems built separately from the CCC descriptions. For the second study we created and applied refactoring descriptions for a number of additional CCCs, showing that our approach can be used to describe and execute refactorings of non-design-pattern concerns.

5.1 Methodology

In both studies, we used our approach and tool to refactor CCCs and tested the transformed systems. The tests focused on two criteria: the resulting code had to compile and we tested the code to see if the intent of the target concern was preserved even if the precise behaviour was not preserved. For the first study, the latter involved extensive tests of the affected code. The original intent was derived from the available documentation. In the second study, all investigated systems contained executable tests of the essential CCC functionality that served as a basis for evaluating the preservation of intent.

In addition to testing the behavior of the resulting system, we also investigated a number of statistics on the transformation to aid in the discussion of the approach. We studied the completeness of the automapping results and investigated the quality of the resulting code.

All refactorings were performed on a Pentium3-700 with 1GB RAM running Eclipse release 3.0.2 under Windows 2000. The tool was used within an Eclipse run-time workbench. Code statistics were collected using the Eclipse Metrics plugin [93] version 1.3.5.

5.2 Study 1: Design Pattern Implementations in JHotDraw

Our first study addresses the questions of whether the approach and tool can be applied to a non-trivial system, and whether a library of CCC descriptions can be applied to a

system that we did not design ourselves. We used JHotDraw [92] release 5.3 as the target system for this study. JHotDraw is a graphical editing framework that encompasses over 15,000 lines of non-commented, non-empty lines of source code in 243 types. The system makes aggressive use of design patterns, predominantly GoF patterns [28]. Originally conceived as a design exercise, it has since then become an open-source framework that is utilized in a wide variety of software projects.

This study also allowed us to investigate the impact of *concern variance*; since design patterns represent only outlines for implementation, actual instances can vary considerably. Because our role-based approach relies on a structural description of the target CCC, such variant implementations can affect its applicability.

JHotDraw was not designed around a single pattern as are the design pattern examples used to calibrate our tool, but rather contains a number of tangled pattern instances. As the usage of design patterns is well-documented in source code comments, we identified individual design pattern instances using textual searches over the sources for parts of the GoF pattern names.

Once an instance was located, we had to manually explore the code base to determine the exact pattern extent (i.e., all involved code fragments), pattern variance (deviations from the pattern definition as represented in the GoF book [28]), and dependencies of the pattern code elements on the rest of the system. This exploration was essential for determining the correctness both of the mappings suggested by the refactoring tool's automapping feature and the actual code transformations.

5.2.1 Results

We successfully refactored a total of 7 instances of 4 different GoF design patterns in JHotDraw.

OBSERVER

Two instances of the OBSERVER pattern were refactored in the JHotDraw sources. A third could not be transformed due to structural deviations of the pattern instance from the expected concern structure, as discussed in Sections 5.4 and 6.2.1. The structure for the OBSERVER CCC refactoring is shown in Figure 3.2.

OBSERVER Instance 1: DrawingView — FigureSelectionListener This pattern instance monitors the set of selected figure elements. Changes to this set occur by selecting additional figures or removing a figure from the selection, and trigger the update mechanism.

This OBSERVER instance deviates subtly from the GoF pattern description and consequently from the concern structure we expected. Unlike in the GoF pattern, the *Subject* DrawingView does not define an equivalent of *notify()*, but leaves that to its implementing types. Only a few subtypes of the *Observer* FigureSelectionListener explicitly implement an *update(Subject)*, most inherit this implementation from a supertype. These deviations did not influence the mapping or refactoring of the concern.

Table 5.1: Role mappings for the DrawingView — FigureSelectionListener instance of the OBSERVER pattern

Role Element	Mapped Program Element
Subject	DrawingView NullDrawingView StandardDrawingView
Observer	FigureSelectionListener AbstractCommand + 18 subtypes of Abstract Command DrawingEditor + 12 subtypes of DrawingEditor
attach(Observer)	DrawingView.addFigureSelectionListener(..) NullDrawingView.addFigureSelectionListener(..) StandardDrawingView.addFigureSelectionListener(..)
detach(Observer)	DrawingView.removeFigureSelectionListener(..) NullDrawingView.removeFigureSelectionListener(..) StandardDrawingView.removeFigureSelectionListener(..)
notify()	StandardDrawingView.fireSelectionChanged()
update(Subject)	FigureSelectionListener.figureSelectionChanged(..) AbstractCommand.figureSelectionChanged(..) UndoableCommand.figureSelectionChanged(..) DrawingEditor.figureSelectionChanged(..) DrawApplet.figureSelectionChanged(..) DrawApplication.figureSelectionChanged(..) JavaDrawViewer.figureSelectionChanged(..)
observers	StandardDrawingView.fSelectionListeners

We derived almost the entire mapping from role elements to program elements using our tool's automapping facilities, based on an initial mapping of `DrawingView` to *Subject* and `FigureSelectionListener` to *Observer*. The automapper suggested the subtypes of these two types for appropriate mappings to their respective roles. The mappings to the role methods *attach(Observer)*, *detach(Observer)*, and *update(Subject)* were identified based on their signatures, and *attach(Observer)* and *detach(Observer)* were properly differentiated based on a lexical analysis (see Section 3.4). *notify()* could be mapped based on its "calls" (*update(Subject)*) relation. The only role-element not mapped by the automapper was the *observers* field, since the automapper does currently not handle aggregate types.

The final mappings are shown in Table 5.1. Note that most types acting as *OBSERVER* inherit the pattern-related methods and field. Therefore, they do not need to change¹³. Strictly speaking, for that same reason they do not even need to be mapped.

¹³They can, however, be indirectly affected by the refactoring, for example when calls to the original *notify()* methods are updated in the course of the refactoring.

Despite the structural differences from the GoF pattern description and the considerable extent of the affected code elements (see Table 5.1) the refactoring was applied without any change to the concern description. The transformation preserved both intent and behavior of the original implementation.

OBSERVER Instance 2: Drawing — DrawingChangeListener In this instance, the OBSERVER pattern is used to decouple Drawing from its views and to enable multiple views, which implement the DrawingChangeListener interface.

The structure of this instance of the pattern deviates noticeably from the GoF pattern description. In this case, the parameter for the *update(Subject)* methods is not of type *Subject*, but a specialized event object. The pattern instance further features multiple (two) *notify()/update()* pairs. This represents a particular variant realization of the *push* model [28] of the OBSERVER pattern. While these discrepancies prevent applying our refactoring description to this instance of the pattern as is, we decided to investigate how much effort is involved in making this implementation work with our CCC refactoring.

In this case, we required an enabling refactoring¹⁴ that reduced the number of *notify()/update()* pairs to one. This enabling refactoring was contained to two types and comprised creating a method *update(Drawing)* in *DrawingChangeListener*, and attaching a dispatch logic to one of the two original update methods via an inter-type declaration. Further, an appropriate method *notify()* in *StandardDrawing*¹⁵ was created. The original two *notify()* methods were removed and their call sites adjusted to call the new *notify()* instead.

Passing an argument of type *Drawing* to the new *update(Subject)* method had the additional benefit that the pattern instance now matched that part of the expected concern structure exactly. Based on an initial mapping of *Drawing* to *Subject* and *DrawingChangeListener* to *Observer*, the automapper correctly mapped *attach(Observer)*, *detach(Observer)* and *update(Subject)* based on their signature and then *notify()* based on the call relationship. Again, the mapping of the *observers* field had to be done by hand as in the previous case. The final mappings are shown in Table 5.2. Program elements with a star (*) are the result of the enabling refactoring.

After the mappings were complete, the CCC refactoring could be applied without affecting the behavior or intent of the original implementation.

SINGLETON

The SINGLETON design pattern has a very simple structure, which is shown in Figure 5.1. We identified two instances of the Singleton pattern in JHotDraw.

SINGLETON Instance 1: Clipboard The *Clipboard* class represents a global clipboard implementation and can be used to store and retrieve application-specific

¹⁴An enabling refactoring is an OO or AO refactoring intended to transform a system so that it matches the structure a CCC refactoring expects.

¹⁵This type provides default implementations for all subtypes of *Drawing*.

Table 5.2: Role mappings for the Drawing — DrawingChangeListener instance of the OBSERVER pattern

Role Element	Mapped Program Element
Subject	Drawing StandardDrawing BouncingDrawing
Observer	DrawingChangeListener DrawingView StandardDrawingView NullDrawingView
attach(Observer)	Drawing.addDrawingChangeListener(..) StandardDrawing.addDrawingChangeListener(..)
attach(Observer)	Drawing.removeDrawingChangeListener(..) StandardDrawing.removeDrawingChangeListener(..)
notify()	StandardDrawing.notify()*
update(Subject)	DrawingChangeListener.update(Drawing) *
observers	StandardDrawing.flListeners

Figure 5.1: Concern structure of the SINGLETON CCC

Table 5.3: Role mappings for the Clipboard instance of the SINGLETON pattern

Role Element	Mapped Program Element
Singleton	Clipboard
instance()	Clipboard.getClipboard()
uniqueInstance	Clipboard.fgClipboard

Table 5.4: Role mappings for the Iconkit instance of the SINGLETON pattern

Role Element	Mapped Program Element
Singleton	Iconkit
instance()	Iconkit.instance()
uniqueInstance	Iconkit.fgIconkit

content. It represents a straightforward instance of the SINGLETON pattern, with a private constructor and public class method `getClipboard()` that provides access to the *Singleton* instance.

The automapper could suggest proper mappings for all role elements given a single starting mapping — regardless which initial mapping the developer provides. Based on the “contains” relation, the role type can be mapped given either the field or the method mapping. Starting with the type mapping, mappings for both role field and method can be properly identified based on its type and signature, respectively. The final mappings are shown in Table 5.3.

The refactoring preserved both behavior and intent of the original implementation.

SINGLETON Instance 2: Iconkit From the JavaDoc comments of the *Iconkit* class: “*The Iconkit class supports the sharing of images. It maintains a map of image names and their corresponding images*”. Interestingly, *Iconkit* is not a proper *Singleton* according to GoF. Aside from in-code comments, there is no indication that this class’ constructor is not to be accessed, and the customary constructor protection (usually using a `private` modifier) is missing. It does, however, have a static *instance()* method and *uniqueInstance* field.

The role mapping was straightforward as in the previous example. From any starting mapping, the other two mappings were properly identified. The complete mappings can be found in Table 5.4.

Unlike in the *Clipboard* case, the constructor for *Iconkit* is parameterized. Since *instance()* methods generally do not have parameters, our SINGLETON CCC refactoring provides default parameters when replacing of calls to *instance()* with constructor calls. For example, `null` is passed for any *Object* or array parameter. Since the aspect-oriented implementation of the pattern circumvents subsequent executions of the constructor after the very first one, this preserves the original behavior of the implementation. Aesthetic issues like this can be addressed by a post-refactoring code

Figure 5.2: Concern structure of the EXTRACT CONCERN IMPLEMENTATION CCC, applied to an instance of the COMPOSITE pattern

review if desired.

COMPOSITE

Although the COMPOSITE pattern is among the more popular GoF patterns (see Section 4.5), has a desirable AOP implementation, and could be refactored for the design pattern examples case, the JHotDraw instance of the pattern is not suitable for our COMPOSITE CCC refactoring as is. Instead, we modularized this pattern instance using a different CCC refactoring.

COMPOSITE Instance 1: CompositeFigure The CompositeFigure class is a relatively complex type (258 LOC, 47 methods) that provides a variety of default implementations for composite graphical elements. This instance of the COMPOSITE pattern deviates considerably from the GoF structure. Most importantly, the concern is, unlike what the GoF pattern suggests, modularized within a single type. The COMPOSITE concern is tangled with three other concerns: administration of ordering information for children figures (Z values), basic update functionality for FigureChangeListener, and facilities for the implementation of quad trees for figures (used to optimize rendering).

To alleviate the problem of tangling present in CompositeFigure a refactoring is applied that moves members (fields and methods) pertaining to the composition concern (or, alternatively, the other three) to an aspect using inter-type declarations. This is accomplished by the refactoring primitive *move object member to aspect*. We create a simple CCC structure, consisting of a role type, a role method and a role field, as shown in Figure 5.2. The transformation requires only three calls to the refactoring primitives: one to create the aspect, one to move the methods, and one to move the fields.

Our structure-based automapping approach is not designed to suggest mappings based on semantic information and could therefore not differentiate between members that belong to the concern and those that do not. The final mapping is shown in Table 5.5.

The refactoring was executed successfully, effectively extracting the tangled implementation of this COMPOSITE instance into an aspect. The creation of this CCC

Table 5.5: Role mappings for the CompositeFigure instance of the COMPOSITE pattern

Role Element	Mapped Program Element
TargetType	CompositeFigure
concernMethod()	CompositeFigure.add(..) CompositeFigure.remove(..) CompositeFigure.replace(..) CompositeFigure.draw(..) + 26 other methods on CompositeFigure
concernField	CompositeFigure.fFigures

Figure 5.3: Concern structure of the TEMPLATE METHOD CCC

refactoring took less than 20 minutes.

TEMPLATE METHOD

Both documented instances of the TEMPLATE METHOD pattern in JHotDraw were investigated. The concern structure for this refactoring is shown in Figure 5.3.

TEMPLATE METHOD Instance 1: AbstractFigure The abstract class `AbstractFigure` provides default implementations for the `Figure` interface and contains a number of template methods.

Although the pattern instance adheres to the (simple) GoF pattern structure, the automapper was not helpful in determining the mappings for the refactoring. Currently our concern description does not include information about role methods indicating whether they are `abstract` or not. This makes it impossible for the automapper to suggest the proper methods. An extension of the role-based concern model to include this information would be helpful. On the other hand, based on any mapping for `templateMethod()`, the correct role type `AbstractClass` can be trivially deduced. The complete mapping for this refactoring is shown in Table 5.6.

The transformation was not straightforward, due to limitations of AspectJ. Since AspectJ does not allow the inter-type declaration of constructors or static inter-type declarations on interfaces, the transformation of members with these properties can

Table 5.6: Role mappings for the AbstractFigure instance of the TEMPLATE METHOD pattern

Role Element	Mapped Program Element
AbstractClass	AbstractFigure
ConcreteClass	CompositeFigure DecoratorFigure AttributeFigure PolyLineFigure
templateMethod()	AbstractFigure.moveBy(..) AbstractFigure.displayBox(..) AbstractFigure.figures(..) AbstractFigure.size(..) + 28 other non-abstract methods on AbstractFigure
requiredField	AbstractFigure.flListener AbstractFigure.serialVersionUID AbstractFigure.abstractFigureSerializedDataVersion AbstractFigure.int_nZ

not be directly realized. To accommodate for this, we had to perform a minor enabling refactoring and remove the (unused and empty) constructor. Further limitations of inter-type declarations in AspectJ affect protected and private members. The tool handles these cases by automatically adjusting the accessibility modifier of affected types to package protected. Since this can affect the intent of the original implementation, the impact analysis notifies the developer of these changes. In this particular case, three private fields and the protected `moveBy(..)` method (and its implementation in 11 subtypes) were affected by this change in accessibility. Overall the refactoring affected 32 methods and 4 fields on the `AbstractFigure` class and its 4 immediate subtypes.

TEMPLATE METHOD Instance 2: AbstractLineDecoration The abstract class `AbstractLineDecoration` provides a standard implementation decorating lines. The automapping process was limited in the same way as in the previous case, and we performed all mappings by hand. The final mappings are shown in Table 5.7.

We encountered similar issues with accessibility modifiers as mentioned above, with four private fields and one protected method requiring accessibility adjustments during the refactoring.

5.2.2 Other Patterns in JHotDraw

We did not transform all design pattern instances in JHotDraw. Because some pattern implementations vary considerably in structure from their GoF pattern definitions, an application of the CCC refactoring was not possible. In other cases, pattern implemen-

Table 5.7: Role mappings for the AbstractLineDecoration instance of the TEMPLATE METHOD pattern

Role Element	Mapped Program Element
AbstractClass	AbstractLineDecoration
ConcreteClass	ArrowTip
templateMethod()	AbstractLineDecoration.draw(..) AbstractLineDecoration.getBorderColor(..) AbstractLineDecoration.setBorderColor(..) AbstractLineDecoration.getFillColor(..) AbstractLineDecoration.setFillColor(..) AbstractLineDecoration.read(..) AbstractLineDecoration.write(..)
requiredField	AbstractLineDecoration.fFillColor AbstractLineDecoration.fBorderColor; AbstractLineDecoration.serialVersionUID;

tations were already sufficiently modularized, rendering a CCC refactoring superfluous.

Decorator

The DECORATOR pattern CCC refactoring is not applicable to the instance found in JHotDraw because of a limitation of the aspect-oriented solution. The AOP solution is static with respect to assigning *Decorators* to *Components*, while in this particular instance the dynamic composition of decorators is essential.

Prototype

The PROTOTYPE pattern instance in JHotDraw is a specialized form of the GoF pattern. Since it does not create exact copies, our AOP solution and consequently the CCC refactoring is not applicable.

State and Command

We did not refactor the STATE and COMMAND pattern instances since we have to date not developed appropriate CCC refactorings. The challenge in specifying such refactorings lies in the fact that the non-modularized pattern structure and the AOP solution are structurally so different that it is hard to describe the transformation in terms of the elements of the original concern (role elements). It is likely that a more detailed concern model would have been helpful, as the transformation granularity for this refactoring is not types and members, but blocks and statements. Such a refinement of our model is future work.

Figure 5.4: Concern structure of the LOGGING CCC

5.3 Study 2: Other Crosscutting Concerns

In this study, we address the applicability of the approach to the refactoring of other non-modularized CCCs, in particular non-design-pattern CCC implementations.

We decided to start with an investigation of what appeared to be the most popular example CCC for AOP: logging. Additionally, we investigated a number of examples taken from Laddad's aspect-oriented refactorings suggestions [49, 51] as representative examples of CCC refactorings. For each investigated CCC we developed a sample system featuring that concern. These systems were relatively small, consisting of 4-6 classes and less than 300 lines of code each. As for the first study, the investigated refactorings create code that compiles correctly and preserves the behavior of the target CCC.

5.3.1 Logging

We picked LOGGING as a canonical example that is mentioned throughout the AOP literature (e.g., see repeated discussions in [88]). Since there is no single generally accepted way for logging to be realized, we used a simplified analogon of the structure offered by Java's `java.util.logging.Logger` class. The concern structure features a role type *Logger* that provides two methods for logging events at the entry and exit of methods. This is reflected in the role methods *logEntryMethod* and *logExitMethod*. The concern structure is shown in Figure 5.4.

The refactoring consists of the following steps:

1. An aspect is created to store the logging code
2. An execution pointcut `methodEntry` is created to capture all call sites to methods to which *logEntryMethod()* is mapped.
3. A execution pointcut `methodExit` is created to capture all call sites to methods to which *logExitMethod()* is mapped.
4. `before` advice is created based on the `methodEntry` pointcut. Its body consists of the bodies of the methods to which *logEntryMethod()* is mapped as suggestions¹⁶.

¹⁶Code suggestions are mentioned in Section 4.1.

5. *after* advice is created based on the `methodExit` pointcut. Its body consists of the bodies of the methods to which `logExitMethod()` is mapped as suggestions.
6. Calls to all methods to which `logEntryMethod()` is mapped are removed.
7. Calls to all methods to which `logExitMethod()` is mapped are removed.
8. All methods to which `logEntryMethod()` is mapped are removed.
9. All methods to which `logExitMethod()` is mapped are removed.

The refactoring was described entirely in terms of existing refactoring primitives, and was applied successfully to the example system. Since the concern description lacks structure, the automapping facility suggests partial mappings for the investigated sample system based on a seed mapping: given a mapping for one of the two role methods, the type mapping could be (trivially) derived.

Describing this CCC refactoring was straightforward and took about one hour. We reused the four refactoring primitives *create pointcut*, *create advice*, *textitremove member*, and *remove method calls* and did not develop any new primitives for this example.

5.3.2 Laddad's AOP Refactoring Examples

Laddad presents a number of OO-to-AO refactorings of crosscutting concerns in [49]. With the exception of the WORMHOLE refactoring, the complexity of these examples is lower than for most of the design pattern refactorings. This means that the concern structures are generally smaller (fewer role elements) and less complex (fewer role relations).

The different nature of the concerns in question required the development of two new refactoring primitives and extensions to three existing primitives. Four primitives, such as the *move member to aspect* primitive, could be reused without changes.

Extract Interface Implementation

This refactoring is used when multiple types implementing an interface provide the same basic implementation for one or more of its methods, causing scattered and duplicated code. In the refactoring, such duplicate implementations are removed from types and replaced with inter-type declarations that attach default implementations directly to the interface. Note that this refactoring has an aspect-oriented solution and is not related to the object-oriented *extract interface* refactoring [28].

The concern structure consists of the role types *Interface* and *Implementation*, the role method *implementedMethod()* and the role field *requiredField*, as shown in Figure 5.5.

The CCC refactoring proceeds as follows:

1. An aspect is created to store the inter-type declarations code
2. All methods to which *implementedMethod()* is mapped are replaced with appropriate inter-type declarations on the type to which *Interface* is mapped.

Figure 5.5: Concern structure of the EXTRACT INTERFACE IMPLEMENTATION CCC

3. All fields to which *requiredField* is mapped are replaced with appropriate inter-type declarations on the type to which *Interface* is mapped.

This CCC refactoring relies heavily on the *move object member to aspect* refactoring primitive. We had to adjust the primitive to prevent duplication of inter-type declarations: since the crosscutting in this CCC is comprised of identical implementations of the same interface method, a duplication-unaware refactoring would introduce the extracted code fragments multiple times. This customization was not required before as the type from which a member was removed and the type which the inter-type declaration targets were always the same. In this case, however, the target of the inter-type declaration (the interface) is different. Extending the refactoring primitive consisted of adding an appropriate check. No new refactoring primitives were required for this example.

The automapping facilities proved to be of little help due to the generic concern description and the resulting minimal concern structure.

Extract Method Calls

This refactoring targets implementations where calls to the same single method are scattered in multiple methods in a class. This can occur in security permission checks, class-level logging, or persistence session management, for example. If it is desired to modularize these calls, the solution, similar to the LOGGING example, is to advise the caller methods in the class to make the method call. Figure 5.6 illustrates the schematics of this transformation, while Figure 5.7 shows the concern structure.

The role-based realization is similar to that of the LOGGING concern above. The developer has to decide whether the resulting advice are before or after advice. It is conceivable that a further refinement of the utilized *replace method calls with pointcut and advice* refactoring primitive could automatically identify the position of the original call statement within their enclosing methods and generate the appropriate advice type, but this is left for future work. The refactoring consists of the following steps:

1. An aspect is created to store the aspect-oriented equivalent of the original method calls.
2. An execution pointcut methodCall is created to capture all call sites for all methods to which *calledMethod()* is mapped.

Figure 5.6: The Extract Method Calls Refactoring as shown in [51].

Figure 5.7: Concern structure of the EXTRACT METHOD CALLS CCC

3. An advice is created based on the `methodCall` pointcut. Its body consists of a call to the method to which `calledMethod()` is mapped as a suggestion. The type of advice (e.g., `after` or `before`) is decided by the developer during the planning phase.
4. Calls to all methods to which `methodCall()` is mapped are removed.
5. All methods to which `methodCall()` is mapped are removed.

Note that this refactoring can not preserve the behavior of the original program in a few special cases. For example, if a call to the extracted method occurs within a loop, or if calls occur neither at the beginning or the end of their enclosing method. Note that this is a limitation of the proposed aspect-oriented solution, not our role-based refactoring approach. All these cases can be resolved through appropriate enabling refactorings that expose the affected method calls. We have not yet developed specialized impact analyses to automatically identify these specific problems, but expect their implementation to be straightforward.

Automapping is not needed for this case as the concern structure consists of a single role element `calledMethod()`, for which we expect only a single mapping. We could have reused the CCC refactoring from the `LOGGING` example that allows a developer to determine the nature of the advice through a mapping to either of the two role methods, but chose to implement a different case for illustrative purposes, where the developer is prompted for a decision during the planning phase. We did not require any new refactoring primitives for this example and were able to reuse *create pointcut*, *create advice*, and *remove method calls*.

Replace Argument Trickle by Wormhole

Among the investigated non-pattern CCCs, the `WORMHOLE` refactoring was clearly the most complex. Its refactoring addresses the problem of argument trickling, that is the passing of arguments through a call stack. Argument trickle can be observed in the implementation of prefetching in operating systems code [16], for example. Other examples include transaction management (where the current transaction is the trickling argument) and authorization (where authorization data is passed). The concept of argument trickle is shown schematically in Figure 5.8. In this concern, a `caller()` method starts a call chain (consisting of various methods symbolified by the `trickle()` role method) that eventually reaches `callee()`. An argument is passed from `caller()` to `callee()`, but is not used by any of the `trickle()` methods. The proposed solution is to remove the offending argument and provide the required context through a `cflow()` construct as shown in [49].

The concern structure for our refactoring is shown in Figure 5.9. It consists of the role type *Argument* and the role methods `caller()`, `trickle()` (capturing methods along the trickle chain), and `callee()`. Note that the “unspecified type” notation means that multiple `trickle()` methods do not have to be methods on the same type, and neither does the `callee()` method.

The following steps describe the refactoring:

Figure 5.8: Argument trickle between *caller()* and *callee()*, taken from [51]. An argument gets passed through the call stack but is only used within the *callee()* method.

Figure 5.9: Concern structure of the REPLACE ARGUMENT TRICKLE WITH WORMHOLE CCC

1. An aspect is created to store the extracted call context
2. The type to which *Argument* is mapped is removed as a parameter from methods to which *caller()* is mapped.
3. The argument of the type to which *Argument* is mapped is removed from all methods to which *trickle()* is mapped.
4. Calls to methods to which *callee()* is mapped from within any method to which *trickle()* is mapped are removed (since the *callee()* will be called by the aspect).
5. A pointcut request is created to capture the context of *caller()*.
6. A pointcut operation is created to capture all call sites to the method to which *callee()* is mapped.
7. A pointcut wormhole is created as a combination of operation and a cflow of request.
8. An after returning advice is created based on wormhole that calls the method to which *callee()* is mapped after operation, supplying the context from request.

This transformation reused the *remove method calls refactoring* and required the addition of a pair of new refactoring primitives to address the task of removing a method argument. *Remove method argument* affects the method declaration, and *remove call argument* its call sites. The existing primitive for the creation of pointcuts was extended to support the creation of cflow pointcuts, which were not needed before, and to allow for pointcut definitions that joined other pointcuts (step 7 above).

The auto-mapping facilities proved to be of limited help, since it does not take into account that methods in the trickle chain can be on different types. This is a limitation of the current implementation, not of the algorithm itself.

As the most complex of the non-pattern CCC refactorings, this one took approximately 5 hours to code and test, including the extensions of the set of existing refactoring primitives.

5.4 Summary

The JHotDraw study indicates that the role-based approach supports the semi-automatic refactoring of crosscutting concerns in non-trivial software, and for refactoring descriptions that were developed independently from the target software system. The second study shows that the approach can express non-design-pattern CCCs and support their refactorings.

We expected problems, especially in the JHotDraw study, since earlier, informal studies have shown that the investigated design pattern refactorings are complex and error-prone to execute by hand. The actual application of the role-based refactorings turned out to be successful, despite the far-reaching changes involved. On the other

hand, the cases that we did not refactor illustrate the importance of concern structure: if a concrete implementation deviates too much from the expected structure, it becomes necessary to employ enabling refactorings or define alternative refactoring descriptions. This is illustrated by the three OBSERVER instances, which show varying degrees of adherence to the GoF design pattern structure. Two instances were successfully refactored, while the third (Figure — `FigureChangeListener`) is structurally so different from our CCC description that applying the refactoring was not possible. Structural deviations of the concern instance from the expected CCC structure appear to be the primary limitation of our approach. This is discussed in more detail in Chapter 6.

Similar limitations can be observed if the original concern and the AO solution are too different, and the refactoring can not be expressed in terms of the role elements of the original concern. This was the case for the STRATEGY and COMMAND pattern CCCs. It is likely that role-based refactoring descriptions exist for such cases, but they are not obvious.

The non-design-pattern refactorings from the second study could utilize the existing refactoring primitives to a large extent, except for the WORMHOLE refactoring, which required three changes to the existing primitives. This is not surprising as this is the first CCC for which the solution involved changing method parameters or creating `cflow` pointcuts. Overall it appears that a library of proven refactoring primitives is conceivable. We expect that, with the study of more CCC refactorings, such a library will converge to a comprehensive set of refactoring primitives.

A number of refactorings show similar underlying principles. The TEMPLATE METHOD, LOGGING, EXTRACT CONCERN IMPLEMENTATION, EXTRACT INTERFACE IMPLEMENTATION, and EXTRACT METHOD CALLS all focus on moving a set of members to an aspect by replacing them with inter-type declarations. This appears to be a common theme in OO-to-AO refactorings.

Based on the number of program elements identified by the automapping facilities as part of the concern implementation, it is apparent that automapping was overall more useful for the design pattern CCCs, mostly because those offered more structural information to leverage. The non-pattern concerns, on the other hand, generally required considerably fewer mappings. Role elements often mapped to only one or a small number of program elements. This is not only a result of the smaller scale of the example systems that we targeted in the second study, but for most of these patterns, multiple mappings are not expected. The exception is again the WORMHOLE case, for which we can expect multiple mappings for the *trickle()* role method.

Chapter 6

Discussion

The previous chapters have shown that our role-based approach enables the semi-automated refactoring of non-modularized crosscutting concerns into aspects. In this chapter we discuss the benefits and limitations of our approach and associated tool implementation.

6.1 Benefits

Our approach abstracts the complexity associated with refactoring CCCs. Refactorings of crosscutting concerns are composite refactorings that consist of multiple refactoring steps, and target multiple program elements. Our approach allows these refactorings to be planned and executed as a single operation to the developer. This has a number of associated benefits.

The role-based approach to describing refactorings aids the execution of a planned refactoring as the developer does not have to conceptualize the system in intermediate refactoring steps, during which the refactoring is only partially finished, reducing the conceptual complexity of dealing with CCC refactorings. It also becomes possible for the developer to undo and reapply a changed/adjusted CCC refactoring without having to go through the various transformations steps by hand.

Part of a refactoring description is a definition of the target concern structure in terms of the role abstractions. Our JHotDraw study showed that a comparison of this structure and the structure of the target program aids developers in identifying program elements pertaining to the implementation of the CCC, and thus helps to avoid incomplete refactorings.

The approach allows the description of CCC refactorings to be defined separately from a concrete instance, enabling a single refactoring description to apply to multiple instances of the target concern over more than one system.

6.2 Limitations

This section outlines the limitations of our approach and tool. For the latter, we distinguish between limitations based on the supporting technologies (AspectJ, AJDT, and the Eclipse platform) and our current implementation.

6.2.1 Approach

The main limitation of the approach is the dependence on a specified concern structure. The complexity of CCC refactorings also impacts the feasibility of behavior preservation.

Importance of Concern Structure

The concern structure, which we define in terms of roles, is integral not only to defining the refactorings, but also helps in mapping role elements to program elements. Potentially problematic are cases where a concrete concern varies structurally from an abstract concern description, and cases where the CCC in question has very little structure to utilize.

The abstract description of a CCC is intended to capture the structural essence of several different ways to implement the CCC. However, depending on the CCC, a particular implementation may vary so much from the defined structure that a refactoring is not possible using the role-based refactoring description. It is an open question how flexible the description of an abstract CCC should be to capture the different variants.

The JHotDraw variants of the OBSERVER pattern are sufficiently diverse to illustrate some of the range of implementation variants that need to be considered. For example, these instances exhibit these five divergences from the pattern as documented in [28].

1. *Subjects* accept only a single *Observer*.
2. The *notify()* role implementation appears only in concrete *Subjects* but not on the *Subject* interface.
3. Multiple pairs of *notify()* and *update()* methods exist within the same pattern instance.
4. The *observers* role field implementation is used outside of the pattern context.
5. Non-observer content is stored in the concrete *observers* role field.

Our refactoring description addressed variants 1, 2, and 4. Both variants 3 and 5 can be addressed by appropriate enabling refactorings.

It seems reasonable to expect the abstract CCC description to be flexible enough to capture at least variants one and two. The fifth variant, on the other hand, has so little in common with the original pattern, that a separate CCC description is necessary to capture it.

In the third variant, the *Observers* provide multiple *update(Subject)* methods for the same set of *Subjects*, which is a special case of the *push* model ([28], pp. 298). Although the structure of the pattern is similar to the CCC pattern description, the associated refactoring instructions do not cover this case, as it is difficult to capture the associations between a varying number of *notify()-update()* pairs. Since the structure is similar, the refactoring description should be extended to cover this variant. This could be realized by extending the way CCC structures are described to include a tag

identifying sets of role elements that need to be treated separately, but in the same principal way. This would require extending the automapping support to differentiate between such different sets, and updating the refactoring instructions.

In the fourth variant, references to the *observers* field from outside the pattern context can be detected by a static analysis; before the field is considered for deletion¹⁷, dealing with references to it would constitute decision points and a dialogue with the developer would be initiated.

In general, cases like this can either be addressed by enabling refactorings (placing the burden on the developer applying the approach), a more expressive role-based refactoring model (placing the burden on the developer creating the refactoring), or by a more intelligent refactoring tool. Ideally, we would like to keep refactoring descriptions simple so that they remain easy to understand. Hence, if enabling refactorings are not an option, it seems better to address as much of the variant implementation within the refactoring engine as possible. Certain common themes in CCC variants are covered this way. For example, not all concrete program elements mapped to a role type like *Subject* need to provide a declaration or definition of members associated with that role type, such as the *notify()* method. This particular case was mentioned above as variant 2. Other variants have to be addressed by enabling refactorings or by explicitly providing multiple CCC descriptions (for example, variant 5 above). It is conceivable that our tool can be extended to provide enabling refactorings in the planning phase that are not committed until the CCC refactoring is committed, and that can be undone as part of undoing the CCC refactoring. This would allow CCC refactoring descriptions to target a wider set of variant concern instances.

The existence of multiple CCC refactoring descriptions for variants of the same CCC raises the question of how a developer picks the proper description to match an implementation. We expect such a selection can be aided by the refactoring tool through a comparison of automapping results, which detects deviations of the expected concern structure and the structure of the actual implementation.

More experience with a broader base of CCCs as they occur in different kinds of systems is needed to better extrapolate commonalities with respect to implementation alternatives.

Concern structure is also relevant for tool support for mapping role elements to program elements. Since our approach for automapping is based on a structural comparison of the concern and the program, it works better when more structural information (number of role elements and role relations) for a concern exists. If a concern has little structure, automapping is less useful, but then the automapping support is also needed less. Consider the *LOGGING* concern: the concern itself has little structure, and even though the calls to the logging facilities can be spread throughout the system, the only mapping required is one type and one or two methods. All call sites to the method(s) that need to be updated are identified automatically as part of the refactoring.

Although the investigated non-design-pattern refactorings were performed on sample systems and not real-world software, they do show that the role-based refactoring approach is not limited to design pattern applications. Unfortunately, Laddad's refac-

¹⁷The refactoring renders the *observers* field obsolete, as the newly created aspect takes over the responsibility of maintaining a mapping between *Subjects* and their *Observers*.

toring examples [49] represent the only at least partially formalized non-pattern CCC refactorings available. A larger set of CCC/AOP-solution pairs would help to extrapolate our findings further. Based on our experiences with the design pattern refactorings, we expect our non-pattern CCC refactorings to be applicable to larger software systems as well, but verification of this conjecture is left for future work.

Preservation of Behavior vs. Preservation of Intent

As mentioned in Section 1.1.2, some transformations may result in minor behavior variations to the code base, which we consider to be acceptable if the original intent of the concern is preserved. An intent-preserving refactoring preserves the overall requirements of the system, but may change the semantics of the program locally.

Impact analysis points out potential behavior variations before refactoring instructions are executed, and developers can usually avoid behavior implications by adjusting or relaxing the original CCC implementation before executing the refactoring instructions¹⁸. There is a tradeoff: these non-CCC refactorings require knowledge of the problem and additional effort, while otherwise the behavior of the system might vary slightly. A further investigation of cases that do not preserve behavior is left for future work.

A related problem is represented by complex transformation steps pertaining to a CCC refactoring that can not be automated and require developer interaction or post-refactoring review. For example, in the CHAIN OF RESPONSIBILITY pattern, concrete instances of the role method *handle(Request)* contain code that does two things: deciding whether to handle a request (or to pass it along in the chain), and actually handling it. In the aspect-oriented version of the pattern, this is broken down into two methods: *acceptRequest(Request)* and *handleRequest(Request)*. Since it is difficult to automatically determine how to split the original method, the current realization only creates the appropriate method bodies, and leaves the original code as a (commented out) suggestion in each of them.

6.2.2 Supporting Technology

Three supporting technologies were utilized in the implementation of our refactoring tool: The program models of the Eclipse IDE, the AOP language AspectJ, and AJDT, a realization of support for AspectJ in the Eclipse environment.

AspectJ

The AspectJ language imposes limitations on certain refactoring primitives. For example, a number of the refactorings involve (among other refactoring steps) replacing members that pertain to the implementation of the target CCC with inter-type declarations, effectively moving their code into an aspect. Since AspectJ does not support the inter-type declaration of protected members, targeting a protected member leaves the developer with three options: either to use package protection for the new IDT, which

¹⁸Since such enabling refactorings occur before the application of the CCC refactoring, impact analyses will take these changes into account.

requires the aspect to be in same package as all subtypes of the type in question (this might not be possible), to change the access modifier to public, which offers even less protection for the member, or to employ an enabling refactoring to adjust the visibility of the member.

A similar problem arises when private members are replaced with inter-type declarations: a private member is not equivalent to a private inter-type declaration; the former is only visible within the type, while the latter is only visible within the aspect. This situation can also be resolved through enabling refactorings. This can occur, for example, in the TEMPLATE METHOD and EXTRACT INTERFACE IMPLEMENTATION refactorings, where concrete members are replaced with inter-type declarations. Generally, these particular transformations are semantics-preserving, but since they can expose program elements by changing their accessibility, the developer needs to consider the implications on the system. Since a tool can not decide whether such a change preserves the intent of the CCC, these refactorings can only be semi-automated and require developer interaction.

In addition to these limitations, AspectJ does also not allow the inter-type declaration of static members and constructors, which can matter in the case of the TEMPLATE METHOD refactoring where an abstract class is replaced with an interface and appropriate inter-type declarations.

AJDT

When manipulating a crosscutting Java program in the AJDT/Eclipse environment, a number of internal representations of the program must be consolidated. Eclipse provides abstract syntax trees of programs (the JDT Core AST) and a Java model on top of that (the IJavaModel, also known as the JDT model). These representations do not contain any information about crosscutting structure (i.e., AspectJ program constructs). AJDT has its own AST (an AspectJ AST) that is the basis for AJDT's structure model ASM. The latter is roughly equivalent to the IJavaModel, but contains crosscutting information. The relationship between the representations is shown in Figure 6.1.

The fact that AJDT provides neither an open API for accessing or changing its AspectJ AST made it unsuitable for our purposes (although an extension of Eclipse's AST manipulation support is planned). The ASM, while containing crosscutting information, does not provide the entire information of the JDT/IJavaModel (for example, it does not contain information about a method's parameter types), which rendered it unsuitable for our purposes as well. It is also more complex to retrieve information from it than from the IJavaModel¹⁹.

This impacts our current implementation and limits its applicability to pure Java systems. An extension of our tool to create and maintain its own AST representation using information from both the IJavaModel and the AJDT AST is conceivable as future work. Currently, aspect-oriented constructs are not part of the utilized program representation (the JDT/IJavaModel), unless they are "masked" as outlined in Section 4.1.5. The masking and unmasking of aspect-oriented constructs requires an additional pre- and post-processing step of the involved sources.

¹⁹ "[The IJavaModel] is better and easier to use [than the ASM]". – Mik Kersten, AspectJ and AJDT Committer. Personal communication.

Figure 6.1: The programming models of Eclipse and AJDT and their relationships (schematic)

We do show that consolidation of the different models is possible to some extent by implementing an impact analysis that checks if the proposed changes affect existing pointcuts. Recently, AJDT incorporated an extension of the `IJavaModel` that maps between the `IJavaModel` and the `ASM`. Although this extension does not yet provide an API, this is planned for a future version of AJDT. With such a facility, it will be possible to combine the benefits of the `IJavaModel` with the crosscutting information of the `ASM`, at which point it will be desirable to adjust our tool to make use of this facility. This is future work.

The Eclipse Program Models

The Eclipse APIs for both refactoring and AST manipulation were not public for large parts of the initial development of the refactoring tool. The APIs also changed considerably from build to build so that we were forced to abandon the idea of tying into these facilities. Instead, we ended up implementing all program transformations as text manipulations of the source code of the system in question. As in our current implementation no AST representation of the refactored system exists until the entire refactoring is completed, the feasibility of statically checking the proposed program transformations is limited. This could be addressed by an extension of our tool that maintains its own AST representation.

6.2.3 Tool

Using text manipulations of the program code instead of AST manipulations has a number of drawbacks related to inference between textual edits, as outlined in Section 4.1.3.

Another issue to consider is the impact of the implementation of CCC refactorings as string manipulations on the performance of the refactoring tool. Much of the refactoring time is spent on pattern matching and string manipulation. It seems likely that an implementation utilizing AST manipulation would achieve better performance.

Program Analyses

The program analyses provided by our tool as part of the mapping algorithm and the various impact analyses are not and were not intended to be comprehensive, and leave considerable room for improvement. Research efforts in the area of program analysis for refactorings (e.g., [3, 7, 23, 27, 47, 58, 68, 81]) can potentially be utilized in future versions of our tool.

Auto-Mapping

It is conceivable that the mapping algorithm could be extended with refactoring-specific program analyses to avoid suggesting invalid mappings. For example, for the *Extract Interface Implementation*, type constraints could be used to restrict the mappings of the *implementedMethod()* role method so that all mapped methods have the same signature. In our current implementation, this check is instead performed before the refactoring is applied (i.e., in the planning phase). A future version of our refactoring framework could move parts of the impact analyses into the mapping phase to avoid illegal mappings in the first place. This appears feasible as the refactoring is always chosen before the mapping is performed.

Impact Analyses

We see several ways to extend the current set of impact analyses. Since CCC refactorings (and a number of the refactorings primitives we employ) are composite refactorings, we believe that Kniesel's and Koch's work on the static composition of refactorings [47] could aid in determining the constraints of these refactorings, allowing us to compute their overall pre-conditions. Since both CCC refactorings and refactorings primitives are defined independently from specific programs and available in libraries (i.e., reusable), this computation would be required only once per composite refactoring, and would affect all subsequent applications of the refactoring. Expected benefits include the optimized verification of pre-conditions due to the removal of redundant or unnecessary checks.

When members are moved up in the inheritance hierarchies (for example, in the *Extract Interface Implementation* CCC refactoring), variable declarations may need to be adjusted. This specific problem of *refactoring for generalization* is addressed for the object-oriented case by the work of Tip et. al. [81]. If these declarations are not updated, they might be too overspecific, which leads to conflicts with object-oriented design principles. Although it is not clear how well this approach works in an aspect-oriented context (consider, for example, how `declare parents` statements might affect the type constraints [68] used), it appears to be a useful technique for automatically identifying and executing generalization refactorings as part of CCC refactorings.

To ensure the safety of the transformation of fields, as in the *Change Field Type* refactoring primitive, we could utilize an aspect-oriented extension of the refactoring support for class library migration presented by Balaban et. al. [3]. Currently, such an analysis is not required as the primitive is only used within a single CCC refactoring (*Memento*) and the replacement is straight-forward. However, for future extensions of the refactoring framework, a safer and more general implementation is desirable.

A possible way to help automatically decide whether the intent of a concern implementation might be compromised in the course of a (planned) refactoring is based on the differentiation between various kinds of behavior preservation as suggested by Mens et. al. [58]. It is conceivable that by classifying different kinds of behavior preservation according to their impact on the rest of the system, a tool could estimate which refactorings are behavior-preserving (safe), which might produce “acceptable” behavior deviations (requiring developer feedback), and which might violate the intent of the implementation (unsafe).

CCC Refactorings

The currently supported set of CCC refactorings aim to generate aspect-oriented implementations, which are mainly based on our previous work [38]. These implementations have been shown to have design tradeoffs [29], and certainly leave room for improvements. It is likely, for example, that future versions of the AOP design pattern implementations will utilize generic types (introduced in Java 1.5, and supported by AspectJ 5). For a transformation of our AOP implementation, an aspect-oriented version of refactoring-to-generics approaches [23, 27] based on type constraints could be utilized. Our investigated CCC refactorings, on the other hand, do not utilize such transformations, and do not require such analyses.

6.3 Future Work

Our results suggest several directions for further experimentation, including investigating other crosscutting concerns and their potential for role-based refactoring to aspects, extending the refactoring model to complex refactorings in general, and utilizing the role-based concern model to provide concern-specific views of software.

6.3.1 Refactorings for other Crosscutting Concerns

One avenue for future research is to apply the role-based refactoring approach to a wider range of crosscutting concerns. Unfortunately, aspect-oriented programming is still a rather new idea and investigations of crosscutting concerns and their AOP equivalents in a fashion detailed enough to provide a basis for a CCC refactoring description are rare. The design pattern study [38] was a good start as it showed how a set of common CCCs can be modularized in AOP, thereby providing suitable problem/solution pairs as a basis for CCC refactorings. Informal refactoring descriptions, such as those by Laddad [49] are an important next step, but more research in the area of crosscutting concerns is needed.

Another option is to vary the implementation language. Although the role-based refactoring approach is inherently language independent (except for refactoring instructions, which are generally tied to an implementation language), it remains to be shown how well the approach works for other aspect-oriented languages. However, the lack of current reference implementations for common crosscutting concerns is even more pronounced for AOP languages other than AspectJ.

The role-based refactoring approach is useful whenever sets of program elements have to be modified in the same systematic way. A possible avenue of research is to investigate how applicable this concept is to large-scale refactorings [26] in general.

6.3.2 Improvements of the Refactoring Tool

Our tool's realizations of the individual steps of the refactoring workflow can be improved, as outlined below.

Suggesting Refactorings

Currently our approach requires the developer to choose an appropriate CCC refactoring by hand. It is conceivable that the tool could be extended to integrate an aspect-mining approach, e.g., [11, 21, 56, 74, 82, 84], so that non-modularized CCCs can be identified and appropriate refactorings can be suggested. The refactoring tool's library of CCC refactorings could form the basis of a directed mining approach, suggesting the CCCs for which to mine.

For the specific case of design pattern refactorings we expect that pattern mining approaches, e.g., [2, 4] cannot only be utilized to identify the program elements comprising a non-modularized design pattern concern, but to also automatically suggest mappings from appropriate role elements.

Describing and Executing Refactorings

Refactorings are described as elemental AOP refactoring steps, referring to the role elements identified in the CCC description. Currently, they consist of calls to a library of refactoring primitives we have implemented. The declarative nature of these descriptions suggests that a language could be developed for them. This would allow us to perform some basic consistency checks during parsing, such as identifying role elements of a concern description that are not referenced in the refactoring instructions.

Mapping Roles

Our current model for the auto-completion of mappings between roles and concrete program elements uses both structural and lexical information. This approach has made the mapping step tractable for the examples we have investigated. However, we may be able to improve this process by using semantic information about the internals of role elements, in particular role methods. For example, we know that within the scope of the *attach(Observer)* method in the OBSERVER pattern, the argument object has to be added to some sort of data structure. In other words, a candidate method whose scope contains calls to an *add(...)*, *put(...)*, or *insert(...)* method of an aggregate data structure is a more likely candidate than one whose scope does not. This additional information could be used to further refine the ranking of candidate methods and to suggest likely matches to the developer.

As an alternative to our comparison of concern and program structure, we considered using a subgraph-matching approach as utilized in [6], but this did not provide us

with feedback on the level of single program elements, making it difficult to combine it with other analyses and utilizing the lexical information available.

Our tool already implements a lexical analysis to differentiate between candidate program elements that have similar structural properties. Instrumental to this process is the specification of name fragments and their “weights”, as described in Section 4.2. We expect that it is possible to extend the tool to “learn” good weights from the mappings of refactorings it performs. For each role element of a CCC refactoring, the tool could keep track of the names of mapped program elements and use this set of names to generate common name fragments.

6.3.3 Automatic Generation of Refactoring Instructions

We have discussed the benefits of an explicit concern structure earlier in this chapter. Ideally, we would like to extend the approach so that a similar structural description exists for the refactored code. Being able to compare the two concern structures would ease reasoning about the refactoring process and reduce its conceptual complexity, as well as serve as a documentation artifact for complex changes to a software system.

It is conceivable that by describing not only the concern structure (as we currently do), but also the structure of the final, refactored concern, an explicit definition of refactoring instructions may be unnecessary: we believe that based on the structural differences of these descriptions, a tool could infer some or all of the refactoring instructions necessary to perform the transformation.

Alternatively, the refactoring could be reduced to *just* the resulting concern structure, requiring only a mapping of (non-modularized) program elements to their associated role elements in the resulting (modularized) concern structure. A tool would generate the refactoring instructions based on the structure of the resulting concern and the provided mapping. We expect this to work best for small concerns, since without a structural description of the non-modularized concern, automated mapping support is difficult, and incomplete mappings are harder to detect.

6.3.4 Exploratory Generation of Role-Based Refactorings

Currently, developers must explicitly create (or reuse) a role-based concern description and associated refactoring instructions. One goal is to make complex refactorings easier to formulate and execute.

We envision a lightweight implementation of our refactoring approach which allows developers to build up a description of their refactoring targets (as sets of role elements) in an exploratory fashion. That is, while exploring the code, developers can define role elements and mappings to concrete program elements they investigate. Role relations and further mappings are suggested based on the structural relationship of the mapped program elements, or are explicitly defined by the developer. We expect this to be helpful for defining any kind of principled, far-reaching refactoring, not just in an aspect-oriented context. One possible set of refactorings to investigate is Fowler’s “big” refactorings [26].

The advantages we expect from creating a role-based concern model are the same as for role-based refactoring in general: the concern structure can help identify over-

looked program elements that need to be changed, and the description of refactoring instructions in terms of roles makes the whole refactoring more planable.

6.3.5 Extensions of the Role-Based Concern Model

The close relationship between roles and crosscutting behavior suggests that incorporating the roles as a language concept can aid reasoning about crosscutting concerns.

Using Join Points to Specify Mappings

Currently, automapping support fails in a few select cases where the provided concern structure is insufficient to identify matching program elements. For these cases it is an open question how to effectively map roles to the concrete program elements that correspond to them. We propose to investigate this overlap and develop a clear join point model that is usable for specifying where roles apply.

Such an approach would make it possible to avoid mapping program elements through code exploration where each element has to be explicitly selected. Instead, we believe that developers would be aided with facilities that translate their partial knowledge of the system into a mapping suggestion. For example, if a developer is trying to identify an instance of the `OBSERVER` pattern in a system, but only knows that the *Observers* are in a subpackage of the `networking` package and their name contains "Listener", he could use `networking.*.*Listener*` to identify matching elements.

We expect that using standard AspectJ techniques for specifying pointcuts, which generally consist of regular expressions over named program elements, will not prove to be sufficient, as this approach fails when trying to specify that two sets of program elements matched by regular expressions are the same. This limitation is addressed by alternative frameworks for expressing join points, such as LogicAJ's generic aspect model [48]. With LogicAJ-like pointcuts it would, for example, be possible to capture instances of the `SINGLETON` pattern by specifying that classes are sought that have a static method returning an object of that class, e.g., `static ?class ?class.*(..)`.

Chapter 7

Related Work

This chapter describes work related to the role-based refactoring approach described in this dissertation, and highlights the novel aspects of our work. We discuss other work on refactoring, alternative models for crosscutting concerns, the relationship between roles and aspects, and other alternative aspect-oriented implementations of design patterns.

7.1 Refactoring

Griswold and Opdyke officially introduced the term refactoring virtually at the same time [32, 67]. Their work provides the theoretical basis for automated refactorings realized in many refactoring tools, most notably the Smalltalk [30] Refactoring Browser [71, 95]. Contemporary IDEs, such as Eclipse [91], typically offer some forms of refactoring support. Fowler popularized refactoring by providing a catalogue of refactorings [26].

Most refactoring research has targeted low-level program transformations in functional and object-oriented systems, with recent efforts concentrating on object-oriented software. The refactoring approach described in this dissertation focuses on refactorings of crosscutting concerns, which are composite refactorings consisting of multiple aspect-oriented refactoring steps.

7.1.1 Composite Refactorings

The concept of composite refactorings, i.e., refactorings that consist of multiple individual refactoring steps, was first introduced by Opdyke [67]. Fowler later used the term “big refactorings” to indicate the complexity of the changes involved [26].

An example of object-oriented composite refactorings is introduced by Kerievski’s concept of “refactoring to patterns”, which describe a series of refactorings that replace implementations with design pattern equivalents [44]. These transformations are similar to our CCC refactorings in that they target multiple, usually scattered and tangled, program elements, although the descriptions are informal and not tied to a concern model.

Composite refactorings introduce the possibility of interference between steps of the composite refactoring. Bardey investigated this problem and presented a dependency analysis approach to modelling multiple refactoring steps [7]. This is similar to the impact analysis we discuss in Chapter 3, but focuses on object-oriented code only and requires a detailed description not only of a refactoring’s preconditions, but also of

its effect on the system. An interesting property of his approach is the ability to reorder the individual refactoring steps to avoid potential conflicts.

Kniesel and Koch provide a similar form of analysis for composite refactorings based on backwards transformation descriptions [47]. Their generic formal model for the automatic composition of conditional program transformations is program-independent (i.e., the combined preconditions for a composite program transformation can be calculated independently of the program to which they might be applied). Interestingly, this approach applies to arbitrary conditional program transformations (even non-behavior-preserving ones). In that sense their approach appears to be a natural fit for analyzing the compositions of the elementary refactorings, such as CCC refactorings. The benefit of adopting this approach for the description of CCC refactorings is obvious: the series of individual refactoring steps can be analyzed in advance and it is likely that redundant (and potentially costly) precondition checks can be avoided (this is referred to as an “optimizing” effect). Although the examples they provide are on a much smaller scale than typical design pattern CCC refactorings, it would appear that the approach applies regardless of refactoring complexity.

7.1.2 Analyses for Ensuring the Behavior-Preservation of Object-Oriented Refactorings

Mens et. al. formalize behavior preserving program transformations as basic graph rewriting operations and statically analyze the dependencies between these operations at a semantic and syntactic level [58]. Their approach is language-independent, but a verification of whether it is possible to express refactorings affecting specific language features (e.g., Java features such as interfaces, inner classes, threads and exceptions) is not provided. *Well-formedness* constraints (i.e., constraints that guarantee a graph corresponds to a syntactically correct program) are essential to tie their graph notation to a particular programming language, like Java. These constraints are expressed using two mechanisms: *type graphs* (meta graphs expressing restrictions on program graphs) and *forbidden subgraphs* (subgraphs expressing illegal program configurations). The examples discussed are of considerably smaller scale than a typical design pattern CCC refactoring, and it thus remains to be seen if the approach scales to complex composite refactorings. Especially interesting about this approach is the differentiation between various kinds of behavior preservation. Mens et. al. discuss three kinds relevant to the investigated refactorings: *access preservation*, *update preservation*, and *call preservation*. It is conceivable that by classifying these and other kinds of behavior preservation according to their impact on the rest of the system, a tool could estimate which refactorings are behavior-preserving, which might produce “acceptable” behavior deviations, and which might violate the intent of the implementation.

Tip et. al. present an approach for the detection, analysis, and resolution of generalization refactorings [81] based on type constraints [68]. The approach addresses the problem of updating variable declarations in the course of refactorings that move members up the inheritance hierarchy, for example, as part of an object-oriented *Extract Interface* or *Pull Up Member* refactoring. Although their approach is designed for use in an object-oriented context and an extension to AOP might introduce new

challenges (consider, for example, how `declare parents` statements might affect the type constraints used), it could be a useful technique for automatically identifying and executing generalization refactorings as part of CCC refactorings, for example the *Extract Interface Implementation* refactoring.

Donovan et. al. present an approach to support the transformation of Java code to incorporate the generic type (i.e., parametric polymorphism) system of Java 1.5 [23]. The approach is based on a context-sensitive pointer analysis and realized in their fully automated translation tool called Jiggetai. A similar approach is presented by Fuhrer et. al. [27]. Fuhrer's approach provides a more scalable and general²⁰ alternative to Donovan's work, and has been realized as part of the refactoring functionality of the Eclipse IDE (introduced in Eclipse version 3.1). Both approaches make heavy use of type constraints, and Fuhrer's work further introduces a set of constraint generation rules that help avoid the manual creation of constraints, which is tedious and error-prone. The refactorings presented in these two works could be a useful basis for the transformation of a number of aspect-oriented design pattern implementations, for example the *OBSERVER* pattern, which would benefit from using generic types for storing *Observer* per *Subject*. Refactorings to use generic libraries are not part of the currently investigated CCC refactorings, however, so this work is currently not applicable to CCC refactorings themselves.

The particular problem of replacing (library) types is addressed by the work of Balaban [3]. A mapping specification between legacy classes and their replacements outlines how the replacement takes places, while type and escape analysis ensures that the required changes are safe. The mapping specification consists of a set of rewrite rules, which are an interesting alternative for a role-based refactoring description for small-scale transformations that focus on single types. We perform a similar, yet much less complex, type transformation in the context of the *Memento* CCC refactoring, which uses the *Change Field Type* refactoring primitive. Specific, non-trivial problems, such as the preservation of thread-safety, do not apply in this case and could therefore be ignored. For a more general implementation of that primitive, an aspect-oriented version of the analyses presented in Balaban's work would be useful.

7.1.3 Aspect-Oriented Refactorings

Refactorings of crosscutting concerns generally consist of multiple refactoring steps, some of which are aspect-oriented equivalents of traditional object-oriented refactorings, but take into account potential references to the changing program element from AO programming constructs. Adopting existing refactorings to make them aspect-aware is the topic of various work, e.g., [34, 42, 86].

With the introduction of AOP, new refactorings are possible that target aspect-oriented programming constructs. Collections of such refactorings have been presented by Monteiro et. al. [61, 62] and Iwamoto [42]. Their work confirms some of the low-level refactoring primitives we developed, for example their *move method from class to inter-type declaration* matches our *move object member to aspect* for methods.

²⁰The approach is more general as it allows for the accommodation of user-defined subtypes of generic library classes.

Aspect-oriented refactorings that target crosscutting concerns currently focus primarily on extractions of single program elements into aspects [62, 83]. Recent approaches utilize aspect mining to identify candidate code fragments for extraction to aspects [9, 74]. Mining approaches are either based on textual patterns [84], patterns in execution traces [11], high fan-in methods [56], or duplicated code fragments [12]. A qualitative comparison of aspect mining techniques is provided by Ceccato [15]. Our approach does not rely on aspect mining, but instead helps determine the extent of a crosscutting implementation using an automapping algorithm that compares the structure of the target program and the concern description. It is conceivable that aspect-mining techniques can be utilized in a future version of our role-based refactoring tool to automatically suggest refactorings.

Besides approaches that provide a formal description of the refactoring step(s), Laddad investigated a number of CCC refactorings and described the required transformation steps informally by providing code examples illustrating the concern implementation before and after the transformation [49]. Although such informal descriptions are helpful for refactorings executed by hand, they do not provide a sufficient basis for realization in a refactoring tool. However, as Section 5.3 illustrates, a set of cases investigated by Laddad could be translated into role-based refactoring descriptions.

Recently Binkley et. al. presented a partially implemented, human-guided approach for OO-to-AO refactorings that focuses on the specific problem of extracting OO code fragments into aspects [9]. Their approach relies on code which is partially *marked* to indicate candidates for the extraction. Although they propose to utilize an automated aspect-mining approach to provide the marking, this is not yet realized, and their tool requires the developer to mark code segments by hand. Their tool supports an interactive approach to aspect extraction consisting of *discovery* (determines applicable refactorings for the marked code), *transformation* (applies enabling OO refactorings), *selection* (lets the developer select appropriate AOP refactorings), and finally *refactoring*, which transforms the code. The actual code transformations are not implemented in the tool, but the authors assure that they pose no conceptual difficulty. Compared to our approach, Binkley et. al. do not provide a means to support identification of program elements pertaining to the implementation of the target concern, and their tool does not generate aspect code. The transformations are series of low-level AOP refactorings that are restricted to the extraction of code fragments into aspects — complex structural differences between original concern and aspect solution are not addressed.

The aforementioned approaches to aspect-oriented refactoring do not employ a concern model. As we have shown in this dissertation, the role-based concern model aids the planning and execution of CCC refactorings and the associated automapping algorithm helps to determine the exact extent of a concern implementation. In previous work we investigated the realization of CCC refactorings based on Robillard's *concern graphs* model [72]. Our role-based approach proved to be more suitable for the description of refactorings as concern graphs are always tied to an implementation, and do not provide a means to abstract sets of program elements of the same kind based on their function within an aspect, for example to differentiate between methods in an OBSERVER implementation acting as *attach()* and *detach()*.

7.2 Concern Modeling

The role-based concern model is the basis of our proposed CCC refactoring approach. Various models for concerns exist, with varying degrees of similarity to our model. For example, textual documentation can be seen as a model of the documented concern, as can program representations, such as aspects.

Our model can be considered to be above the program model and not tied to the implementation level. Sutton and Rouvellou present a general-purpose concern-space modeling schema called Cosmos [77] that offers a distinction between logical and physical concerns. Logical concerns capture “matters of interest” and can be considered separately from physical concerns, which represent more tangible properties of a systems, such as its hard- and software. Logical concerns are loosely related to roles, and physical concerns to concrete implementations, although their model is more general in that the logical concerns do not differentiate on a method/field level, and physical concerns comprise other tangible elements of a system besides its implementation, such as its hardware.

The model most closely related to our concern model is Robillard’s concern graph model [72], in that both are graphs consisting of program nodes that represent program elements. Robillard’s concern graphs aim to provide a description of the concern structure that is easy to manage and robust towards changes in the underlying program. Concern graphs are always tied to a concrete implementation, while our concern representations are abstractions of the essential elements of multiple, similar concern implementations. Furthermore, the actual program structure is only indirectly reflected in our concern descriptions: a single role element can correspond to multiple program elements in a concrete implementation, while for concern graphs the mapping is bijective. Lastly, concern graphs do not contain labels for individual nodes (role names are labels for role elements in our approach), so that different elements of the same type can only be differentiated by their position in the graph and not based on descriptive names.

For the specific purpose of modeling design pattern CCCs, Mapelsden et. al. introduce the visual design pattern modeling language DPML, built upon similar concepts as UML [55]. This multi-level approach (design patterns, pattern instances, and object models) makes it possible to show objects and their roles within the pattern and to generate pattern instances from them. The approach limits the role concept to types and instead models their behavior using constraints on types. A similar approach is presented by Florijn et. al., who suggest a fragment-based representation of design patterns [25]. A fragment depicts a design element such as a class, method or association. Patterns themselves and all elements in a pattern instance (classes, relationships among them, code) are represented as (graphs of) fragments. Neither of these two approaches uses their concern models for describing refactoring tasks but solely the concern structures.

7.3 Aspects and Roles

Both aspect-oriented programming and the role modeling concept are techniques that are similar in that they permit a flexible adaptation of object-oriented constructs [35]. Kendall has investigated aspect-oriented designs of role models, in particular alternative realizations for the `ROLE OBJECT` pattern and has shown that aspect-oriented mechanisms can represent roles effectively [43]. Lieberherr et. al. introduce the concept of Aspectual Components [53] and Aspectual Collaborations [54] which integrate component-based programming and AOP to support dynamic assignment of object behavior similar to role models. Similarly, Tamai et. al. propose a role model called Epsilon and an associated language EpsilonJ to realize object adaptation to environments [79]. Compared to these works, roles in our approach are not tied to a particular implementation, but the role concept is used in a more abstract fashion to describe the function of sets of program elements within a particular concern.

For the context of design pattern roles in particular, Mikkonen presents a formal model of behavioral layers called object slices [59] to model roles and their relationship to the concrete classes playing those roles. Florijn et. al. introduce a fragment model (see above) that represents participant roles as a particular kind of fragments [25]. Both of these approaches are specific to design patterns.

7.4 Aspects and Design Patterns

Design pattern implementations are strongly influenced by the implementation language, causing some patterns to either disappear when they are captured by language constructs, or to have structurally simpler implementations [66, 78]. For example, DemeterJ [90] is an adaptive aspect-oriented extension to Java that provides explicit support for the `VISITOR` design pattern.

Multiple authors have proposed and discussed individual aspect-oriented implementations of existing object-oriented design patterns. For example, Noda and Kishi provide an early comparison of their AspectJ and Hyper/J implementations of the `OBSERVER` pattern [63], separating the pattern concern from the application, while Sletten investigated an alternative realization of the `COMMAND` pattern [75]. Nordberg discusses the `VISITOR`, `OBSERVER`, and `EXTENSION OBJECT` patterns from the point of view of how AOP and component-based development can help in software module dependency management [64], and how replacing or augmenting OO indirection with AOP indirection can lead to better structure for some design pattern [65] implementations.

The study presented in Chapter 2 represents the first comprehensive investigation of a larger set of design pattern implementations in an aspect-oriented language. Since then, Tanaka and Ichisugi have confirmed that AOP implementation of design patterns can exhibit modularity benefits [94] in their AOP language MixJuice [41].

More recently, Garcia et al. have analyzed our proposed AOP design pattern implementations quantitatively [29] with respect to various code quality metrics, comparing them to their object-oriented equivalents. They confirm most of our findings from [38], but illustrate that separation of concerns, which was the main focus of our study, can be

conflicting with other desirable code properties, such as small size or cohesion of modularity units. In general, their study shows that our aspect-oriented implementations do have a set of associated design tradeoffs.

Chapter 8

Conclusions

Describing and executing refactorings of non-modularized crosscutting concerns (CCCs) into aspects is a complex task. Multiple scattered and tangled program elements need to be considered and changed together. Other than for traditional refactorings, preservation of behavior may not always be possible or even desired. Instead, these refactorings focus on preserving the *intent* of the original implementation.

The thesis of this research is that a role-based concern model enables the semi-automatic refactoring of crosscutting concerns, while preserving their original intent. To investigate this claim, we have developed a role-based refactoring approach and associated tool.

In our approach, a crosscutting concern refactoring is described using a concern description and associated refactoring instructions. The concern description abstracts the concern structure by expressing it in terms of role elements and their relationships. Refactoring instructions are formulated in terms of these role elements. Invoking a refactoring involves mapping the role elements to specific concrete program elements. After the mapping has been checked, the refactoring instructions are executed on the mapped program elements. Thus, refactoring descriptions are effectively separated from the systems to which they may be applied.

We demonstrated the feasibility of this approach through two case studies. In the first study, we showed how our approach and tool could be applied to refactor non-modularized crosscutting concerns in a non-trivial system, using refactoring descriptions developed separately from the investigated code base. The second study showed that the role-based refactoring approach can express and help execute a variety of crosscutting concern refactorings. For both studies, the tool we developed provided support for the semi-automation of the investigated refactorings.

This research makes four contributions to the field of software engineering.

First, we provide a working, language-independent approach for the semi-automated refactoring of crosscutting concerns into aspects. We provide a refactoring tool as a proof-of-concept implementation of the approach, instantiated for the Java and AspectJ languages. The application of this refactoring tool to refactor various CCCs into aspects provides valuable insights into the issues associated with developing tool support for the refactoring of crosscutting concerns.

Second, we provide a general model for describing crosscutting concerns and their refactorings in terms of roles. Because the model is abstract, CCCs and their refactorings described using our role-based model are not tied to a particular implementation. In addition, the model makes it possible to conceptualize complex program transformations as a single refactoring, and provides a basis for our approach to semi-automating CCC refactorings.

Third, we provide a library of refactoring primitives and CCC refactoring descriptions. As the case studies have shown, the generality of these descriptions allows CCC refactorings to be applied to software systems for which they were not specifically developed, and the refactoring primitives to be reused in the definition of new CCC refactorings.

Finally, we provide a comprehensive study of object-oriented and aspect-oriented implementations of design patterns, a set of commonly used concerns with often cross-cutting properties. This study asserts that refactorings of non-modularized crosscutting concerns into aspects can be beneficial. Researchers and developers can download our reference implementations freely to utilize the aspect-oriented implementations we provide. Our implementations have had considerable impact already, being featured in various books and articles on AspectJ, e.g., [31, 50, 52, 60].

In conclusion, role-based refactoring makes it possible to semi-automate the complex process of refactoring crosscutting concerns into aspects. As the library of refactoring primitives and CCC refactorings matures, we can envision the eventual integration of our approach into aspect-oriented integrated development environments.

Appendix A

Aspect-Oriented Terminology

- **Advice** – Encapsulates code to be run at *join points* specified by a particular *pointcut*. This code can either run before, after, or instead of the code at the join point, by using *before*, *after*, and *around* advice, respectively.
- **Aspect** – A unit of modularity for the encapsulation of *crosscutting concerns*.
- **Aspect-Oriented Programming** – A programming paradigm that provides explicit support for the encapsulation of *crosscutting concerns*.
- **AspectJ** – An aspect-oriented extension to the Java programming language.
- **Pointcut** – A program element that describes a set of *join points*.
- **Concern** – A facet of a program that might be of interest to the developer, for example a functional or non-functional requirement.
- **Crosscutting Concern** – A *concern* with an implementation that can not be modularized in a given decomposition. Non-modularized crosscutting concerns have an implementation that is *scattered* across multiple modules and *tangled* with other code. *Aspect-oriented programming* provides facilities for the modularization of crosscutting concerns. Common examples of crosscutting concerns include persistence, logging, caching, etc.
- **Declare Parents** – An AspectJ construct that allows aspects to add an additional supertype to an existing type.
- **Inter-Type Declaration** – Refers to an AspectJ-specific construct that allows an aspect to declare members on existing types.
- **Join Points** – A point in the execution of a program. Examples of join points include method calls, method executions, and field accesses.
- **Scattered Code** – Concern code that is not localized.
- **Tangled Code** – Concern code that is interwoven with that of other concerns.

Appendix B

Refactoring Primitives

In our refactoring framework, instructions for refactorings of crosscutting concerns are described as a series of calls to the library of refactoring primitives. Each primitive is an aspect-oriented composite refactoring. In the following, we describe the individual primitives we have developed.

B.1 Aspect Creation

As outlined in Section 4.1.2, primitives for aspect creation are different from other primitives in that they are applied before the rest of the refactoring instructions are executed.

The tool provides an aspect factory class that offers three parameterized options for the creation of aspects, which it delegates to the following specialized primitives:

- *Create Aspect* – creates a new stand-alone aspect to hold the modularized implementation of the target CCC. The new aspect is created in a specified package, and assigned the specified accessibility modifiers.
- *Create Subaspect* – as above, but creates an aspect as a subaspect of an existing abstract library aspect. Abstract members of the abstract aspect, including pointcuts, are automatically concretized as stubs.
- *Create Aspect from Library Aspect* – also creates the new aspect from an abstract aspect, but duplicates all of the abstract aspects concrete implementation, resulting in a stand-alone aspect.

B.2 Other Creational Primitives

These primitives create new program elements on aspects.

- *Create Advice* – creates an advice in a given aspect. This primitive is parameterized to allow specification of the advice body and the kind of advice to create (e.g., before, after returning, etc.).
- *Create Pointcut* – creates a pointcut in a given aspect. This primitive is comprised of several specialized pointcut creation primitives, for example *capture*

calls in pointcut, which captures all calls to a specified set of methods in a pointcut. Other primitives create compound pointcuts (i.e., pointcuts that are a composition of two or more existing pointcuts), or capture the execution of a set of methods.

- *Create Inter-Type Declaration* – creates an inter-type declaration (ITD) by delegating the actual creation to two more specialized primitives that are responsible for the creation of field ITDs and method ITDs, respectively.

B.3 Transformational Primitives

Refactorings in this category manipulate existing programming constructs.

- *Move Object Member to Aspect* – replaces an object member with an equivalent inter-type declaration. This is realized by specialized primitives for fields and methods. All primitives in this category rely on the creational primitives for ITDs mentioned above.
- *Change Abstract Class to Interface* – replaces an abstract class with an interface and appropriate inter-type declarations. The primitive relies on the *move object member to aspect* primitive, which in turn uses the creational primitives for ITDs.
- *Change Accessibility Modifier* – changes the accessibility modifier (public, private, etc.) of a member, for example to prepare them for replacement with an ITD.
- *Change Field Type* – changes the type of a field by adjusting the field definition. This primitive is used within other primitives that ensure that the field is not referenced improperly.
- *Change Imports* – adds or removes elements from the list of imports for a given type. This primitive is utilized by all other primitives that can have an impact on the imports of a type.
- *Remove Argument* – removes an argument from a method. This is a specialized refactoring primitive used within the REPLACE ARGUMENT TRICKLE WITH WORMHOLE refactoring.
- *Replace Method* – replaces a method with another, existing method and updates the call sites accordingly. Unlike the traditional *rename method* refactoring, this effectively changes the implementation of the target method.
- *Replace Object Method with Aspect Method* – replaces an instance method with a (static) aspect method. Call sites are updated so that the original target object of calls is passed as an additional parameter, for example
`mySubject.attach(myObserver) →
aspect.instanceOf().addObserver(mySubject, myObserver).`

B.4 Destructive Refactorings

Destructive refactorings remove program elements such as methods, fields and types. They are mainly used to dispose of program elements that are rendered obsolete by other refactorings, especially the “*replace ...*” series of refactoring primitives.

- *Remove Type* – removes a type from the system.
- *Remove Member* – removes a member. This addresses the removal of both methods and fields.
- *Remove Method Calls* – deletes all call sites to a particular method.
- *Remove Field References* – deletes all references to a given field.

Bibliography

- [1] Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. In *Proceedings of the 13th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 134–143. ACM Press, 1998.
- [2] Giuliano Antoniol, Roberto Fiutem, and Luca Cristoforetti. Using metrics to identify design patterns in object-oriented software. In *Proceedings of the 5th International Software Metrics Symposium (METRICS '98)*, pages 23–34. IEEE Computer Society, 1998.
- [3] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *Proceedings of the 20th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '05)*, pages 265–279. ACM Press, 2005.
- [4] Zsolt Balanyi and Rudolf Ferenc. Mining design patterns from C++ source code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM '03)*, pages 305–314. IEEE Computer Society, 2003.
- [5] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, 1999.
- [6] Elisa Baniassad. *Design Pattern Rationale Graphs: Linking Design to Source*. PhD thesis, University of British Columbia, BC, Canada, 2002.
- [7] Uwe Bardey. Abhängigkeitsanalyse von Softwaretransformationen. Diploma thesis, Computer Science Dept. III, University of Bonn, Germany, Feb 2003.
- [8] Laszlo A. Belady and Meir M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.
- [9] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated refactoring of object-oriented code into aspects. In *Proceedings of the International Conference on Software Maintenance (ICSM '05)*, pages 27–36. IEEE Computer Society, 2005.
- [10] Tim Bray, Jean Paoli, and Michael Sperberg-McQueen (Eds). Extensible Markup Language (XML) 1.0 (2nd Edition). W3C Recommendation, 2000.
- [11] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE '04)*, pages 310–315. IEEE Computer Society, 2004.

-
- [12] Magiel Bruntink, Arie van Deursen, Tom Tourwé, and Remco van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM '04)*, pages 200–209. IEEE Computer Society, 2004.
 - [13] Timothy Budd. *An introduction to object-oriented programming*. Addison Wesley, 1991.
 - [14] Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.
 - [15] Mariano Ceccato, Marius Marin, Kim Mens, Leon Moonen, Paolo Tonella, and Tom Tourwe. A qualitative comparison of three aspect mining techniques. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC '05)*, pages 13–22. IEEE Computer Society, 2005.
 - [16] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th International Symposium on Foundations of Software Engineering*, pages 88–98. ACM Press, 2001.
 - [17] James O. Coplien. Idioms and patterns as architectural literature. *IEEE Software*, 14(1):36–42, 1997.
 - [18] James O. Coplien. Software design patterns: Common questions and answers. In *The Patterns Handbook: Techniques, Strategies, and Applications*, pages 311–320. Addison-Wesley, 1998.
 - [19] Ole J. Dahl. *Structured Programming*. Academic Press, 1972.
 - [20] Jim des Rivières and John Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.
 - [21] Arie van Deursen, Marius Marin, and Leon Moonen. Aspect mining and refactoring. Presented at the International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE'03) at WCRE '03, 2003.
 - [22] Edsger W. Dijkstra. The structure of the “THE”-multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.
 - [23] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting java programs to use generic libraries. In *Proceedings of the 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pages 15–34. ACM Press, 2004.
 - [24] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. Seesoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

-
- [25] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *LNCS*, pages 472–495. Springer-Verlag, 1997.
 - [26] Martin Fowler. *Refactoring: Improving the Design of Existing code*. Addison-Wesley, 1999.
 - [27] Robert Fuhrer, Frank Tip, Adam Kiezun, Julian Dolby, and Markus Keller. Efficiently refactoring Java applications to use generic libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3586 of *LNCS*, pages 71–96. Springer-Verlag, 2005.
 - [28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
 - [29] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: a quantitative study. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 3–14. ACM Press, 2005.
 - [30] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1983.
 - [31] Joseph D. Gradecki, Nicholas Lesiecki, and Joe Gradecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, Inc., 2003.
 - [32] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, Seattle, WA, USA, 1992.
 - [33] William G. Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering Methodology*, 2(3):228–269, 1993.
 - [34] Stefan Hanenberg, Christian Oberschulte, and Rainer Unland. Refactoring of aspect-oriented software. *NetObject Days '03*, Erfurt, Germany, 2003.
 - [35] Stefan Hanenberg and Rainer Unland. Roles and aspects: similarities, differences, and synergetic potential. In *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS '02)*, volume 2425 of *LNCS*, pages 507–520. Springer-Verlag, 2002.
 - [36] Jan Hannemann, Ruzanna Chitchyan, and Awais Rashid. Report on the workshop on analysis of aspect-oriented software. In *ECOOP '03 Workshop Reader*, volume 3013 of *LNCS*, pages 154–164. Springer-Verlag, 2004.
 - [37] Jan Hannemann, Thomas Fritz, and Gail C. Murphy. Refactoring to aspects: an interactive approach. In *Proceedings of the Workshop on Eclipse Technology eXchange (ETX) at OOPSLA '03*, pages 74–78. ACM Press, 2003.

-
- [38] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, pages 161–173. ACM Press, 2002.
 - [39] Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 135–146. ACM Press, 2005.
 - [40] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, pages 26–35. ACM Press, 2004.
 - [41] Yuuji Ichisugi and Akira Tanaka. Difference-based modules: A class-independent module mechanism. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)*, volume 2374 of *LNCS*, pages 62–88. Springer-Verlag, 2002.
 - [42] Masanori Iwamoto and Jianjun Zhao. Refactoring aspect-oriented programs. Presented at the Workshop on AOSD Modeling With UML at UML '03, 2003.
 - [43] Elizabeth A. Kendall. Role model designs and implementations with aspect-oriented programming. In *Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 353–369. ACM Press, 1999.
 - [44] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.
 - [45] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072 of *LNCS*, pages 327–353. Springer-Verlag, 2001.
 - [46] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.
 - [47] Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52:9–51, 2004.
 - [48] Günter Kniesel, Tobias Rho, and Stefan Hanenberg. Evolvable pattern implementations need generic aspects. Presented at the Workshop on Reflection, AOP, and Meta-Data for Software Evolution. In *ECOOP '04 Workshop Reader*, volume 3344 of *LNCS*, pages 111–126. Springer-Verlag, 2004.
 - [49] Ramnivas Laddad. Aspect-oriented refactoring series. Online document at The Server Side, <http://www.theserverside.com/>, 2003.

-
- [50] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003.
 - [51] Ramnivas Laddad. Aspect-oriented refactoring: Taking refactoring to a new level. Tutorial at AOSD '05, <http://aosd.net/2005/tutorials/refactoring.php>, 2005.
 - [52] Nicholas Lesiecki. Enhance design patterns with AspectJ. Online document at IBM Developer Works, <http://www-128.ibm.com/developerworks/java/library/j-aopwork5/>, 2005.
 - [53] Karl J. Lieberherr, David H. Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.
 - [54] Karl J. Lieberherr, David H. Lorenz, and Johan Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 46(5):542–565, 2003.
 - [55] David Mapelsden, John Hosking, and John Grundy. Design pattern modelling and instantiation using DPML. In *Proceedings of the 40th International Conference on Tools Pacific (CRPITS '02)*, pages 3–11. Australian Computer Society, 2002.
 - [56] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE '04)*, pages 132–141. IEEE Computer Society, 2004.
 - [57] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP '03)*, volume 2743 of *LNCS*, pages 2–28. Springer-Verlag, 2003.
 - [58] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Proceedings of the 1st International Conference on Graph Transformation (ICGT '02)*, volume 2505 of *LNCS*, pages 286–301. Springer-Verlag, 2002.
 - [59] Tommi Mikkonen. Formalizing design patterns. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, pages 115–124. IEEE Computer Society, 1998.
 - [60] Russell Miles. *AspectJ cookbook*. O'Reilly, 2005.
 - [61] Miguel P. Monteiro and Joao M. Fernandes. Object-to-aspect refactorings for feature extraction. Industry Track paper at the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04), 2004.
 - [62] Miguel P. Monteiro and Joao M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 111–122. ACM Press, 2005.

-
- [63] Natsuko Noda and Tomoji Kishi. Implementing design patterns using advanced separation of concerns. Presented at the Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA '01, 2001.
 - [64] Martin E. Nordberg III. Aspect-oriented dependency inversion. Presented at the Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA '01, 2001.
 - [65] Martin E. Nordberg III. Aspect-oriented indirection beyond object-oriented design patterns. Presented at the Workshop "Beyond Design: Patterns (Mis)used" at OOPSLA '01, 2001.
 - [66] Peter Norvig. Design patterns in dynamic programming. Presentation at Object World '96, 1996.
 - [67] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.
 - [68] Jens Palsberg and Michael I. Schwartzbach. *Object-oriented Type Systems*. John Wiley and Sons, 1994.
 - [69] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
 - [70] Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 120–129. ACM Press, 2003.
 - [71] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.
 - [72] Martin P. Robillard and Gail C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 406–416. ACM Press, 2002.
 - [73] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
 - [74] David Shepherd and Lori L. Pollock. Ophir: A framework for automatic mining and refactoring of aspects. Technical Report 2004-03, Department of Computer and Information Sciences, University of Delaware, DE, USA, 2003.
 - [75] Brian Sletten. Beyond actions — a semantically rich command pattern for the Java Foundation Classes (JFC/Swing) API. Presentation at JavaOne '02, 2002.
 - [76] Jiri Soukup. Implementing patterns. In *Pattern languages of program design*, pages 395–412. ACM Press/Addison-Wesley, 1995.

-
- [77] Jr. Stanley M. Sutton and Isabelle Rouvellou. Modeling of software concerns in Cosmos. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD '02)*, pages 127–133. ACM Press, 2002.
 - [78] Gregory T. Sullivan. Advanced programming language features for executable design patterns. Lab Memo AIM-2002-005, MIT Artificial Intelligence Laboratory, 2002.
 - [79] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. An adaptive object model with dynamic role binding. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pages 166–175. ACM Press, 2005.
 - [80] Aleksandra Tesanovic, Mehdi Amirijoo, Mikael Björk, and Jörgen Hansson. Empowering configurable QoS management in real-time systems. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 39–50. ACM Press, 2005.
 - [81] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *Proceedings of the 18th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pages 13–26. ACM Press, 2003.
 - [82] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE '04)*, pages 112–121. IEEE Computer Society, 2004.
 - [83] Paolo Tonella and Mariano Ceccato. Migrating interface implementations to aspects. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM '04)*, pages 220–229. IEEE Computer Society, 2004.
 - [84] Tom Tourwé and Kim Mens. Mining aspectual views using formal concept analysis. In *Proceedings of the 4th Workshop on Source Code Analysis and Manipulation (SCAM '04)*, pages 97–106. IEEE Computer Society, 2004.
 - [85] Niklaus Wirth. On the composition of well-structured programs. *ACM Computing Surveys*, 6(4):247–259, 1974.
 - [86] Jan Wloka. Refactoring in the presence of aspects. Presented at the Workshop for PhD Students in Object-Oriented Systems (PhDOOS) at ECOOP '03. In *ECOOP '03 Workshop Reader*, volume 3013 of *LNCS*, pages 50–61. Springer-Verlag, 2004.
 - [87] The AspectJ development tools web site. <http://www.eclipse.org/ajdt>.
 - [88] The AspectJ user mailing list aspectj-users@eclipse.org.
 - [89] The AspectJ web site. <http://www.aspectj.eclipse.org>.
 - [90] The DemeterJ web site. <http://www.ccs.neu.edu/research/demeter/DemeterJava>.

- [91] The Eclipse web site. <http://www.eclipse.org>.
- [92] The JHotDraw project web site. <http://www.jhotdraw.org>.
- [93] The Metrics plugin for Eclipse web site. <http://metrics.sourceforge.net>.
- [94] The MixJuice improvement catalogue web site.
<http://cvs.m17n.org/ akr/mj/design-pattern/en>.
- [95] The Refactoring Browser web site.
<http://st-www.cs.uiuc.edu/users/brant/Refactory>.