

# Aspect-Oriented Refactoring: Classification and Challenges

Jan Hannemann  
University of Tokyo

jan@graco.c.u-tokyo.ac.jp

## ABSTRACT

This paper provides an overview of the three different kinds of AOP refactorings: aspect-aware OO refactorings, refactorings for AOP constructs, and refactorings of crosscutting concerns. We discuss recent developments for each of them and highlight their commonalities with respect to associated challenges and ties to related research, such as program analysis and aspect mining.

## 1. INTRODUCTION

Useful software systems are constantly evolving and changing, and often those changes require that the software be re-modularized, so that the system becomes easier to understand, extend, or maintain [8]. The technique for disciplined program transformations that change a system's structure while preserving its behavior is called refactoring. 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 [5, 8, 19].

While most refactoring research focuses on object-oriented (OO) system transformations, refactorings play a similar role in an aspect-oriented programming (AOP) context. In addition, they are instrumental for the migration of legacy OO systems to use AOP. To distinguish AOP refactorings from traditional OO refactorings, and to differentiate between individual approaches to aspect-oriented refactoring, we will use the simple AspectJ example shown in Figure 1. It shows part of a class for a banking system. The code presented 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.

We use the term traditional refactorings to refer to parameterized, behavior-preserving transformations of object-oriented systems (although refactorings are not limited to OO systems, the majority of research targets that paradigm). Such refactorings usually focus on single program elements or small sets of non-scattered program elements (plus their associated references), and they can often be automated. Despite the limited number of elements targeted, such refactorings may cause program-wide changes. For example, a *Rename Method* refactoring [5] requires all call sites to the targeted method to be updated. Renaming the `deduct(int)` method in the example system represents a traditional refactoring.

## 2. AOP 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

```
public class Account {
    . . .
    public void deduct (int amount)
        throws InsufficientFundsException {
        log("attempting deduction...");
        acquireLock(this);
        try {
            doIt(amount);
        } finally {
            releaseLock(this);
        }
    }

    private void doIt(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.doIt(int)) &&
        target(acct) && args(amount);
    after(Account acct, int amount) returning:
```

Figure 1. A simple banking example

groups as outlined below. Their properties and differences are summarized in Table 1.

1. Aspect-aware OO refactorings, which are traditional object-oriented refactorings that are extended to ensure they do not break AO programming constructs.
2. Refactorings for AOP constructs, which are refactorings explicitly involving AOP program elements.
3. Refactorings of Crosscutting Concerns, which are composite refactorings aiming to transform non-modularized CCCs into aspects.

In the following, we describe each kind of aspect-oriented refactoring in detail.

### 2.1 Aspect-Aware OO Refactorings

Traditional refactorings focus on object-oriented code. When applied as-is to an AOP system, references in AOP constructs are not taken into account, potentially introducing errors.

For instance, applying the *Rename Method* refactoring 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, like the `deduction(..)` pointcut. Similarly, inlining the `doIt(int)` method would require aspect-oriented version of an OO refactoring. This case is more

**Table 1. Overview of AOP refactoring approaches.**

Approach / Technique	Target	Focus / Motivation	Examples
Aspect-Aware OO Refactorings	OO constructs	Ensure OO refactorings update references in AOP constructs properly	Rename/Inline Method etc.
Refactorings for AOP Constructs	OO and AO constructs	Provide new refactorings involving AOP constructs	Replace Member with Intertype Declaration, Pull up Pointcut, etc.
CCC Refactorings	CCC implementations	Provide refactorings to replace non-modular CCC implementations with equivalent aspects	[Replacement of any non-modularized CCC implementation with an aspect]

complex, as the join points identified by the pointcut cease to exist [9].

Research in the area of aspect-aware OO refactorings focuses on extending traditional refactorings with appropriate steps to properly update references in AOP constructs. Making OO refactorings aspect-aware is the topic of several research projects, for example [9, 14, 23].

## 2.2 Refactorings for AOP Constructs

While the aforementioned refactorings target OO constructs, this class or refactorings explicitly involves AOP constructs. Examples include inlining a pointcut and advice body, or replacing an object method with an intertype declaration. Many of these parallel existing OO refactorings: 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 OO refactorings such as *Add Parameter*, *Pull Up Method*, or *Push Down Method* [5] when applied to a pointcut declaration. Similarly, equivalents of *Collapse Hierarchy* or *Extract Super/Subclass* may be conceived for aspects, as shown by Monteiro et al. [18], for example.

In addition, AOP programming constructs allow for a set of new refactorings without OO equivalents, like 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 [14, 18].

Finally, there are low-level object-to-aspect program transformations that ‘move’ small OO code fragments to aspects by replacing them with their aspect-oriented equivalents. For example, the *Replace Implements with Declare Parents* refactoring [18] moves an `implements` declaration to an aspect by replacing it with the equivalent AspectJ construct `declare parents`. Replacing the explicit call to `acquireLock(..)` at the beginning of `deduct(int)` with a corresponding pointcut and advice pair is an example for such a ‘move’ refactoring.

Much like aspect-aware refactorings, refactorings for AOP constructs have a similar scope as traditional refactorings, i.e., they target single program elements or localized sets of them.

## 2.3 Refactorings of CCCs

Refactorings of crosscutting concerns (CCCs) transform scattered CCC implementations into a modularized form (an aspect), which is schematically shown in Figure 2.

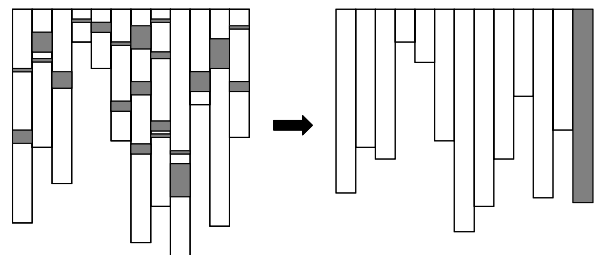
For instance, replacing all code pertaining to preventing race conditions in the example banking system with an equivalent, modular AOP implementation constitutes a CCC refactoring. Or, consider that the account class would incorporate an update mechanism, in which interested objects can register themselves and are notified whenever the account balance is accessed or changes. Such functionality would in OO likely be implemented using the Observer design pattern [7] (not shown). Replacing the code pertaining to this mechanism with an AOP version of the Observer pattern [12] is another example for 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 access to the same data files. These refactorings are generally composite refactorings, consisting of multiple aspect-aware OO refactorings and refactorings for AOP constructs. Currently two approaches with differing tradeoffs exist that support CCC refactorings.

### 2.3.1 Extract-to-Aspect Refactorings

Recently Binkley et al. have developed a partially implemented, human-guided approach to support OO-to-AO refactorings [1], which focuses on the specific problem of extracting OO code fragments into aspects. Their approach relies on code bases in which the code segments pertaining to the implementation of the target CCC have been specially marked. Identifying and marking the CCC code is currently done manually, but the authors plan to employ automated concern mining approaches in future versions.

Their tool supports an interactive aspect extraction workflow 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 tool does not yet create aspect code, but the authors assure it poses no



**Figure 2. CCC refactoring (schematic). A non-modularized CCC implementation is replaced with an equivalent aspect.**

conceptual difficulties.

The discovery phase uses TXL transformation rules to identify and suggest candidates for one of the five supported extraction refactorings (such as the *Extract Beginning/End* refactoring, which replaces a code block at the start/end of methods with appropriate *before/after* advice, respectively). A developer can review the suggestions or perform two different OO transformations (*Statement Reordering* and *Extract Method*) to potentially enable further suggestions. This process of discovery and selection is iterative, and meant to be performed until all marked code is refactored.

In the banking example, consider refactoring the synchronization concern: all code pertaining to acquiring and releasing locks when performing account transactions would initially be marked by the developer. Then the tool would then suggest applicable refactorings (such as suggesting an *Extract End* refactoring for the call to `releaseLock(..)` in the `deduct(..)` method), and the developer can further perform statement reorderings so that *Extract Beginning* applies to the call to `acquireLock(..)`, or move on to the next marked code segment.

Binkley's approach treats CCC refactorings a series of low-level AOP refactorings that are restricted to the replacement of code fragments with pointcut and advice pairs; the overall structure of the CCC and the relationships between extracted code segments is not explicitly taken into account. The approach has very little requirements with respect to the structure of the concern to be extracted, and applies whether the marked code represents a single concern, multiple concerns, or just parts of one concern. On the other hand, the resulting aspect code has thus very little structure to it, consisting merely of an accumulation of pointcut and advice pairs. If more meaningful aspect structures are desired, developers have to manually review and refactor the resulting aspect code, or utilize role-based refactoring, as outlined in the next section.

### 2.3.2 Role-Based Refactorings

Prior to Binkley's work, role-based refactoring (RBR) was proposed as a technique to both capture and refactor CCCs [13]. RBR takes a very different approach: it describes the refactoring based on an abstract model of the target CCC's principal elements (called *role* elements) and their relationships. Role elements represent sets of program elements, such as classes, methods, and fields, which fulfill the same purpose within a concern's implementation. The idea behind RBR is that for all concrete program elements corresponding to a given role element, refactoring instructions are the same. RBR allows for a description of the refactoring specific to the target concern, yet independently from an actual implementation. Refactorings can therefore take the concern structure into account, but can, at the same time, apply to multiple possible implementations of the same CCC.

In the banking example, the logging CCC would be described as consisting of two role methods (e.g., `getLock(..)`, `releaseLock(..)`), and potentially their enclosing type `CurrencyControl` as a role type<sup>1</sup>. Refactoring instructions are then defined in terms of these

---

<sup>1</sup> Names for role elements in a role-based refactoring description are not tied to an actual implementation, but are meant to abstract and describe principal functionality.

elements, one of which would likely be: “apply the *Replace Method Call With Pointcut and Advice*” [10] to calls to the `getLock(..)` and `releaseLock(..)` role methods”. In order to apply the refactoring to a concrete system, like the banking example, a mapping between role elements and program elements in the system is required. For example, by specifying that the concrete method `acquireLock(Account)` plays the role of the role method `getLock(..)`, the refactoring instructions defined on `getLock(..)` can be applied to `acquireLock(Account)`.

Replacing crosscutting OO design pattern implementations with their AOP equivalents (see [12]) has been shown to be a useful application of this technology [13], but the approach is not limited to design pattern refactorings, and can be used to capture and refactor other, non-pattern concerns, such as logging, as well [10]. The RBR approach has been realized in a refactoring plugin for the Eclipse IDE. The structural model of the concern is further useful for determining the concern extent by comparing the abstract concern structure to the structure of the actual program it is applied to, helping to identify the entire concern implementation.

The concern model, while allowing for complex transformations, is also the main limitation of the approach. For example, if the actual implementation varies too much from the expected concern structure (as can be the case with design pattern variants, for example), the refactoring description or the target system may need to be adjusted before the RBR refactoring can be applied. In case that the concern structure and its principal elements are not known to the developer, the extract-to-aspect approach mentioned above is more suitable.

## 3. CHALLENGES

Even though three refactorings have very different purposes, they have many commonalities in technical challenges. This section outlines the challenges and potential ways to address them.

### 3.1 Utilizing Synergy

Some sub-problems are similar to all three areas of AOP refactoring research; all have to transform aspect-oriented programs. There is considerable overlap in the transformations used. For example, refactorings of crosscutting concerns generally consist of several smaller transformation steps, each of which is either an aspect-aware OO refactoring or a refactoring of an AOP construct. But on a very fine-grain level, all refactorings consist of multiple transformation steps. Insights regarding these *elementary* refactorings should therefore be shared.

Ideally, the wheel should not be re-invented. For example, Binkley's *Extract Beginning/End* [1] and Monteiro's *Extract Fragment to Advice* [18] perform basically the same function<sup>2</sup>. Similarly, Hannemann's *Move Object Member to Aspect* [10] is equivalent to Monteiro's *Move Field/Method from Class to Inter-Type* [18] and Iwamoto's *Move Class Method/Field to Advice* [14]. The above example illustrates that the same technique can be assigned multiple names, which can be confusing for AOP refactoring research. In other cases, refactorings have different granularity. For example, Monteiro's *Move Method from Class to Inter-Type* and *Replace Inter-Type Method with Aspect Method*

---

<sup>2</sup> Monteiro's version appears more general, but given the limitation of AspectJ advice, the two are effectively the same.

are combined in Hannemann’s *Replace Object Method with Aspect Method*. A refactoring catalog, like the one proposed by Monteiro [18], can help systematically collecting elementary AOP refactorings, and give researchers an idea what transformations have already been considered, avoiding to duplicate work.

### 3.2 Suggesting AOP Refactorings: Aspect Mining

The refactoring techniques mentioned in Section 2 assume that the developer knows what part of the target system is going to be refactored. If this is not the case, aspect mining approaches can provide a starting point and suggest refactoring candidates by identifying program elements pertaining to the implementation of non-modularized crosscutting concerns. Such mining approaches are either based on textual patterns [22], patterns in execution traces [2], high fan-in methods [16], or duplicated code fragments [3]. A qualitative comparison of aspect mining techniques is provided by Ceccato [4]. The accurateness of mining approaches is very important, because if the concern extent is not correctly identified, the resulting refactoring will be incomplete. Recent approaches utilize or plan to utilize aspect mining to identify candidate sets of code fragments for extraction to aspects [1, 10, 20]. RBR uses an algorithm to help map role elements to concrete program elements. Although this is strictly speaking not an aspect mining technique, it does help to determine the extent of a crosscutting implementation by comparing the structure of the target program and the abstract concern description.

Both CCC refactoring approaches mentioned above can conceivably be extended so that aspect mining techniques can be utilized in future versions. In the case of Binkley’s approach, this is already part of the refactoring model [1]. The output of the aspect mining algorithm can be directly used to avoid manually making the code to be extracted. For the RBR case the information would help choose a proper refactoring from the library of CCC refactorings and provide an initial role mapping.

### 3.3 Preservation of Behavior and Intent

For AOP refactorings there is occasionally a trade-off between behavior preservation and preservation of *intent* of the code. Consider, for instance, inlining the `doIt(int)` method in the banking system shown in Figure 1 (an aspect-aware OO refactoring). This transformation 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. Hanenberg shows how pointcuts can become very complex as part of an aspect-aware behavior-

preserving refactoring [9], which can be seen as a motivation for intent preservation (as opposed to behavior preservation).

### 3.4 User Interaction

Choices in the application of the refactoring and the space of possible aspect-oriented implementations make it necessary to consider 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 its reasoning for later review (e.g., using Java metadata annotations), a complex refactoring (in particular complex CCC refactorings) may require multiple choices, each of which can influence subsequent refactoring steps. Similarly, choosing between multiple refactorings that may apply to the same piece of code requires user interaction.

Currently, both CCC refactoring approaches have their own specific solutions: Binkley’s approach sidesteps parts of the problem by providing only five elementary refactorings, none of which require extensive choices. However, developer interaction is still required in the selection and transformation phase to select applicable refactorings and to perform enabling OO refactorings, respectively. A set of special rules is used to rank possibly applying refactorings. The RBR tool’s approach to user interaction focuses on exploring how such interaction would fit into the refactoring workflow, and the RBR tool currently provides only a few example interactions. For choices related to the role mapping, RBR provides a ranking of mapping candidates based on structural and lexical analyses.

### 3.5 Program Analysis for AOP Refactorings

Modifying an AOP system can have a number of undesired effects (for a partial list, see [11]), if the transformation not performed carefully. Program analyses can help identify the applicability of a refactoring, specifically whether a planned refactoring step has an adverse effect on the behavior or intent of the affected program. This applies both to low-level program transformations and (composite) CCC refactorings. Thankfully, AOP refactorings can utilize techniques for OO program analyses to a large extent.

#### 3.5.1 Resolving Generalizations

Several AOP refactorings involve generalizations. For example, aspect-aware versions of the *Extract Interface* or *Pull Up Member* [5] refactorings, but generalizations can also be included in CCC refactorings like *Extract Interface Implementation* [10]. Tip et al. present an approach for the detection, analysis, and resolution of (object-oriented) generalization refactorings [21] based on type constraints, including updating variable declarations in the course such refactorings. 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 can provide the basis for an equivalent aspect-oriented analysis.

#### 3.5.2 Adapting AOP Systems to Use Generics

Java 1.5 introduced generic types, and AspectJ 5 generic aspects. Adapting an existing system to make use of generics can involve all three kinds of AOP refactorings. The type-constraint-based approach presented by Fuhrer [6] for OO transformations could

be a useful basis for these. If Fuhrer's work would be extended to account for AOP constructs, it can be utilized for all three kinds of AOP refactorings.

### 3.5.3 Classifying Behavior Deviations

An approach presented by Mens et al. formalizes behavior preserving program transformations as basic graph rewriting operations and allows to statically analyze the dependencies between these operations at a semantic and syntactic level [17]. Especially interesting about this approach is the differentiation between various kinds of behavior 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.

### 3.5.4 Composing AOP Refactorings

CCC refactorings consist of series of lower-level AOP refactorings. When taking into account multiple, consecutive program transformations, an analysis of their overall effect is difficult because of potential affects of the individual steps on each other. A promising analysis approach for this problem is provided by Kniesel and Koch [15]. Their generic formal model for the automatic composition of conditional program transformations is program independent and applies to arbitrary conditional program transformations (even non-behavior-preserving ones).

## 4. CONCLUSION

Current research in AOP refactoring focuses on three related problems: extending existing OO refactorings to properly handle references to the affected program elements in AOP constructs, developing new refactorings that explicitly target AOP constructs, and providing refactoring support for the transformation of non-modular CCC implementations into aspects. The first two kinds represent low-level program transformations that can be employed similarly to traditional refactorings, but also serve as building blocks for composite AOP refactorings, such a CCC refactorings.

Research in the area of AOP refactoring is very synergistic, and it is important that the overlap between the different refactoring techniques is not only recognized, but also utilized. Additionally, related work that has been done for object-oriented refactorings can be leveraged to a large extent, as outlined in this paper.

## 5. ACKNOWLEDGMENTS

Thanks to Hidehiko Masuhara for providing valuable comments on earlier drafts of this paper. This work was funded by a JSPS fellowship.

## 6. REFERENCES

- [1] Binkley, D., Ceccato, M., Harman, M., Ricca, F., Tonella, P. Automated refactoring of object-oriented code into aspects. Proc. ICSM '05, pp. 27–36. IEEE Computer Society, 2005.
- [2] Breu, S., Krinke, J. Aspect mining using event traces. Proc. ASE'04, pp. 310–315. IEEE Computer Society, 2004.
- [3] Bruntink, M., Deursen, A., Tourwe, T., van Engelen, R. An evaluation of clone detection techniques for identifying crosscutting concerns. Proc. ICSM '04, pp 200–209. IEEE Computer Society, 2004.
- [4] Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., Tourwe, T. A qualitative comparison of three aspect mining techniques. Proc. IWPC '05, pp. 13–22. IEEE Computer Society, 2005.
- [5] Fowler, M. Refactoring: Improving the Design of Existing code. Addison-Wesley, 1999.
- [6] Fuhrer, R., Tip, F., Kiezun, A., Dolby, J., Keller, M. Efficiently refactoring Java applications to use generic libraries. Proc. ECOOP'05, LNCS vol. 3586, pp. 71–96. Springer, 2005.
- [7] Gamma, E, Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- [8] Griswold, W. Program Restructuring as an Aid to Software Maintenance. PhD thesis, University of Washington, Seattle, WA, USA, 1992.
- [9] Hanenberg, S., Oberschulte, C., Unland, R. Refactoring of aspect-oriented software. NetObject Days '03, Erfurt, Germany, 2003.
- [10] Hannemann, J. Role-based refactoring of crosscutting concerns. PhD thesis, University of British Columbia, BC, Canada, 2005. <http://www.cs.ubc.ca/~jan/>
- [11] Hannemann, J., Chitchyan, R., Rashid, A. Report on the WS on analysis of aspect-oriented software. ECOOP'03 WS Reader, LNCS vol. 3013, pp. 154–164. Springer, 2004.
- [12] Hannemann, J., Kiczales, G. Design pattern implementation in Java and AspectJ. Proc. OOPSLA '02, pp. 161–173. ACM Press, 2002.
- [13] Hannemann, J., Murphy, G., Kiczales, G. Role-based refactoring of crosscutting concerns. Proc. AOSD '05, pp. 135–146. ACM Press, 2005.
- [14] Iwamoto M., Zhao, J. Refactoring aspect-oriented programs. WS on AOSD Modeling With UML at UML '03, 2003.
- [15] Kniesel, G., Koch, H. Static composition of refactorings. Science of Computer Programming, 52:9–51, 2004.
- [16] Marin, M., van Deursen, A., Moonen, L. Identifying aspects using fan-in analysis. In Proc. WCRE '04, pp. 132–141. IEEE Computer Society, 2004.
- [17] Mens, T., Demeyer, S., Janssens, D. Formalising behaviour preserving program transformations. Proc. ICGT '02, LNCS vol. 2505, pp. 286–301. Springer, 2002.
- [18] Monteiro, M. and Fernandes, J. Towards a catalog of aspect-oriented refactorings. Proc. AOSD '05, pp. 111–122. ACM Press, 2005.
- [19] Opdyke, W., Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.
- [20] Shepherd, D. Pollock, L. Ophir: A framework for automatic mining and refactoring of aspects. Technical Report 2004-03, Dept. of Computer and Information Sciences, University of Delaware, DE, USA, 2003.
- [21] Tip, F., Kiezun, A., Bäumer, D. Refactoring for generalization using type constraints. Proc. OOPSLA'03, pp. 13–26. ACM Press, 2003.
- [22] Tourwe, T., Mens, K. Mining aspectual views using formal concept analysis. Proc. WS on Source Code Analysis and Manipulation (SCAM '04), pp 97–106. IEEE Computer Society, 2004.
- [23] Wloka, J. Refactoring in the presence of aspects. WS for PhD Students in Object-Oriented Systems. In ECOOP '03 WS Reader, LNCS vol. 3013, pp. 50–61. Springer, 2003