

Issues in Performing and Automating the “Extract Method Calls” Refactoring

Andy Kellens* and Kris Gybels†
Programming Technology Lab
Vrije Universiteit Brussel
{andy.kellens,kris.gybels}@vub.ac.be

February 21, 2005

1 Introduction

A large part of converting a pre-AOP application into an aspect-oriented one consists, next to *aspect mining*, out of *aspect refactoring*. One such aspect refactoring named “Extract Method Calls” refactoring [11, 17] transforms certain crosscutting concerns which are implemented by calls to a single method into an advice by turning the called method into an advice and all the calls to that method into a crosscut.

In this paper we present the *Joinpoint Choice Problem* and the *Flattened Expression Problem*. Both problems are inherent to the joinpoint models as used by current-day weavers and can have a negative impact on the maintainability and evolvability of the crosscuts resulting from performing the “Extract Method Calls” refactoring either by hand or automatically. As a solution to the *Joinpoint Choice Problem*, we propose an algorithm for calculating the appropriate crosscut. In order to solve the *Flattened Expression Problem*, we introduce a new language construct named the “statement joinpoint”.

In section 2 we describe the “Extract Method Calls” refactoring. Section 3 we present the *Joinpoint Choice Problem* and *Flattened Expression Problem*. We discuss solutions for these problems in section 4. After presenting some related work and research directions for the future (section 5),

*Author funded by a doctoral scholarship of the “Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen)”

†Research assistant of the Fund for Scientific Research - Flanders, Belgium (F.W.O.)

we conclude this paper in section 6.

2 “Extract Method Calls” Refactoring

In this section we follow the pattern established by Fowler [2] to describe the “Extract Method Calls” refactoring.

Typical Situation and Motivation

When not using AOP technologies, developers can deal with certain crosscutting concerns by implementing them as calls to a single “global” method. We’ve dubbed such methods “unique methods” [6] and examples of crosscutting concerns that are typically implemented using “unique methods” are logging, observer updating, authorization checking etc.: they can be implemented as a method `log:`, `update:` or `authorize:` that is called before or after other messages as appropriate. The concern may have been directly implemented in this way, or the situation may have arisen through subsequent applications of the “Extract Method” refactoring [2], up to the point where non-aspect-oriented refactorings can no longer improve the code, leaving the calls to the method as evidence of the concern’s tangling [17, 11].

To further improve the readability, maintainability and other “-ilities” of the code, the method should be turned into an advice. This requires that all calls to the method are removed and turned into the crosscut of the advice.

```

BankAccount>>withdraw: amount
  TheLogger log: 'Client updates balance'.
  self balance: self balance - amount.
  TheLogger log: 'Client updated balance'.

BankAccount>>deposit: amount
  TheLogger log: 'Client updates balance'.
  self balance: self balance + amount.
  TheLogger log: 'Client updated balance'.

Logger>>log: message
  Transcript show: message

```

Figure 1: Example of a scattered logging call.

```

before ?jp matching {
  ?jp isReceptionOf: #withdraw: ||
  ?jp isReceptionOf: #deposit:
} do
  TheLogger log: 'Client updates balance'.

after ?jp matching {
  ?jp isReceptionOf: #withdraw: ||
  ?jp isReceptionOf: #deposit:
} do:
  TheLogger log: 'Client updated balance'.

```

Figure 2: Resulting advices for the example refactoring

Example

Figure 1 shows an example non-AOP way of implementing logging¹: a `log:` method is defined on the object contained in the global variable `TheLogger`, and called from both of the methods `withdraw:` and `deposit:` right before and after a change is made to the balance. The change is made using accessor and mutator messages.

To refactor the logging, the calls to the `log:` method can be turned into two advices as shown in figure 2. In the first advice we’ve encapsulated the

¹Examples are given in Smalltalk and the logic-language-based crosscut language CARMA [4]. The version of CARMA used here features a keyword-based syntax similar to that of Smalltalk for logic functors and conditions, logic variables are written with question marks (e.g. “?x” and “?jp” are logic variables) and logic lists are written with angular brackets (“<” and “>”).

logging of the balance being about to be changed, while the second encapsulates the logging after the change has happened. In the crosscuts of both of the advices we’ve exploited the fact that the logging calls occur at the start and end of the methods respectively, thus by weaving before and after the message reception joinpoints of `withdraw:` and `deposit:` the logging is guaranteed to occur at the exact same point in the execution of the code as before, ensuring that the refactoring is behavior-preserving.

Related Refactorings

The purpose of this refactoring is only to extract the calls of a certain method, and put these in an advice, while the method itself remains where it is. This refactoring can be combined with “Move Method” or “Inline Method” [2] to fully separate both the calls and the method itself into an aspect. Using “Inline Method” in the example would allow us to remove the `log:` method altogether by changing the `TheLogger log:` messages in the advices to `Transcript show:` messages. The combined refactoring is a true “Method to Advice” refactoring.

Preconditions

In order for this refactoring to be applicable, the calls should not invoke a method that returns a value, or in other words the calls should occur as statements rather than be part of expressions. In the latter case the method clearly implements a part of the base functionality of the program, or is part of a more tangled implementation of a concern that should be cleaned up using the “Extract Method” refactoring [2] first.

Though the typical situation for applying this refactoring is the extraction of “unique methods”, the existence of only one method that is the unique implementer of the message sent by the extracted calls is not an absolute requirement for this refactoring. However, in order for the “Inline Method” refactoring to be applicable as well, the method that is invoked by the calls should at least be statically identifiable [2], which is obviously the case when the method is the sole implementer of the message.

In previous work [17, 7] other preconditions were taken into account that are a result of limitations of the AspectJ language. If the calls to extract rely for example on local variables, the crosscuts should expose these variables to the advice, which is not possible in AspectJ. Such preconditions are however dependent on the aspect and crosscut language used and should not be strongly featured in a language-neutral description of this refactoring.

Mechanics

Our description of the mechanics of this refactoring closely follows that of Laddad [11], who distinguished three major steps to perform this refactoring manually:

Introduce a no-op refactoring aspect: in the first step, a new aspect is introduced which has the necessary crosscuts but with empty advices. To avoid unwanted effects due to interception of the wrong joinpoints, mechanisms such as wildcarding should not be used to write the crosscut in this step, the crosscut should rather simply enumerate the locations of the to-be-extracted calls. In Laddad’s work only calls occurring at the beginning or end of methods were considered, in which case the crosscut can be written by writing a reception joinpoint condition for each of the methods and combining these using the “or” operator, as done in the logging example for the `withdraw:` and `deposit:` methods. This gets more complex when the calls can also occur at other locations in methods, as will be discussed further on in this paper. After writing the crosscut, one can and should use IDE support [1] to check that the crosscut actually captures the right joinpoints.

Introduce the crosscutting functionality: this step consists of actually moving the calls to the advice(s) and crosscut(s) introduced in the previous step. As indicated by Hanenberg et al. this may require the addition of “context exposing” conditions to the crosscuts [7], for example when the calls rely on instance variables or the “self” (= “this”) variable: suppose that in our example the `log:` messages were sent to `self` instead of `TheLogger`, moving these calls to an advice requires the crosscuts to include a condition `?jp inObject:`

`?object`, so that the advice can send the `log:` message to the object in the `?object` variable.

Make crosscuts pattern-based: while Laddad indicates this as an optional step [11], it is in fact critical to simplify the enumeration-based crosscuts from the previous steps to avoid the well-known maintainability problems associated with such crosscuts [4]: an enumeration-based crosscut is too tightly coupled to a specific version of the base program and may require adaptation whenever the base program is changed. To avoid this problem, the crosscuts should be turned into pattern-based ones: a pattern-based crosscut exploits “patterns” or “commonalities” in the joinpoints that are to be crosscut, thus automatically capturing new joinpoints that match this pattern when the base program is changed.

Suppose that in our example, the `withdraw:` and `deposit:` methods are implemented in a protocol² “balance updating”. Further supposing that these methods are the only ones in that protocol, we could change the crosscuts to:

```
?jp isReceptionOf: ?msg &&
?msg isImplementedInProtocol:
    "balance updating"
```

Doing so is a likely to be a good generalization of why logging occurred in the two methods: new methods added to the same protocol later on will likely also require logging, with the above crosscut this is done automatically.

3 Problems

In the previous section we’ve given an overview of the “Extract Method Calls” refactoring for a simple example. Current literature on this refactoring also only takes simple examples into account [11, 17, 7, 13]. There are however a number of problems that crop up with more complex examples, such as when the calls to be refactored do not occur at the beginning or end of methods. In this section we discuss two complexities that can arise when performing the mechanics of the refactoring.

²A method categorization mechanism provided by Smalltalk systems, Java programmers can think of these as annotations.

```

BankAccount>>withdraw: amount
self checkPositive: amount.
TheLogger log: 'Client updates balance'.
self balance: self balance - amount.

```

```

BankAccount>>deposit: amount
self checkPositive: amount.
TheLogger log: 'Client updates balance'.
self balance: self balance + amount.

```

```

BankAccount>>addInterests
self balance: (self balance +
               self calculateInterests).

```

Figure 3: Example of a scattered logging call.

```

before ?jp matching {
  ?jp isSendOf: #balance: &&
  ( ?jp isWithin: #deposit: ||
    ?jp isWithin: #withdraw: )
} do
  TheLogger log: 'Client updates balance'.

```

Figure 4: Possible crosscut to log all updates to the balance done by a client

These complexities arise both when the refactoring is performed manually as well when it is to be automated, though they more severely hinder the automation of the refactoring.

Example

Figure 3 shows a more complex variation of the example given in figure 1. The complexity arises because part of the logging now occurs mid-method instead of at the beginning of a method. This introduces problems in applying the mechanics of the refactoring as will be discussed next.

Mechanics: Flattened Expression Problem

One might be tempted to write the crosscut in the “introduce no-op aspect” step of the refactoring as in figure 4. However, this does not actually result in a behavior-preserving refactoring. In the

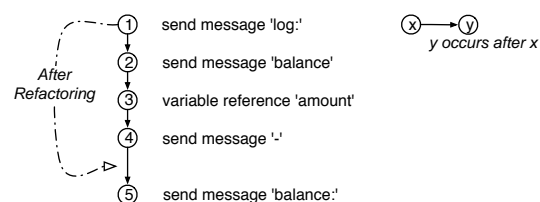


Figure 5: Joinpoint graph of example before and after refactoring using crosscut from figure 4

```

BankAccount>>withdraw: amount
| newbalance |
self checkPositive: amount.
newbalance := self balance - amount.
TheLogger log: 'Client updates balance'.
self balance: newbalance.

```

Figure 6: Alternative implementation of logging invocation

example the execution order of the messages - and thus the joinpoints - starting from the `log:` message is as depicted in figure 5: first the message `log:` is sent, then the message `balance` is sent, the variable `amount` is looked up, the message `-` is sent³ and finally the message `balance:` is sent. If we were to apply the refactoring with the above crosscut however, the execution order of the messages changes as also depicted in figure 5: the `log:` message will be sent after the `-` message, right before the `before:` message.

While it is reasonable to assume in this example that moving the actual logging closer to the execution of the `balance:` message makes no observable difference in the behavior of the program, and is actually closer to what was really the intention of the logging, this can not be generally assumed, especially when the refactoring is automated. The problem lies in a difference between the dynamic joinpoint model and the static one as necessarily applied by programmers when not using AOP technology to implement crosscutting concerns: while the logging should really occur right before the `balance:` operation, this requires either implementing a special `logBalance:` method

³-, +, * etc. are really messages in Smalltalk, and not special arithmetic operations as in some other languages, thus they also lead to message joinpoints.

or writing code that computes the argument of the `balance:` message to a temporary variable first, then does the logging and only then sends the `balance:` message as illustrated in figure 6. Either option only further increases the clutter caused by the logging aspect, and the programmer might as well resort to writing the `log:` statement right before the `balance:` statement.

The problem for the refactoring is however that statements become flattened in current dynamic joinpoint models as used by most crosscut languages, making it difficult to write a crosscut that is both concise and ensures the behavior-preserving of the refactoring. In the example, to ensure that the refactoring is behavior-preserving, meaning that the logging occurs in the exact same execution order as before the refactoring, the crosscut should in the `deposit:` method capture the `balance` message joinpoint: the first joinpoint in the flattened list of joinpoints that result from the `self balance: self balance + amount` statement. This on the other hand leads to several problems with making the crosscut pattern-based: there may be several other `balance:` statements before which the same logging occurs, but where the argument and thus the joinpoint to weave on is completely different, and there may be no common pattern in these. Even if there's a pattern there that can be used to make the crosscut pattern-based, this pattern wouldn't actually reflect the intended semantics of the original implementation of the concern.

Mechanics: Joinpoint Choice Problem

Another problem that becomes especially problematic when automating the refactoring is that something like logging in our example (figure 1) can be there because of what happens before it, or what happens after it. For each `log:` message to be extracted, this gives two possible joinpoints to weave on in combination with either a “before” or an “after” advice. In the example it is pretty clear to a human reader that the logging has something to do with the updating of the balance, and not with the check that happens before it of the amount being positive, this is simply clearly suggested by the string that is being logged. For an automated tool that performs the refactoring, this would not be as

simple.

4 Solutions and Automation

The problems discussed in the previous section are mostly problematic because they hinder the further maintainability and evolvability of the resulting refactored aspect which might have a crosscut that reinjects calls at the right points but is not semantically an actual reflection of what was originally intended. This is especially a problem when we consider the automation of the refactoring. A possible solution we propose for the first problem is comparatively simple and consists of adding a “statement joinpoint” to the joinpoint model, the proposed solution for the second problem ties in with our approach to automating the “make crosscut pattern-based” step of the refactoring using Inductive Logic, which we will therefore discuss first, followed by a discussion of the two solutions.

4.1 Automating the Refactoring

Introduce a no-op refactoring aspect & the crosscutting functionality: The first two steps of the refactoring are fairly easy to automate. The moving of the calls to the advice in the second step is a code transformation akin to the “Move Method” refactoring [2]. The generation of a purely enumerative crosscut in the first step is straightforward as well, if we disregard the “joinpoint choice” problem for a moment and just pick any of the possible joinpoints for re-injecting each extracted call: for any set of joinpoints, generating an enumerative crosscut simply consists of including enough conditions in the crosscut to ensure that exactly that set of joinpoints is picked out by the crosscut. The latter can be done as illustrated in figure 4 by generating a crosscut that ORs together conditions for each of the joinpoints that exactly identify that joinpoint⁴. The condition for each joinpoint should in turn simply be an AND of crosscut conditions that draws on *background data* that is known about the joinpoint: its type, name of the message that is sent in case of message send

⁴Exact identification is not really necessary, as long as the final crosscut captures no joinpoints that should not be captured.

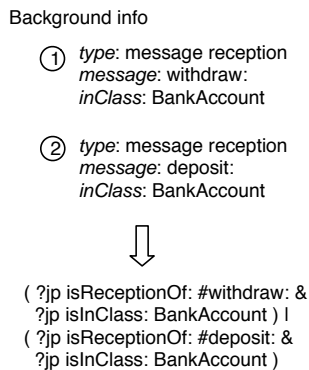


Figure 7: Illustration of generating an enumerative crosscut

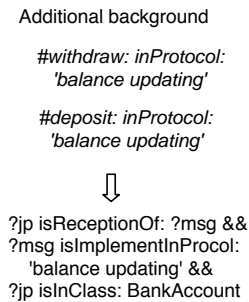


Figure 8: Illustration of generating an intensional crosscut

joinpoints, type of arguments, the joinpoint's lexical extent etc.

Make crosscuts pattern-based: The third step consisting of making the crosscut pattern-based is more difficult to automate, partly because this step deals with making the crosscut better reflect the intended semantics of the aspect. Nevertheless, the purely technical mechanism of making the crosscut more concise and pattern-based can likely be dealt with using techniques from the field of machine learning and data mining [12], which provide techniques for finding patterns in sets of data. We've previously already reported on initial experiments using the Logic Induction algorithm [5], but other techniques such as Concept Analysis will likely be useful as well [3].

It is beyond the scope of this paper to give any-

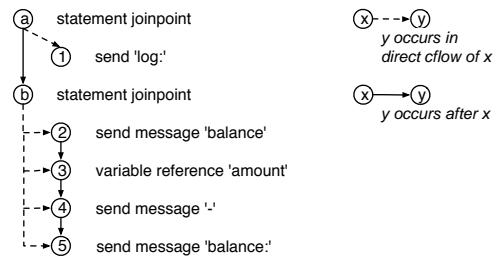


Figure 9: Joinpoint graph as in figure 5 but including statement joinpoints

thing but a brief overview of Logic Induction, but we can informally explain the algorithm's working in our example. Reduced to its essence, the LI algorithm introduces wildcards into the conditions of a crosscut to make these more general. For example in the crosscut from figure 4, it could generalise the conditions `?jp isReceptionOf: #withdraw:` and `?jp isReceptionOf: #deposit:` into a single condition `?jp isReceptionOf: ?msg.` Of course this makes the crosscut *too* general, it would capture joinpoints that it shouldn't. But an important point to grasp about LI is that the wildcards it introduces are *logic variables* and that it includes steps to restrain the crosscut again by adding conditions that restrict the value of those variables. The latter is done by drawing on additional background information on the background information (!) for the joinpoints, as illustrated in figure 4 where the information is provided that `#withdraw:` and `#deposit:` are messages implemented in the protocol `'balance updating'`, since these are the two values the variable `?msg` was substituted for the condition `?msg isImplementInProcol: 'balance updating'` can be added. Note that LI follows steps similar to what we described in section 2 for manually making the crosscut pattern-based.

4.2 Statement Joinpoints

The simplest solution to bridge the gap between the static and dynamic joinpoint model which causes the flattened expression problem is to extend dynamic joinpoint models with statement execution joinpoints. Such a statement joinpoint would best be modeled as having a cflow-relationship with all the joinpoints caused by its execution, e.g. in figure

5 joinpoint 1 would then be a joinpoint occurring in the cflow of a statement joinpoint, and joinpoints through 2 to 5 would all be in the cflow of another; this is illustrated in figure 9.

There is however a question of what data should be associated with a statement joinpoint and what predicates to offer to specify conditions on statement joinpoints in the crosscut language. For solving the “Flattened Expression” problem, it is only necessary to allow crosscuts to capture statement joinpoints based on the type and any other *static* data associated with the *last* joinpoint in its cflow: this is sufficient for expressing crosscuts as necessary in our refactoring example where the `log:` message needs to be injected before the statement joinpoints which have a `balance:` message joinpoint as the last joinpoint in their cflow.

An issue with associating only static data of the last joinpoint with the statement joinpoint is however that it seems rather arbitrary from a language design standpoint. It would be more appropriate to design the crosscut predicates for statement joinpoints such that the crosscut in the following advice would be possible:

```
before ?jp matching {
  ?jp isStatementJoinpoint &
  ?jp hasInDirectCflow: ?jp2 &
  ?jp2 isSendOf: #foo: with: <10>
} do
  TheLogger log: '?'
```

Such a design would orthogonally integrate statement joinpoints with the rest of the crosscut language by adding only an `isStatementJoinpoint` predicate and a `hasInDirectCflow:` predicate. The latter predicate however introduces the problem of having to know about the dynamic future of a joinpoint: for the example advice above, whether it should be executed when a statement joinpoint is reached depends on whether the last message in the statement is `foo:` and whether the argument of that message is *going to be* 10, which is an obvious problem for actually weaving the advice. It would be possible to simply associate only statically-known information with joinpoints such as `?jp2` in the example and report an error when crosscut predicates that depend on knowing dynamic information are used on such joinpoints, but this again breaks the orthogonality of the language

design. Properly providing support for statement joinpoints, and the whole issue of predictive joinpoints [10], may actually require a rethinking of currently available crosscut languages.

Another issue with the statement joinpoints is that their use may need to be discouraged for any other use besides ensuring the behavior-preservation of the “Extract Method Calls” refactoring. The whole underlying idea of using a dynamic rather than a static joinpoint model is to make crosscuts less dependent on syntactical elements and more on semantical events of a program, so as to prevent maintainability and evolvability problems. Whether statement joinpoints would cause such problems and thus crosscuts using them as a result of a refactoring should eventually be ‘cleaned up’, or whether statement joinpoints are a more generally useful addition to dynamic joinpoint models remains to be evaluated.

4.3 Uncovering Crosscuts

A potential solution for the joinpoint choice problem, that especially arises when fully automating the refactoring, may be to rely on how well the different choices of joinpoints lead to pattern-based crosscuts. This relies on the hypothesis that there is typically a pattern to be found in the set of the semantically “right” joinpoints and none or only complicated ones in the set of the “wrong” ones: for example, when our example financial application logs all ‘balance updating’ operations before they are executed, there is definitely a pattern in the set of statement joinpoints that follow a logging message in that they invoke operations implemented in a ‘balance updating’ protocol for example, while the statement joinpoints that precede the message will likely be more “random”. Thus it might be possible to generate an enumerative crosscut for both the set of “before” and the set of “after” joinpoints for the calls to be extracted, and run both crosscuts through an algorithm for making them pattern-based, after which the two should be compared to see which of two has actually been made more pattern-based than the other. The latter likely reflects the real semantics of *why* the calls occurred at the place in the code they originally did.

Of course this potential solution relies on a number of tentative hypotheses, not least of which is

that there is a way to compare crosscuts to see which is “better”. More generally we can state that there currently does not seem to be a good way to evaluate crosscuts to help programmers decide whether they might form a future maintainability problem. There are only a few guidelines known such as that crosscuts should not depend on specific names of a program. This may thus form a good topic of discussion for the SPLAT workshop: how to evaluate the software-engineering properties not of crosscut languages, but of crosscuts themselves? How does one tell if a crosscut is more pattern-based than another?

5 Future and Related Work

Despite the problems we’ve outlined here with uncovering of pattern-based crosscuts for the “Extract Method Calls” refactoring, a few early experiments have indicated positive results for automating this step of the refactoring using machine learning techniques. In a previous paper on our logic crosscut language, an example of a very strongly pattern-based crosscut was given [4]: a crosscut using a description of “state changing” methods based on the pattern of variable assignments occurring in the methods themselves or in the other methods they invoke as needed for an “observer” aspect. We later performed an experiment where a logic description of such methods was automatically derived by using Logic Induction [15]: methods that were state changing were used as input for a Logic Induction algorithm, together with background information on those methods such as their complete body as well as the same information on methods that were not state changing, the output of the algorithm was a logic query describing those methods in a pattern-based way similar to the manually written crosscut. However more experiments need to be performed on the practical scalability and applicability to crosscut uncovering in large case studies.

The problem of capturing the intent of a crosscutting concern versus behavior-preservation of the refactoring is also recognized by Hannemann et al. [8]. In their approach this is resolved by relying on interaction with the developer. As one of the major cases where this problem arises is due to mismatches between the static and dynamic joinpoint models we suggest adding new crosscut language

constructs such as the statement joinpoint.

Aspect-Oriented Programming of course introduces a number of other possibilities for refactoring besides the “Extract Method Calls” refactoring and also necessitates a review of the object-oriented refactorings to take aspects into account [7, 9, 16, 13, 17, 14]. A number of the works we cite here also describe the “Extract Method Calls” refactoring though they do not all follow the pattern for describing refactorings as established by Fowler [2]; we’ve added onto the existing descriptions of how to manually perform the refactoring by identifying “unique methods” as a typical situation for applying it. The problems we discussed that arise when generalizing the refactoring to calls that do not necessarily occur at the start and end of methods and our proposal for automating the crosscut generation step of the refactoring are to our knowledge not treated by other authors.

6 Conclusion

In this paper we took an in-depth look at the “Extract Method Calls” refactoring. The purpose of this refactoring is to transform crosscutting concerns implemented by scattered calls to a single method into a corresponding crosscut and advice. We identified two problems that occur when considering calls which are not necessarily implemented at the start or end of methods and which complicate the process of writing a crosscut that is both behaviour-preserving and covers the intent of why the original calls occurred: the *flattened expression problem* and the *joinpoint choice problem*. These problems become especially pronounced when we consider automated application of the refactoring. As the first problem is often due to a mismatch between the “static joinpoint” model inherent to non-AOP implementations of concerns and the dynamic joinpoint models currently employed in AOP we propose the addition of the *statement joinpoint* to bridge the mismatch. We’ve discussed our proposal for using machine learning techniques for automating the step of the refactoring which involves making crosscuts pattern-based, which may also help solve the joinpoint choice problem by evaluating how well the different choices of joinpoints lead to pattern-based crosscuts. While further research is needed to investigate the full applicability

of this proposal, early experiments have indicated that automating the pattern-based crosscut uncovering step of the “Extract Method Calls” refactoring is feasible.

References

- [1] A. Colyer, A. Clement, and M. Kersten. Aspect-Oriented Programming with AJDT. In *Workshop on Analysis of Aspect-Oriented Software, ECOOP2003*, 2003.
- [2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [3] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.
- [4] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the Second International Conference of Aspect-Oriented Software Development*, 2003.
- [5] K. Gybels and A. Kellens. An experiment in using inductive logic programming to uncover pointcuts
an experiment in using inductive logic programming to uncover pointcuts. In *First European Interactive Workshop on Aspects in Software*, 2004.
- [6] K. Gybels and A. Kellens. Experiences with identifying aspects in smalltalk using ‘unique methods’. In *Workshop on Linking Aspect Technology and Evolution (submitted)*, 2005.
- [7] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World*, 2003.
- [8] J. Hanneman, G. C. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In *Fourth International Conference on Aspect-Oriented Software Development*, 2005.
- [9] M. Iwamoto and J. Zhao. Refactoring aspect-oriented programs. In *Fourth AOSD Modeling with UML Workshop*, 2003.
- [10] G. Kiczales. The fun has just begun. Keynote at AOSD2003.
- [11] R. Laddad. Aspect-oriented refactoring, dec 2003.
- [12] T. Mitchell. *Machine Learning*. McGraw-Hill International Editions, 1997.
- [13] M. P. Monteiro. Catalogue of refactorings for aspectj. Technical Report UM-DI-GECS-200401, Universidade Do Minho, 2004.
- [14] S. Rura. Refactoring aspect-oriented software. Master’s thesis, Williams College, Massachusetts, 2003.
- [15] T. Tourwé, J. Brichau, A. Kellens, and K. Gybels. Induced intentional software views. Submitted to European Smalltalk Users Group conference 2003.
- [16] T. Tourwé, A. Kellens, W. Vanderperren, and F. Vannieuwenhuysse. Inductively generated pointcuts to support refactoring to aspects. In *Workshop on Software Engineering Properties of Languages for Aspect Technologies*, 2004.
- [17] F. Vannieuwenhuysse. Aspect-oriented refactoring. Licentiate’s thesis, Vrije Universiteit Brussel, june 2004.