

Automated Pointcut Extraction

D. Binkley⁽¹⁾, M. Ceccato⁽²⁾, M. Harman⁽³⁾, P. Tonella⁽²⁾

(1) Loyola College, Baltimore, MD, USA

(2) ITC-irst, Trento, Italy

(3) King's College London, UK

binkley@cs.loyola.edu, ceccato@itc.it, mark@dcs.kcl.ac.uk, tonella@itc.it

Abstract

Software refactoring consists of the modification of internal program structure without altering the external behavior (semantic preservation). It aims at improving internal quality factors, such as modularity, in order to make the code easier to understand and evolve in the future. Among the various refactorings, a category quite unexplored is refactoring Object Oriented Programming (OOP) to Aspect Oriented Programming (AOP). AOP is expected to improve the structure of existing code by offering modular units for functionalities whose implementation is otherwise scattered through many modules.

An investigation of automated support for a programmer in the migration from OOP to AOP code is presented. In particular, the applicability of semantic-preserving code transformations to automate the migration task is considered. The contribution of this work is a list of refactorings, that are described together with the conditions under which they can be applied. Several variants of each refactoring are also considered. Supporting the refactoring activity by means of automated tools allows migration difficulties to be mitigated and thus the benefits of AOP become easier to achieve for legacy OOP applications.

Keywords: refactoring, program transformations, aspect extraction.

1 Introduction

Aspect Oriented Programming provides explicit constructs for the modularization of the *crosscutting concerns*: functionalities that traverse the principal decomposition of an application and thus cannot be assigned to a single modular unit in traditional programming paradigms. Existing software often contains several instances of such crosscutting concerns such as persistence, logging, caching,

etc. Consequently, refactoring of these applications towards AOP is beneficial, as it separates the principal decomposition from these other functionalities, by modularizing the crosscutting concerns.

The process of migrating existing software to AOP is highly knowledge-intensive and any refactoring toolkit should include the user in a change-refine-loop. However, there is considerable room for automation in two respects:

- aspect mining – identification of candidate aspects in the given code [2, 17, 20, 22], and
- refactoring – semantic-preserving transformations that migrate the code to AOP [9, 23].

This paper focused on the second problem. Moreover, it considers the difficult part of refactoring: the determination of good and effective pointcuts to intercept the execution and redirect it to the aspect code. Such a process must be guaranteed to preserve the original behavior, while modularizing the code that pertains to the crosscutting functionality. Our approach to this problem derives from the field of program transformations and amorphous slicing [12, 13]. Its outcome entails one or more aspects containing the crosscutting code, with pointcuts able to redirect the execution whenever necessary. Manual refinement of such an outcome, for example to generalize the pointcut definitions, remains an advisable step. Overall, the costs associated with the refactoring activity to migrate OOP to AOP is expected to be greatly reduced through automation.

The paper is organized as follows: the next Section gives some background on the AOP language AspectJ, used in the examples of the paper. It can be safely skipped by readers who are already familiar with this language. Section 3 presents the refactorings that are required for the automated generation of pointcuts, in the migration of an existing OOP application to AOP. Finally, Sections 4 and 5 describe the related work, followed by future work and conclusions.

2 Background on AspectJ

Among the programming languages and tools that have been developed to support AOP, AspectJ [16], an extension of Java with aspects, is one of the most popular and best supported. The two main new programming constructs provided by AspectJ for the definition of the behavior of an aspect are *pointcuts* and *introductions*.

Pointcuts identify *join points* in a program: points in the control flow where execution can be intercepted by an aspect to alter the original behavior of the code. For example, if a persistence aspect is defined to serialize all objects of class *Person* as soon as they are created, an appropriate pointcut can be used to intercept calls to any constructor of class *Person*. In AspectJ, this looks like

```
aspect PersistentPerson {
    pointcut personCreation(String data):
        call(Person.new(String)) && args(data);
    after (String data): personCreation(data)
        { // save Person data to db }
}
```

Pointcuts can also include the dynamic context of another pointcut (e.g., in AspectJ `cflow(move())` identifies all join points that occur between receiving method calls for the methods in `move` and returning from those calls, either normally or by throwing an exception.)

The statements to be executed after (alternatively, “before” or “instead of”) a given join point are defined inside a so-called *advice*. In the example, an advice executed after the pointcut *personCreation* saves information about the *Person* object being created into a database. The AspectJ keyword `args` is used to expose the `String` parameter of the constructor, making it available inside the after-advice.

While pointcuts operate dynamically, introductions aim at altering the static structure of an Object Oriented program. Introductions can be used to change the inheritance hierarchy, by modifying the super-class or interfaces of a given class. Moreover, they can be used to insert new members (attributes or methods) into a class.

For example, it is possible to declare that a class implements a given interface inside an aspect, while the original code does not do it. In such a case, the methods required by the interface must be introduced as well.

With reference to the persistence aspect, it is possible to declare that the interface *Serializable* is implemented by class *Person* and to introduce the related methods, by means of the following code fragment:

```
aspect PersistentPerson {
    declare parents: Person implements Serializable;
    private void
        Person.readObject(ObjectInputStream in)
        {...}
    private void
        Person.writeObject(ObjectOutputStream out)
        {...}
}
```

3 Refactorings for pointcut extraction

In the following, a set of program transformations is described that allow the automated extraction of pointcuts. The input to such transformations is the source code, with fragments to be migrated to aspects already identified. This means that aspect mining (i.e., identification and manual validation of candidate aspects in the original code) is assumed to have already taken place and that the code fragments to be aspectized are known [2, 17, 20, 22]. Such code fragments break down into three cases:

1. whole methods,
2. calls to methods, or
3. arbitrary sequences of statements.

This paper focuses on call extraction (Case 2). In other words, it provides definitions for *pointcuts* and *advices* that replace method calls. Method extraction (Case 1) is straightforward and requires simply moving the whole method from a class to an aspect, where it is turned into an introduction. When a sequence of statements (Case 3) has to be aspectized, it is more convenient to first apply the standard Object-Oriented refactoring *Extract method* [6] to turn the statement sequence into a separate method and then to apply method extraction (Case 1) and call extraction (Case 2).

In the remainder of this paper, a single method call to be extracted is considered. When there are multiple calls, and correspondingly multiple pointcuts, to be associated with the same advice, it is sufficient to define a new pointcut as the logical *or* of the given pointcuts. For example, if there are N calls to method `g()` inside methods `A.f1()`, ..., `A.fN()`, the pointcuts `p1()`, ..., `pN()` can be extracted to intercept each call. Then, the pointcut `p()` can be defined as `p1() || ... || pN()` to intercept all of them.

Based upon the authors’ experience with (manual) refactoring towards AOP, the three main cases to deal with are those considered in the next three subsections. Each is accompanied by an example written in AspectJ. The basic ap-

plicability conditions for the three refactorings are as follows:

1. The call to be moved to the aspect is at the beginning of the body of the calling method.
2. The call to be moved is always before another call.
3. A conditional statement controls the execution of the call to be moved to the aspect.

Variants of these conditions are also described.

3.1 Refactoring 1: Call at the beginning/end

The first refactoring deals with the following case:

The call to be moved to the aspect is at the beginning of the body of the calling method.

Figure 1 shows the mechanics of this refactoring. The call to method `g` is removed from the body of `f`. A new aspect, named `B`, is responsible for intercepting the execution of `f` and reinserting the call to `g` at the beginning.

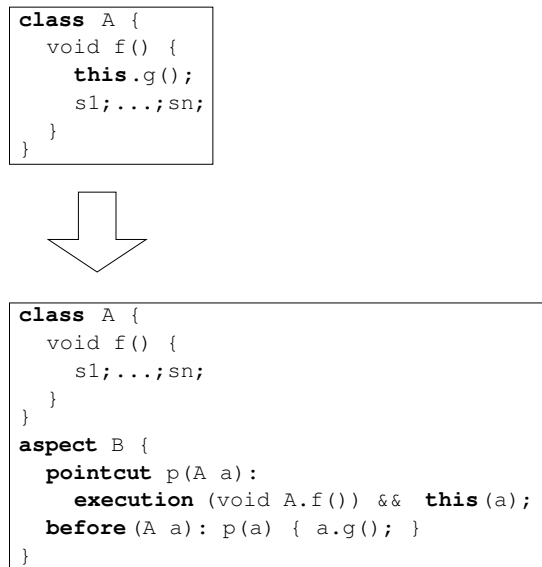


Figure 1. Mechanics of Refactoring 1 (call at the beginning).

3.1.1 Variants

1. The call `this.g()` is the last statement of `f()`. In this case, it is sufficient to replace the before-advice with an after-advice.

2. The call `x.g()` is present instead of `this.g()`, with `x` a class field. In this case, it is sufficient to replace `a.g()` with `a.x.g()` in the before-advice.
3. The call `x.g()` is present instead of `this.g()`, with `x` a parameter of `f`. In this case, it is sufficient to add `args(x)` to the pointcut to expose the argument and then use `x.g()` in the advice.
4. Although `g()` is not at the beginning (end) of `f()`, it is possible to apply Refactoring 1 by first applying semantics preserving re-ordering transformations. For example, Figure 2 shows a reordering rule that allows two statements (st_1 and st_2) to be interchanged. Above the line in the rule is a pre-condition required for the interchange. The given pre-condition is rather restrictive. It requires that the defined and referenced variables of the two statements not overlap. As discussed in Section 5, future work will study less restrictive pre-conditions.
5. Finally, if statement re-ordering transformations cannot move the call to the beginning/end of the method body, but they can move the call to the beginning/end of a sequence of statements, then it is possible to first turn the statement sequence into a method (*Extract method*) and then apply Refactoring 1.

3.2 Refactoring 2: Call before/after another call

The second refactoring deals with the following case:

The call to be moved is always before another call.

Figure 3 shows the mechanics of this refactoring, under the assumption that in the presence of more than one call to `h()`, `g()` precedes `h()` in all of them. The aspect `B` intercepts only the calls to `h` that occur within the execution of method `f` (e.g., using AspectJ's `cflow` construct). A before-advice reintroduces the call to `g` at the proper execution points.

3.2.1 Variants

1. `g()` follows the call `h()`, instead of preceding it. In this case, it is sufficient to replace the before-advice with an after-advice.
2. The call `x.g()` is present instead of `this.g()`, with `x` a class field. In this case, it is sufficient to replace `a.g()` with `a.x.g()` in the before-advice. (This is Variant 2 of Refactoring 1.)

Rule 1 (Statement reordering)

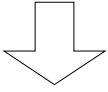
$$\frac{\text{DEF}(st_1) \cap \text{REF}(st_2) = \emptyset, \text{REF}(st_1) \cap \text{DEF}(st_2) = \emptyset, \text{DEF}(st_1) \cap \text{DEF}(st_2) = \emptyset}{\llbracket st_1; st_2 \rrbracket \Rightarrow \llbracket st_2; st_1 \rrbracket}$$

DEF(*s*) : Defined variables of *s*.

REF(*s*) : Referenced variables of *s*.

Figure 2. Statement re-ordering transformation

```
class A {
    void f() {
        s1; ...; sk;
        this.g();
        c.h();
        sk; ...; sn;
    }
}
```



```
class A {
    void f() {
        s1; ...; sk;
        c.h();
        sk; ...; sn;
    }
}
aspect B {
    pointcut p(A a):
        cflow(execution(void A.f())) &&
        this(a) && call(void C.h());
    before(A a): p(a) { a.g(); }
}
```

Figure 3. Mechanics of Refactoring 2 (call before another call).

3. The call `x.g()` is present instead of `this.g()`, with `x` a parameter of `f`. In this case, it is sufficient to add `args(x)` to the pointcut to expose the argument and use `x.g()` in the advice. (This is Variant 3 of Refactoring 1.)
4. Although `g()` does not precede (follow) `c.h()`, it is possible to apply Refactoring 2 by applying the semantics preserving re-ordering transformations (See Figure 2).
5. The sequence of statements that precede (follow) `g()` are turned into a separate method (*Extract method*), so as to make Refactoring 2 still applicable.

3.3 Refactoring 3: Conditional call

The final refactoring deals with the following case:

A conditional statement controls the execution of the call to be moved to the aspect.

Figure 4 shows the mechanics of the third refactoring. The conditional statement `if (b)` is considered to be part of the aspect, in that it determines the execution of the call being aspectized (`g()`). Thus, it becomes a dynamically checked condition incorporated into the aspect's pointcut (using the syntax `if(a.b)` in AspectJ). For the execution to be intercepted by pointcut `p`, the condition `a.b` must be **true**. In which case, the new body of method `f` is replaced by the call to `g`, as specified in the around-advice.

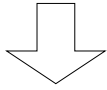
3.3.1 Variants

1. The call `x.g()` is present instead of `this.g()`, with `x` a class field. In this case, it is sufficient to replace `a.g()` with `a.x.g()` in the before-advice.
2. The call `x.g()` is present instead of `this.g()`, with `x` a parameter of `f`. In this case, it is sufficient to add `args(x)` to the pointcut to expose the argument and use `x.g()` in the advice.
3. If `g()` is in the else-part of the conditional statement, it is sufficient to use `if(!a.b)` instead of `if(a.b)` in pointcut.
4. `s1; ...; sn` are not under the control of the condition `b`, in that they are at the top-level in `f`, instead of being inside the else-part of the if-statement. In this case, it is sufficient to add `proceed()` at the end of the around-advice to ensure that `s1; ...; sn` are always executed.
5. If `b` is a parameter of `f`, it is sufficient to add `args(b)` to the pointcut to expose it.

```

class A {
    boolean b;
    void f() {
        if (b) {
            this.g();
        } else {
            s1;...;sn;
        }
    }
}

```



```

class A {
    boolean b;
    void f() {
        s1;...;sn;
    }
}

aspect B {
    pointcut p(A a):
        execution (void A.f()) &&
        this(a) && if(a.b);
    void around(A a): p(a) { a.g(); }
}

```

Figure 4. Mechanics of Refactoring 3 (conditional call).

6. The conditional statement is not at the top level in method `f()`. If it is possible to move it to a separate method (*Extract method*), then Refactoring 3 can be applied.
7. The conditional statement follows another call (say, `h()`) and has no else-part. Refactoring 2 applies, with `if(a.b)` in the pointcut definition and an after-advice.

3.4 Known problems

The following problems derive from specific features currently unsupported by AspectJ. One reason they are reported here is to stimulate discussion of revisions to AspectJ that would incorporate new features to address these limitations.

1. If `x.g()` is present instead of `this.g()`, with `x` a local variable of `f()`, it is not possible to expose `x` in the pointcut. A similar problem occurs in Refactoring 3, when `b` is a local variable of `f()`. This suggests the need for a construct such as `localvars(x)` in AspectJ, in order to expose some local variables of the intercepted method.

2. Inner classes cannot be inserted into given classes by means of aspect introductions (while methods and attributes can).

With the current version of AspectJ, workarounds must be adopted to circumvent the problems listed above. A possible workaround for Problem 1 consists of extracting a code portion surrounding the call `x.g()`, making it a separate method having `x` among its parameters. Once the local variable `x` has become a method parameter, it can be exposed by means of the standard AspectJ constructs. A possible workaround for Problem 2 consists of bringing the inner class to the top-level, adopting proper naming conventions (e.g., `externalClass_innerClass`) and adjusting the visibility of attributes and methods, if necessary.

4 Related work

In the migration of existing OOP code to AOP, the problem that has received most attention is the detection of candidate aspects (aspect mining) [2, 3, 7, 10, 15, 17, 18, 19, 20, 22], while the problem of refactoring [1, 23] towards AOP has, hitherto, been somewhat neglected. Some of the various aspect mining approaches rely upon the user definition of likely aspects, usually at the lexical level, through regular expressions, and support the user in the code browsing and navigation activities conducted to locate them [7, 10, 15, 18]. Other approaches try to improve the identification step by adding more automation. They exploit either execution traces [2, 20] or identifiers [22], often in conjunction with formal concept analysis [20, 22]. Clone detection [3, 19] and fan-in analysis [17] represent other alternatives in this category. For the present work, aspect mining is a prerequisite that is assumed to have been performed, possibly exploiting any of these methods.

The most closely related extraction work is that of Hanenberg et al. [9], Gybels and Kellens [8], and Tourwe et al. [21]. The work of Hanenberg deals with the re-definition of popular OOP refactorings taken from Fowler's work [6] in order to make them aspect-aware, so that each potentially affected aspect is properly updated when the base code is refactored. Moreover, this work considers refactorings to migrate from OOP to AOP and refactorings that apply to AOP code. Among them, the *Extract advice* refactoring is the one we aim to automate in this paper.

Gybels and Kellens, and Tourwe's work uses inductive logic programming to transform an extensional definition of pointcuts (that just enumerates all the join points), into an intensional one, which generalizes the former by introducing variables where facts differ (anti-unification). The underlying assumption is that the pointcut definition language is rule-based (this is not the case, for example, of AspectJ). This work is complementary to ours in that the

problem of generalizing and abstracting the automatically produced pointcuts is not our focus, but is definitely a desirable next step.

5 Conclusions and future work

This paper presents a technique to automate the extraction of aspects from existing code, considering in particular the (hard) problem of pointcut definition. Given a source program with the aspectual fragments identified, our technique produces a semantically equivalent version of the program with the marked statements migrated to aspects and the original execution properly intercepted by the aspect code when necessary.

Several open issues represent the agenda of ongoing and future work. The most important of these are

1. Given a code fragment to be aspectized, it is possible to produce the aspect code following different paths. The three refactorings listed in Section 3 are not always mutually-exclusive, especially if combined with the standard refactoring *Extract method*. Moreover, several variants of the three refactorings may be applicable at the same time. To further complicate the matter, application of one refactoring might enable (or disable) other refactorings for other code fragments, a phenomenon common in program transformations, since its inception [5]. The refactoring path that is chosen influences the quality of the resulting aspect code (for example, application of the refactoring *Extract method* might be artificial and detrimental to the code quality, in some cases). Consequently, a method (perhaps a search-based one [4, 14]) must be devised to select the best path.
2. Implementation of the proposed program transformations is currently under way. The semantic preservation requirements make them non-trivial to realize. On the other hand, since there is no strict syntax-preservation constraint, amorphous techniques could be experimented as well [11, 12, 13].
3. The pre-condition to the statement re-ordering transformation (Figure 2) requires that the defined and referenced variables of the statements to be re-ordered not overlap. Where this constraint is not satisfied, the rule cannot re-order the statements. In the context of this work, this prevents automated aspect extraction. However, a more sophisticated transformation technique could be employed. It might, for example, transform the body of a method to allow re-ordering. Construction of more sophisticated transformations is a challenge as multiple methods must be considered simultaneously.

4. Experimental evaluation of the proposed approach will consist of the automated extraction of aspects from real-world OOP applications. The applicability of the proposed refactorings, with the respective variants, to real code will give us feedback on their usefulness and possibly on the need for further refinement. The quality of the resulting aspects will also be assessed, to determine the actual usability in the field.
5. Generalization of the automatically extracted pointcuts, aimed at decoupling the aspects from the base code, is also an interesting topic for future research. In AspectJ this is achieved through the usage of wildcards. Other, more powerful, pointcut definition languages will also be considered.
6. The implementation and use of our refactoring technique on real-world programs is expected to give us feedback also on the AspectJ language itself. The lack of constructs to intercept the execution at the desired join points and the inadequacy of the pointcut definition language are examples of what could emerge, once we conduct experiments on existing OOP programs. This is invaluable feedback for the aspect languages community.

This paper proposes a transformation based solution to the problem of automated aspect extraction and establishes how such an approach can lead to the development of a theory and algorithms for a largely automated process of aspect extraction. When combined with existing (and automated) approaches to aspect identification, the combined approach opens up the possibility for a tool-supported process of migration from OOP to AOP. This will bring tremendous potential savings in effort, both in the migration process and consequent regression testing. The paper also sets out an ambitious programme of ongoing and future work in the hope of raising interest and awareness of the potential for transformation-based automated aspect extraction.

References

- [1] P. Borba and S. Soares. Refactoring and code generation tools for AspectJ. In *Proc. of the Workshop on Tools for Aspect-Oriented Software Development (with OOPSLA)*, Seattle, Washington, USA, November 2002.
- [2] S. Breu and J. Krinke. Aspect mining using event traces. In *Proc. of Automated Software Engineering (ASE 2004)*, pages 310–315, Linz, Austria, September 2004. IEEE Computer Society.
- [3] M. Bruntink, A. van Deursen, T. Tourwé, and R. van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proc. of the International Conference on Software Maintenance (ICSM)*, pages 200–209. IEEE Computer Society, September 2004.

- [4] J. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEEE Proceedings — Software*, 150(3):161–175, 2003.
- [5] J. Darlington and R. M. Burstall. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.
- [6] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Publishing Company, Reading, MA, 1999.
- [7] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proc. of the 2001 International Conference on Software Engineering (ICSE)*, pages 265–274, Toronto, Canada, March 2001. IEEE Computer Society.
- [8] K. Gybels and A. Kellens. An experiment in using inductive logic programming to uncover pointcuts. In *Proceedings of the First European Interactive Workshop on Aspects in Software*, 2004.
- [9] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *Proceedings of the 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays)*, pages 19–35, September 2003.
- [10] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Proc. of Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering (ICSE)*, Toronto, Canada, 2001.
- [11] M. Harman, D. Binkley, R. Singh, and R. Hierons. Amorphous procedure extraction. In *4th International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, Chicago, Illinois, USA, Sept. 2004. IEEE Computer Society Press, Los Alamitos, California, USA. To Appear.
- [12] M. Harman, D. W. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, Oct. 2003.
- [13] M. Harman, L. Hu, M. Munro, X. Zhang, D. W. Binkley, S. Danicic, M. Daoudi, and L. Ouarbya. Syntax-directed amorphous slicing. *Journal of Automated Software Engineering*, 11(1):27–61, Jan. 2004.
- [14] M. Harman and B. F. Jones. Search based software engineering. *Information and Software Technology*, 43(14):833–839, Dec. 2001.
- [15] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *Proc. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 178–187, Boston, Massachusetts, USA, March 2003. ACM press.
- [16] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, Indiana, USA, 2002.
- [17] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proc. of the 11th IEEE Working Conference on Reverse Engineering (WCRE 2004)*, Delft, The Netherlands, November 2004. IEEE Computer Society.
- [18] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proc. of the 24th International Conference on Software Engineering (ICSE)*, pages 406–416, Orlando, FL, USA, May 2002. ACM press.
- [19] D. Shepherd, E. Gibson, and L. Pollock. Design and evaluation of an automated aspect mining tool. In *MASPLAS*, April 2004.
- [20] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proceedings of the 11th Working conference on Reverse Engineering (WCRE)*, pages 112–121. IEEE Computer Society, November 2004.
- [21] T. Tourwe, A. Kellens, W. Vanderperren, and F. Van-nieuwenhuysse. Inductively generated pointcuts to support refactoring to aspects. In *Proceedings of the Software Engineering Properties of Languages for Aspect Technology (SPLAT) Workshop at AOSD '04*, 2004.
- [22] T. Tourwe and K. Mens. Mining aspectual views using formal concept analysis. In *Proc. of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 97–106, Chicago, Illinois, USA, September 2004. IEEE Computer Society.
- [23] A. van Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE)*, with WCRE, Waterloo, Canada, November 2003.