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 the internal program structure without altering the external behavior (semantic preservation). It aims at improving internal quality factors, such as the modularity, in order to make the code easier to understand and evolve in the future. Among the various refactorings, a category quite unexplored is that of the refactoring from 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 the modules.

In this paper, we investigate the automated support that can be provided to programmers in the migration from OOP to AOP code. We consider the applicability of semantic-preserving code transformations to achieve the migration task. The contribution of this work consists of a list of refactorings, that are described together with the applicability conditions and with all the variants that can be found. By supporting the refactoring activity by means of automated tools the migration difficulties are mitigated and 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*, i.e., of functionalities that are transversal with respect to the principal decomposition of the application and thus cannot be assigned to a single modular unit, in the traditional programming paradigms. Existing software often contains several instances of such crosscutting functionalities, as, for example, persistence, logging, caching, etc. Consequently, refactoring of these applications towards AOP is expected to be beneficial to them, by clearly separating the principal decomposition from the other functionalities and by modularizing the crosscutting concerns.

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

- aspect mining: identification of candidate aspects in the given code;
- refactoring: semantic-preserving transformations that gradually migrate the code to AOP.

In this paper, we focus on the second problem. Moreover, we consider the difficult part of refactoring, that consists of the definition of proper pointcuts to intercept the execution and redirect it to the aspect code, so as 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. 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 refactoring activity is expected to be highly simplified by integrating an automated support such as the one we are proposing.

The paper is organized as follows: in Section 2 we present the refactorings that are required for the automated generation of pointcuts, in the migration of an existing OOP application to AOP. In the next section, we describe the related works, followed by our future work and conclusions.

2 Refactorings for pointcut extraction

In the following, a set of program transformations is described that allow the automated extraction of pointcuts. Input to such transformations is the source code, with the fragments to be migrated to aspects properly marked. This means that aspect mining (i.e., identification and manual validation of candidate aspects in the original code) has already been performed and that the code fragments to be aspectized are known. Usually, such code fragments consist of:

1. whole methods to be aspectized;

2. calls to methods to be aspectized;

3. arbitrary sequences of statements.

In this paper, the focus is on call extraction (case 2), i.e., definition of pointcuts and advices that replace method calls. Method extraction (case 1) is straightforward and requires just to move 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 [2] *Extract method* to turn the statement sequence into a separate method and then to apply method extraction (1) and call extraction (2).

In the following, 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 `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.

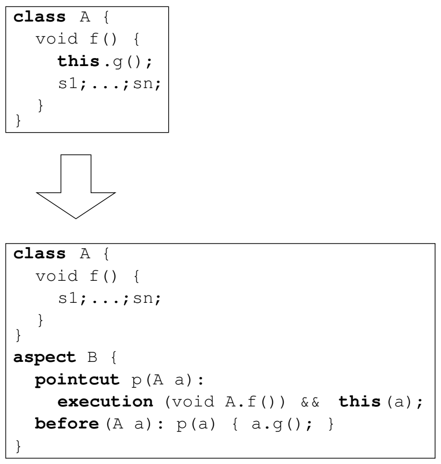
The example code that accompanies the refactorings listed below is written in AspectJ [7], a popular extension of Java with aspects.

2.1 Refactoring 1: Call at the beginning/end

This 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.



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

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

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

2.1.1 Variants

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

2. `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. `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.

4. Although `g()` is not at the beginning (end) of `f()`, it is possible to fall into Refactoring 1 by applying the semantic preserving transformations *Pull call* or *Push call* (see [5]).

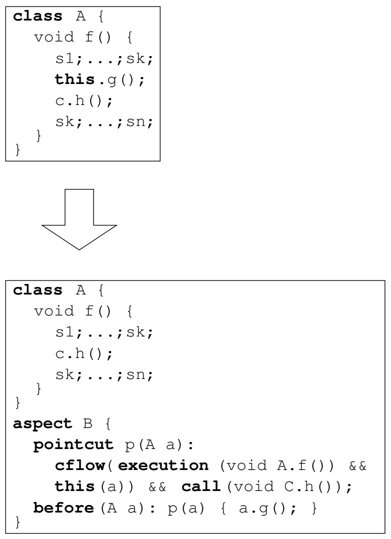
5. Both *Pull call* and *Push call* fail to move the call to the beginning/end of the method body. Assuming they can move the call to the beginning/end of a sequence of statements, it is possible to first turn the statement sequence into a method (*Extract method*) and then apply Refactoring 1.

2.2 Refactoring 2: Call before/after another call

This refactoring deals with the following case:

The call to be moved is always before another call.

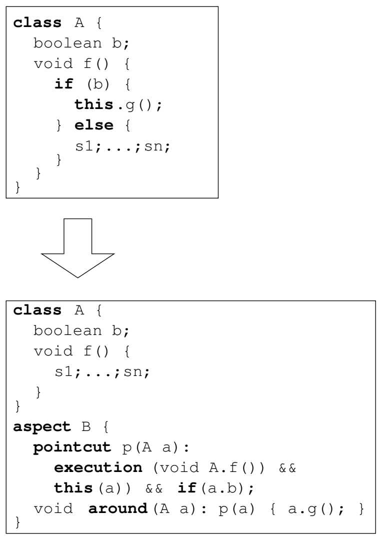
Figure 2 shows the mechanics of this refactoring, under the assumption that in the presence of more than one call to



```
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 2. Mechanics of Refactoring 2 (call before another call).



```
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 3. Mechanics of Refactoring 3 (conditional call).

`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` (`cflow` construct). A before-advice reintroduces the call to `g` at the proper execution points.

2.2.1 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. Variants 2 and 3 of Refactoring 1 apply here as well.

3. Although `g()` does not precede (follow) `c.h()`, it is possible to fall into Refactoring 2 by applying the semantic preserving transformations *Pull call* or *Push call*.

4. The sequence of statements that precede (follow) `g()` are turned into a separate method (*Extract method*), so as to make Refactoring 2 still applicable.

2.3 Refactoring 3: Conditional call

This refactoring deals with the following case:

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

Figure 3 shows the mechanics of this 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 pointcut (with the syntax `if(a.b)`). When the execution is intercepted by the pointcut `p`, requiring that the condition `a.b` is true, the new body of method `f` is replaced by the call to `g`, as specified in the around-advice.

2.3.1 Variants

1. Variants 2 and 3 of Refactoring 1 apply here as well.

2. 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.

3. `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.

4. If `b` is a parameter of `f`, it is sufficient to add `args(b)` to the pointcut to expose it.

5. 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*), Refactoring 3 can be still applied.

6. 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.

2.4 Known problems

The following problems derive from specific features currently unsupported by AspectJ. One of the reasons why we are reporting them is to stimulate a revision of the language aimed at incorporating new features that address such 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, in the case `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.

3 Related works

In the migration of existing OOP code to AOP, the problem that received most attention is the detection of candidate aspects in given programs (aspect mining), while the problem of refactoring [1, 9] towards AOP was 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 [3, 4, 6, 8]. Other approaches try to improve the identification step by adding more automation. They exploit either execution traces [**?**, **?**] or identifiers [**?**], often in conjunction with formal concept analysis [**?**, **?**]. Clone detection [**?**, **?**] and fan-in analysis [**?**] represent other alternatives in this category. For the present work, aspect mining is a prerequisite that is assumed to have been completed, possibly exploiting any of these methods.

The most related works are [**?**, **?**, **?**]. The work described in [**?**] deals with the re-definition of popular OOP refactorings taken from [2] 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.

Inductive logic programming is used in [**?**, **?**] to transform an extensional definition of pointcuts, which 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 more than desirable further step.

4 Conclusions and future work

We have proposed 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 properly marked, our technique produces a semantically equivalent version of the program with the marked statements migrated to aspects and the original execution properly intercepted in order to redirect it to the aspect code when necessary.

Several open issues are not currently addressed and represent the agenda of our future work. The most important of them 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 2 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. The refactoring path that is chosen influences the quality of the resulting aspect code. Consequently, a method (maybe a search-based one) must be devised to select the best path.

2. Implementation of the proposed program transformations is currently on the 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.

3. 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 be also assessed, to determine the actual usability in the field.

4. Generalization of the automatically extracted pointcuts, aimed at decoupling the aspects from the base code as much as possible, will be also an interesting topic for our future research. In AspectJ this is achieved through the usage of wildcards. Other, more powerful, pointcut definition languages will be possibly considered as well.

5. The implementation and usage 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.

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] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Publishing Company, Reading, MA, 1999.

[3] 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.

[4] 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.

[5] M. Harman and S. Danicic. Amorphous program slicing. In *Proc. of IWPC'97, International Workshop on Program Comprehension*, pages 70–79, Dearborn, Michigan, May 1997.

[6] 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.

[7] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, Indiana, USA, 2002.

[8] 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.

[9] 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.