A Program Analysis for Tool-supported Refactoring of Aspect-oriented Programs

Jan Wloka
Fraunhofer FIRST
Kekulstr. 7
Berlin, Germany
jan.wloka@first.fraunhofer.de

Stefan Jähnichen
Technical University Berlin
Franklinstr. 28/29
Berlin, Germany
jaehn@cs.tu-berlin.de

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*patterns, classes and objects*

General Terms

Algorithms, Measurement, Languages, Reliability

Keywords

Software refactoring, Aspect-oriented programming, Change impact analysis, Static program analysis, Code generation

1. PROBLEM & MOTIVATION

Aspect-oriented programming (AOP) has been proposed for improving the modularity of implementations that cannot be encapsulated with traditional means, so called crosscutting concerns. It introduces new adaptation concepts that allow the structural extension of implementation modules and the adaptation of existing program behavior.
The adaptation concepts are often achieved by new language mechanisms and constructs, such as pointcut and advice. A *pointcut* specifies where and when an *advice* is executed by selecting well-defined points in the program execution, so called *join points*. Every time a join point is executed, some runtime support looks for matching pointcuts. If a matching pointcut is defined, every bound advice is executed. With this complex but powerful mechanism, aspects can declare an adaptation without modifying the source of the adapted implementation module.
For selecting a set of join points, a pointcut can specify structural and behavioral properties that have all desired join points in common. These properties are anchored in internal program structures, such as the program's name space, code containment, inheritance relationships or the control flow. A pointcut expresses a join point property by referencing the *anchor* structures and elements, and thereby ties the advice execution closely to these implementation internals. For example, the pointcut

```
call(public int A.bar())
&& withincode(public void A.foo())
```

selects all method calls of method `bar()` that are defined within method `foo()`. Additionally, it references the declarations of `class A` and method `foo()`, although these elements does not belong to the selected join point set. Hence, any modification of `class A` or method `foo()`, e.g., by a refactoring, does not modify selected join points, but may break the pointcut and therefore the specified advice execution.
Refactorings for aspect-oriented programs have to preserve the *anchor* structures and elements but also still allow to improve the program's design. A refactoring tool that is able to identify these structures, can allow safe modifications (even if they affect the set of selected join points), identify and adjust affected pointcuts and prevent the programmer from breaking a pointcut.

2. RESEARCH PROPOSAL

A classification of the change impact of object-oriented refactorings on pointcuts is investigated to cope with the fragile pointcut problem in tool-supported refactoring.

3. BACKGROUND & RELATED WORK

Various approaches have been proposed to cope with the issues arisen by the tight coupling between aspects and the base program. The most related results provide extensions to object-oriented refactorings, new IDE support for determining altered pointcut matches and more expressive pointcut languages, making a pointcut less coupled to brittle implementation details. This section describes the refactoring and program analysis related approaches in more detail.

3.1 Aspect-oriented Refactoring

In [4, 1, 2, 7, 3, 9] new refactorings for aspect-oriented programs has been identified which allow the improvement of object-oriented programs by using AO-modlarization, the refactoring of aspect language constructs or the refactoring of base code while existing AO adaptations are preserved.
In particular, *Monteiro et al.* have published in [4] a catalogue of new refactorings and bad smells for AspectJ programs.

Doctoral Symposium FSE 14, 2006 Portland, Oregon USA

Ceccato et al. present in[1] an AOPMigrator which supports the extraction of class members and statements into aspects. They propose a specific refactoring workflow that generates a single pointcut for every extracted program element.
Hannemann et al. present in [2] also a tool for extracting crosscutting concerns into aspects. The tool supports a specific workflow for migrating design patterns implemented in Java to an AspectJ implementation.

All these refactoring approaches have to cope with base code changes that may affect existing pointcuts. Our approach reveals altered join point selections, proposes suitable pointcut updates and generates rephrased pointcuts. Thus it can be seen as fundamental to AO refactoring in general.

3.2 Behavior Preservation in AO Refactoring

Hanenberg et al. describe in [9] initial ideas on how to treat pointcuts within a refactoring workflow. They extended the refactoring constraints to preserve the number of selected join points, the position of a selected join point within the program's control flow and the information provided at a selected join point for every pointcut. Based on these additional constraints, the previous program behavior is reestablished by updating affected pointcuts.
Two problems are not considered in this approach: (i) a pointcut may intentionally select additional join points after a refactoring and (ii) join points are points in the program execution which have to be statically approximated.
Rura and Lerner advance the AO-specific refactoring constraints in [7]. They present a "pointcut pattern equivalence" constraint, that reduces the rule "each advice must apply at semantically equivalent join points" to a more simple determinable but stronger requirement "signature patterns used in a pointcut must match semantically equivalent program elements". In cases where the signature patterns do not match the same elements as before the refactoring, the patterns are broadened/narrowed in a way that new and lost pattern matches are prevented.
However, additional and lost matches are always excluded or included, which makes, on the one hand, pointcuts unrecognizable and worse readable, and, on the other hand, it does not consider the programmer's intention behind a pointcut.

Our approach can be seen as an extension to this work. We explicitly consider the change impact of a refactoring by assigning atomic changes with a distinct impact on a pointcut to referenced program structures (i.e., pointcut expressions). Rura and Lerner's approach is limited to pattern matches, which e.g., does not allow to move a program element if its matching pattern is restricted by a certain location (i.e., intersected with a within expression). We distinguish if a referenced program element is the actual join point shadow or used to specify one of its properties. With this distinction, we can ensure semantically equivalent matches not only of signature patterns, but also of every program structure used to express a join point property.

3.3 Change Impact Analysis

Another possibility to cope with fragile pointcuts is tool support that assesses changes between two program versions. Such an assessment can be done either by comparing to different program versions or by analyzing the applied changes.
An approach that detects differently bound advices by comparing two program versions is presented by *Störzer and Graf* in [10]. The proposed pointcut delta analysis approximates the bound join points for every pointcut and compares the pointcut matches between two program versions. *New*, *lost* and *modified* (in terms of match quality) matches are discovered.
Our change impact analysis extends this approach in several ways. The computed match delta is extended to a so called pointcut selection delta that contains matches of every referenced program structure, rather than only complete pointcut matches. The selection delta entries are directly associated with the responsible change. This direct association allows the automated calculation of pointcut updates.

Similar to *Ryder et al.* in [6, 8] we divide program edits (performed by a refactoring) into their constituent atomic changes amenable to program analysis. Atomic changes have syntactic dependencies, denoting prerequisite relationships and represent a distinct impact on the program's source. Based on the atomic change model Ryder et al. have developed different change classifications that indicate the likelihood of a change to be failure-inducing. Every change can be labeled, as red (highly likely to be failure-inducing), yellow (somewhere in between) or green (highly unlikely to be failure-inducing).
In contrast to Ryder et al. we do not employ the atomic changes to identify affected unit tests, but affected pointcuts. Our change classification indicates the likelihood of a change to break the program structures a pointcut relies on. A combination of both approaches allows to separate changes that just alter the set of pointcut matches from changes that break the pointcut. With this separation a tool can be enabled to deny critical transformations while it safely performs all others.

4. HYPOTHESIS

In aspect-oriented programs, a pointcut defines where and when advices are executed by selecting a set of join points. It can specify some structural and behavioral properties that have all join points in common. A set of selected join points can be unintentionally altered through various (even local) source code changes. In order to decide if a certain change actually breaks a pointcut, two different kinds of changes have to be distinguished: changes that affect a program element that possesses specified properties (join point shadows) and changes that affect a structure or element that is referenced by the pointcut for expressing a certain property (pointcut anchor).
This distinction allows a refactoring tool to decide whether a certain transformation alters the resulting join point set (*no pointcut update*), modifies a referenced structure or element (*adjust the pointcut*) or breaks a referenced structure or element (*cancel the refactoring*).

5. A CHANGE IMPACT CLASSIFICATION FOR POINTCUTS

A refactoring tool for aspect-oriented programs has to preserve structural and behavioral adaptations defined in aspects. Extended preconditions ensure that defined struc-

tural adaptations are considered in behavior preservation checks and additional transformations adjust new and additional references in aspect modules.
The preservation of behavioral adaptations, however, needs quite more tool support to achieve a similar result. Pointcuts refer to program structures and elements, but they differ in several ways from "traditional" symbolic references. A pointcut can be seen as

- a *multi reference*, as it can bind multiple join points,
- a *fuzzy reference*, as it binds join points by specifying some of their common properties, and
- a *meta-level reference*, as it refers to program structures and elements from the meta-level.

The major difference is that a pointcut can intend the binding of newly created join points, such as the pointcut `call(*)` selects all method calls in a program, regardless how many will be added in the future.

Consequently, the preservation of pointcuts during a refactoring can follow two different strategies: (i) the preservation of the existing program behavior, i.e., the advices are invoked in exactly the same situations as before the refactoring, or (ii) the preservation of the pointcut's meaning, i.e., the advice execution can be altered in situations where the execution conditions are not modified by the refactoring.
Some of the above presented approaches follow the first strategy. In the following we will describe how the second strategy can be achieved and how a tool can decide which strategy has to be applied.

5.1 Modeling Refactoring Changes

A refactoring (if applied) always performs the same transformations. We are interested in changes of program elements that can generally be referenced by a pointcut. Thus, we model every *edit* done by a refactoring's transformation at the level of join points as so called atomic changes. The *atomic changes* comprise *addition* and *deletion* of program elements, such as classes, methods, field and expressions.
The atomic change model combines every atomic change with its change reason represented by the code transformation. Figure 1 shows the extract method refactoring, defined in a similar way as by Opdyke in [5]. Two low-level refactorings are performed for a given list of statements (L) and a given method (M) name are mapped to three transformations that are provided by the refactoring tool. Every transformation causes its specific atomic changes: added method (AM), deleted expression (DE), added expression (AE).

Every atomic change has a distinct impact on the join point space, smaller changes are ignored (because of no impact on pointcuts) and bigger changes are composed of other atomic changes. Since more complex refactorings are composed of low-level refactorings, the change of every refactoring can be represented in terms of atomic changes.

5.2 Decomposition of Pointcuts

A pointcut can select a set of join points by specifying their properties, e.g., the AspectJ pointcut `call(public void foo()) && withincode(public void bar())`, selects all calls of method `foo()` that are located within the method `bar()`. The pointcut expresses this property by referring to the

$$ExtractMethod(L, M) :=$$
$$\quad CreateMethodDeclaration(M)$$
$$\quad +ReplaceStatementListWithMethodCall(L, M)$$

$$ExtractMethod(L, M) :=$$
$$\quad CreateMethodDeclaration(M)$$
$$\quad + MoveStatementList(L)$$
$$\quad + CreateMethodCall(M)$$

$$ExtractMethod(L, M) :=$$
$$\quad \{(AM, CREATE), (AE, MOVE),$$
$$\quad (DE, MOVE), (AE, CREATE)\}$$

Figure 1: Atomic changes of the extract method refactoring

- signatures of method `foo()` and `bar()`,
- calls of method `foo()`, i.e., join points must be method calls and calls of method `foo()`
- containment of method `bar()`'s body, i.e., join points must be in there.

We have developed a model for decomposing AspectJ-like pointcuts into elementary pointcut expressions. If we consider the AspectJ pointcut again, it can be decomposed in

$$within($$
$$call(method(\{public\}, type(void), "foo", \{type(void)\})),$$
$$method(\{public\}, type(void), "bar", \{type(void)\}))$$

Every pointcut expression, like *type*, *method*, *call* and *within*, specifies a single reference to a program structure, e.g., *method* refers to the signature of method declarations, *call* to calls of a method and $within(target, anchor)$ to the containment between target and anchor.
We gain three major benefits from decomposition of pointcuts:

- (i) we know which join point property is specified, by referring to which program structures,
- (ii) we know the specification quality, i.e., whether a reference to a structure or element is partially (using *, or ..) or completely specified, and
- (iii) we represent the effects of an atomic change directly as association to pointcut expressions.

5.3 Change Impact Analysis

Based on the atomic change model and the decomposed pointcuts we have developed a change impact analysis that determines new and lost pointcut matches and associates them with the responsible atomic change.
The **first step** decomposes every pointcut into elementary pointcut expressions as described above. The decomposed representation is normalized into a disjunctive normal form (DNF), expressing commonly specified join point properties within a logical formula combined by "||" operators.
In a **second step**, a static program analysis calculates the pointcut matches by computing the matches for every pointcut expression (PCE). The resulting structure holds all matching program structures and elements. The so called pointcut selection (PCS) is calculated twice, for the original and the

$$CI : AC \times PCS \longrightarrow PCD \times PCE \times TRANS$$
$$AC = \{added, deleted\}$$
$$PCD = \bigcup\{+matches, -matches\}$$
$$TRANS = \{CREATE, REMOVE, RENAME, MOVE\}$$

Figure 2: Change impact representation

$$PCUP : SQ \times CI \times TRANS \longrightarrow \{NoUp, ExM, InM, Ask, Err\}$$

Figure 3: Pointcut update proposition

refactored program.
In the **third step**, a comparison of both pointcut selections computes a delta (PCD) and augments every delta entry with the change reasons (TRANS) from the atomic change model (AC). The resulting impact structure (CI) holds only deltas which actually affect a pointcut, e.g., a renamed method that causes additions and removals of the "same" (semantically equivalent) matches is ignored. The resulting impact structure is shown in Figure 2.

5.4 Pointcut update Computation

For a reasonable update decision the specification quality (SQ) is taken into account in addition to the change impact. Every modified program structure or element that is referenced by a partially specified pointcut expression cannot simply be updated. Especially if a pointcut anchor (element) is modified in a way that the pointcut matches differently, the tool has to consult the programer. Partially specified pointcut anchor make the update calculation often impossible. The analysis finally proposes either to keep the original pointcut (NoUp), to include lost matches (InM) or exclude additional matches (ExM), to ask for the programmer's decision (Ask) or to cancel the refactoring (Err).

6. RESULTS & CONTRIBUTIONS

The following contributions are expected from this work.
An *atomic change model* is developed for representing typical changes done by refactorings. Every atomic change represents a distinct effect on program structures that can be referenced by pointcuts and relates it the responsible transformation (change reason).
A *model for decomposing pointcuts* into elementary pointcut expressions is defined, that makes every referenced program structure and element explicit and distinguishes between references to join point shadows (pointcut matches) and other references used for expressing a certain join point property (pointcut anchors).
Definition and evaluation of a *change classification*. The atomic changes done by typical refactorings are classified regarding their effects on pointcut matches. If a change results in altered pointcut matches, the classification defines the effects on a pointcut in terms of specification quality (of affect pointcut expression), match impact (pointcut match or anchor) and change reason (create, remove, rename, move etc.).
A *change impact analysis* for pointcuts is developed, which allows the computation of an update proposal. Considering the defined change classification any affected pointcut expression is either not adjusted, broadened (to include lost matches), narrowed (to exclude new matches) or labeled as broken or unresolvable.
Definition of *pointcut update patterns* (as extensions to refactorings). In specific circumstances an affected pointcut expression can be completely replaced by an adjusted expression. A few patterns are defined that to replace a pointcut expression, rather than broadening or narrowing it.
A small *impact visualization* is developed depicting reasons and effects in terms of how many new, lost and equal matches are caused by which change. The benefit of this simple visualization metaphor is evaluated especially for partial specifications.
A *prototype refactoring tool* is developed as extension to the Eclipse JDT refactoring support. It appends two additional refactoring steps, pointcut impact review and pointcut update, to the refactoring workflow. The tool implements the proposed analysis and pointcut update approach as well as the impact visualization. It will be the foundation for evaluating the change classification and pointcut update proposition strategy.

The current state of this work encloses the atomic change model, the decomposition into elementary pointcut expressions, a preliminary change classification, the change impact analysis as well as extended refactoring patterns and the definition of an aspect-aware refactoring process. Furthermore, the implementation of the impact analysis works for some pointcuts and few refactorings. The impact visualization and pointcut update proposition is currently under development. A pre-release of the prototype was already presented.

7. EVALUATION STRATEGY

Existing AOP programs, probably implemented in AspectJ, will be augmented with unit tests. Every unit test does not test the aspect, rather than its effects on a certain base code functionality. As a result, whenever an advice binding is altered the tests will fail.
Change impact analysis. Several adapted refactorings that affect either the pointcut anchors, the pointcut matches or none of them will be applied. The tests will help to validate our expectations. In every case, the analysis should determine the correct impact.
Pointcut update decisions. Combinations of four different kinds of pointcuts (simple, complex, partial and complete) will be specified and refactorings which either modify a pointcut matches or an anchor will be applied. Again the unit tests are used to check if the proposed pointcut update is correct.
Pointcut update patterns. In many cases the extended pointcut expressions (if broadened or narrowed) can be simplified by just replacing the original expression. We validate for several situations whether the proposed simplification is valid.

Acknowledgments

This work has been supported by the German Federal Ministry for Education and Research under the grant 01ISC04A (Project TOPPrax).

8. REFERENCES

[1] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated refactoring of object oriented code into aspects. In *ICSM*, pages 27–36. IEEE Computer Society, 2005.

[2] Jan Hannemann, Gail Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In Tarr [11], pages 135–146.

[3] Masanori Iwamoto and Jianjun Zhao. Refactoring aspect-oriented programs. In Omar Aldawud, Mohamed Kandé, Grady Booch, Bill Harrison, Dominik Stein, Jeff Gray, Siobhán Clarke, Aida Zakaria Santeon, Peri Tarr, and Faisal Akkawi, editors, *The 4th AOSD Modeling With UML Workshop*, October 2003.

[4] Miguel Monteiro and Joao Fernandes. Towards a catalog of aspect-oriented refactorings. In Tarr [11], pages 111–122.

[5] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, 1992.

[6] Xiaoxia Ren, Barbara G. Ryder, Maximilian Stoerzer, and Frank Tip. Chianti: a change impact analysis tool for java programs. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 664–665, New York, NY, USA, 2005. ACM Press.

[7] Shimon Rura and Barbara Lerner. A basis for AspectJ refactoring. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04)*. ACM, 2004.

[8] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, New York, NY, USA, 2001. ACM Press.

[9] Rainer Unland Stefan Hanenberg, Christian Oberschulte. Refactoring of aspect-oriented software. In Rainer Unland, editor, *NetObjectDays*, volume 2591 of *Lecture Notes in Computer Science*. Springer, 2003.

[10] Maximilian Störzer and Jürgen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM*, pages 653–656. IEEE Computer Society, 2005.

[11] Peri Tarr, editor. *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*. ACM Press, March 2005.