Deriving Refactorings for AspectJ

Leonardo Cole[*]
lcn@cin.ufpe.br

Paulo Borba[†]
phmb@cin.ufpe.br

Informatics Center
Federal University of Pernambuco
P.O. Box 7851 - 50.732-970 Recife, PE, Brazil

ABSTRACT

In this paper we present aspect-oriented programming laws that are useful for deriving refactorings for AspectJ. The laws help developers to verify if the transformations they define preserve behavior. We illustrate that by deriving several AspectJ refactorings. We also show that our laws are useful for restructuring two Java applications with the aim of using aspects to modularize common crosscutting concerns.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Aspect-Oriented Programming*; D.3.2 [**Programming Languages**]: Language Classifications—*AspectJ*

General Terms

Languages

Keywords

Refactoring, AspectJ, Aspect-Oriented Programming, Separation of Concerns

1. INTRODUCTION

Refactoring [4, 16, 17] has been quite useful for restructuring object-oriented [15, 1] applications. It can bring similar benefits to aspect-oriented [3] applications as well. Moreover, refactoring might be a useful technique for introducing aspects to an existing object-oriented application.

In order to explore the benefits of refactoring, aspect-oriented developers are identifying common transformations for aspect-oriented programs, mostly in AspectJ[11], a general purpose aspect-oriented extension to Java [6]. However, they lack any kind of support for assuring that the transformations preserve behavior and are indeed refactorings.

This paper focuses on that problem and introduces AspectJ programming laws that can be used to derive behavior preserving transformations for this language. Each law basically consists of two transformations, one applying the law from left to right and another in the opposite direction. Our set of laws not only establishes how to introduce or remove AspectJ constructs, but also how to restructure AspectJ applications.

By applying and composing those laws, one can show that an AspectJ transformation is a refactoring. The laws are suitable for that because they are much simpler than most refactorings. Contrasting with refactorings, they involve only localized program changes, and focus on a specific language construct.

2. EVALUATION

We derive large and global refactorings from laws that are simple and localized. Our solution simplifies to show that one refactoring preserves behavior because we intuitively show that each law preserves behavior and thus a composition of those laws also preserves behavior. Moreover, the representation of the precondition as syntactic conditions simplifies the implementation of those refactorings in a tool providing automation.

We used our laws and the derived refactorings to transform two commercial application separating a crosscutting concern with aspects. In the first case, we successfully separated concurrency control from the core logic of the system. In the second, we separated distribution concern from the core. However, distribution could not be completely modularized, since distribution specific exceptions could not be removed from the Facade [5]. This remnant part could not be removed by our Extract Exception Handling refactoring because we do not have access to the class that implements the remote Facade interface.

We also used our laws to derive some of the refactorings proposed in the literature [13, 14, 7, 10]. In most cases, we were able to show that the proposed refactorings preserve behaviour. Nevertheless, there are cases where the refactoring, as proposed, do not preserve behaviour. For instance, the Extract Worker Object [14] makes a generalization of one worker object which implies a change in behavior on the resulting code. The example before the refactoring uses two distinct worker objects, one that completes execution

[*]Supported by CAPES.

[†]Partially supported by CNPq.

without raising an exception and one that may raise an exception. The resulting aspect after the refactoring, has only one advice that uses the second version of the worker object in both cases. Hence, the resulting aspect generalizes the use of the worker object to raise an exception every time, including it in the method that did not handle this exception before. This is similar to merging the advices even though they have a different bodies.

Even though our set of laws is not complete in the sense that it does not represent every feature of AspectJ, it is representative enough to derive several complex refactorings and to completely restructure common implementations of concurrency and distribution concerns. In future work, we intend to extend this set of laws to include more AspectJ constructs. Moreover, we also intend to implement our laws, providing tool assistance and automation.

Another limitation of our laws is related to their soundness. Although soundness, with respect to a formal semantics, is a required property, it is beyond the scope of this article. However, we intend to work on this in the feature. For now, we rely on the simplicity of the laws, which involve only local changes and deal with one AspectJ construct each.

3. RELATED WORK

The behavior preserving property of refactoring is not trivially demonstrated. Several authors [16, 17, 12] show that the use of preconditions would help on this task. We use the concepts of preconditions to define our laws as behavior preserving transformations. Further, our laws are intended to be composed, generating useful behavior preserving refactorings.

The second part of Hanenberg, Oberschulte and Unland's [7] research regards refactorings to AspectJ. In fact, they propose some new refactorings from Java to AspectJ. However, they only discuss coarse-grained refactorings and the conditions to apply them. Our approach focuses instead on fine-grained refactorings as basic laws of programming in order to simplify their understanding and proof. We also derived the proposed refactorings using our laws, showing that they preserve behavior.

Analogously, Iwamoto and Zhao [10] show examples of refactorings from Java to AspectJ. However, there is no argumentation about necessary conditions to apply the refactorings to ensure that they preserve behavior. We used the suggested refactorings and derived them as a composition of our laws. We were able to state in which conditions we can apply the refactorings as well.

Another related work [8] is about a tool implemented to support the task of refactoring an aspect-oriented system. Their approach consists in developing a tool to be integrated with the Eclipse IDE. It is designed to involve the developer in a dialog to build the refactoring based on the concern description. They have two approaches to achieve that. The first uses a concern graph to describe and implement the refactoring. The second, chooses a target design pattern from the GoF [5] and restructures it using aspects according to a previous work [9].

Finally, there is a related work [13, 14] that discusses aspect-oriented refactorings showing problems when applying object-oriented refactorings in the presence of aspects. It proposes several complex and interesting refactorings and shows clear and easy to understand examples. We derived most of the proposed refactorings.

4. REFERENCES

[1] G. Booch. *Object–Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition, 1994.

[2] M. d'Amorim, C. Nogueira, G. Santos, A. Souza, and P. Borba. Integrating Code Generation and Refactoring. In *Workshop on Generative Programming, ECOOP02*, Malaga, Spain, June 2002. Springer Verlag.

[3] T. Elrad, R. E. Filman, and A. Bader. Aspect–Oriented Programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison–Wesley, 1999.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison–Wesley, second edition, 2000.

[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 (Net.ObjectDays)*, pages 19–35, Erfurt, Germany, Sept. 2003.

[8] J. Hannemann, T. Fritz, and G. C. Murphy. Refactoring to aspects: an interactive approach. In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange*, Anaheim, California, USA, Oct. 2003.

[9] J. Hannemann and G. Kiczales. Design pattern implamentation in java and AspectJ. In *OOPSLA'2002*, page 161, 2002.

[10] M. Iwamoto and J. Zhao. Refactoring aspect-oriented programs. In F. Akkawi, O. Aldawud, G. Booch, S. Clarke, J. Gray, B. Harrison, M. Kandé, D. Stein, P. Tarr, and A. Zakaria, editors, *The 4th AOSD Modeling With UML Workshop*, 2003.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[12] G. Kniesel and H. Koch. Static composition of refactorings. In R. Lämmel, editor, *Science of Computer Programming*, Special issue on "Program Transformation". Elsevier Science, 2004.

[13] R. Laddad. Aspect-Oriented Refactoring Series - Overview and Process. TheServerSide.com, Dec. 2003.

[14] R. Laddad. Aspect-Oriented Refactoring Series - The Techniques of the Trade. TheServerSide.com, Dec. 2003.

[15] B. Meyer. *Object–Oriented Software Construction*. Prentice–Hall, second edition, 1997.

[16] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[17] D. Roberts. *Practical Analysis for Refactoring*. PhD thesis, Urbana-Champaign, IL, USA, 1999.