Improving Safety When Refactoring Aspect-Oriented Programs

Diego Cavalcanti
diego@diegocavalcanti.com
Federal University of Campina Grande
Campina Grande, PB, Brazil

Research Advisor: **Prof. Rohit Gheyi**

1. PROBLEM & MOTIVATION

The term refactoring was coined by Opdyke [Opdyke 1992] as behavior-preserving program transformations that improve the quality of the resulting object-oriented (OO) code. Then, Fowler [Fowler 1999] popularized the term, encouraging its use in practice. Commonly, developers perform refactoring either manually (which is error-prone and time consuming) or using IDEs that support refactoring, such as Eclipse, Netbeans, JBuilder and IntelliJ. The use of tools is recommended since they help avoid various errors that can easily be introduced by manual refactoring.

In order to preserve the external behavior, each refactoring may contain a number of preconditions. For instance, to rename an attribute, name conflicts must be avoided. However, refactoring tools are commonly implemented in an ad hoc way, checking only some conditions, without a formal basis, since checking correctness with respect to a formal semantics is prohibitive. Therefore, popular object-oriented (OO) refactoring tools (such as Eclipse and Netbeans) allow transformations intended to be refactorings but that introduce behavioral changes [Ekman et al. 2008]. To detect the changes of behavior, programmers must rely on compilation and tests. However, test suites are often not good at catching behavioral changes in the refactoring context, since tests may also be refactored. For instance, a simple renaming may introduce defects [Ekman et al. 2008].

Similar to OO programs, we have aspect-oriented (AO) refactorings [Monteiro and Fernandes 2005, Cole and Borba 2005, Hanenberg et al. 2003, Wloka et al. 2008]. In this case, the problem becomes even worse, since the most used refactoring tools do not have good support for refactorings in the presence of aspects. The Eclipse IDE supports a small number of refactorings. Nevertheless, it also introduces behavioral changes in programs.

Motivating Example. Next, we show an example which introduces a behavioral change that is not detected by Eclipse. Consider a financial environment in which one needs to calculate a bank fee whenever the account's balance is updated. Listing 1 shows a class which represents this situation and Listing 3 presents an aspect which calculates and updates the bank fee. Whenever the method *deposit* is called, it calls the method *updateBalance* in order to update the balance. This method is a join point picked out by a pointcut at *TaxAspect* (Listing 3). It updates the field *bankFee*.

Listing 2 presents the Eclipse resulting code when we apply the Inline Method refactoring in *updateBalance* (Listing 1). However, Eclipse does not detect the behavioral change: the aspect (Listing 3) is not changed, so the field *bankFee* (Listing 2) is not updated anymore.

Listing 1. Original Version

```
public class Account {
  double bankFee = 0.0;
  double balance = 0.0;
  void deposit(double value) { ...
        updateBalance(value);
  }
  void updateBalance(double v) {
    balance = balance + v;
  } }
```

Listing 2. Refactored Version

```
public class Account {
  double bankFee = 0.0;
  double balance = 0.0;
  void deposit(double value) {
    ...
    balance = balance + value;
  }

}
```

Listing 3. Aspect to update the income tax

```
after(Account a, double v) :
    set(double Account.balance) &&
    withincode(* Account.updateBalance(double))
    && args(v) && target(a) {
  a.bankFee = a.bankFee + 0.20*v;
} }
```

2. Several AO refactorings are proposed in order to restructure existing AO programs [Monteiro and Fernandes 2005, Hanenberg et al. 2003, Wloka et al. 2008]. Moreover, some works also show how to migrate from the OO to AO paradigm, mainly from Java to AspectJ [Binkley et al. 2005, Monteiro and Fernandes 2005, Hannemann 2006]. However, since AO refactorings are not formally proposed, developers must test their applications to check if the transformations preserve behavior.

Moreover, formally verifying program refactorings is a challenge [Schafer et al. 2009]. Some approaches contribute in this direction. Cole and Borba [Cole and Borba 2005] propose AO programming laws for deriving refactorings for AspectJ. The laws help developers ensure that the proposed transformations preserve behavior and are indeed refactoring. However, the proposed laws do not include all AspectJ's resources. Because of this, developers still need to rely on tests to verify behavior preservation.

3. APPROACH & UNIQUENESS

We propose a technique for detecting behavioral changes in AO programs. By analyzing the transformed program, our technique generates tests that can be run on two versions of the program: the source (original program, before the refactoring); and the target (refactored program). They aim at exercising the program changes introduced by transformations.

Our technique is broken into seven steps for each transformation, as follows: first of all (Step 1), the developer chooses a refactoring which will be applied and uses our technique to increase confidence that the refactoring will not introduce behavioral changes in the code, which is passed as argument for our technique. Then, a refactored version of the code is generated (Step 2) using the refactoring API of used IDE (e.g., Eclipse). So, at the end of these both steps, we will have two versions of the program: the original version (*source*) and the refactored version (*target*). At the Step 3, we apply a static analysis in order to identify methods with the same characteristics (return type, name, arguments types and exceptions thrown) which are present in the classes affected by the desired transformation, in both source and target versions. After that, at Step 4, we use a modified version of Randoop [Pacheco et al. 2007] to generate unit tests for the methods identified in Step 3. Randoop generates unit-tests using feedback-directed random test generation. We modified it in order to generate unit tests to a set of methods in common to both source and target versions. Additionally, it was modified to accept Java classes woven with aspects. Therefore, all generated tests only have calls to methods identified in the previous step. Moreover, the Steps 3 and 4 guarantee that the same tests can be applied to both source and target versions, without changes. Step 5 aims at running the generated test suite on the source. If no test fails,

the same test suite is also run on the target version of the program (Step 6). If all tests run successfully on the target, we improve the confidence that the transformation does not introduce changes of behavior (Step 7) and then the refactoring can be applied.

The technique's workflow is depicted in Figure 1. In summary, our technique analyzes the program, generates unit tests for common methods, runs it on the source version of the program and also runs all the successful tests on the target. If some test on the target fails, we assume that the transformation introduced a behavioral change in the program. In that case, our technique warns the developer that the transformation is not safe and should not be applied. Our work proposes a practical approach which aims at facilitating the task of detecting behavioral changes through the analysis of changes and generation of tests, in this way increasing refactoring safety.

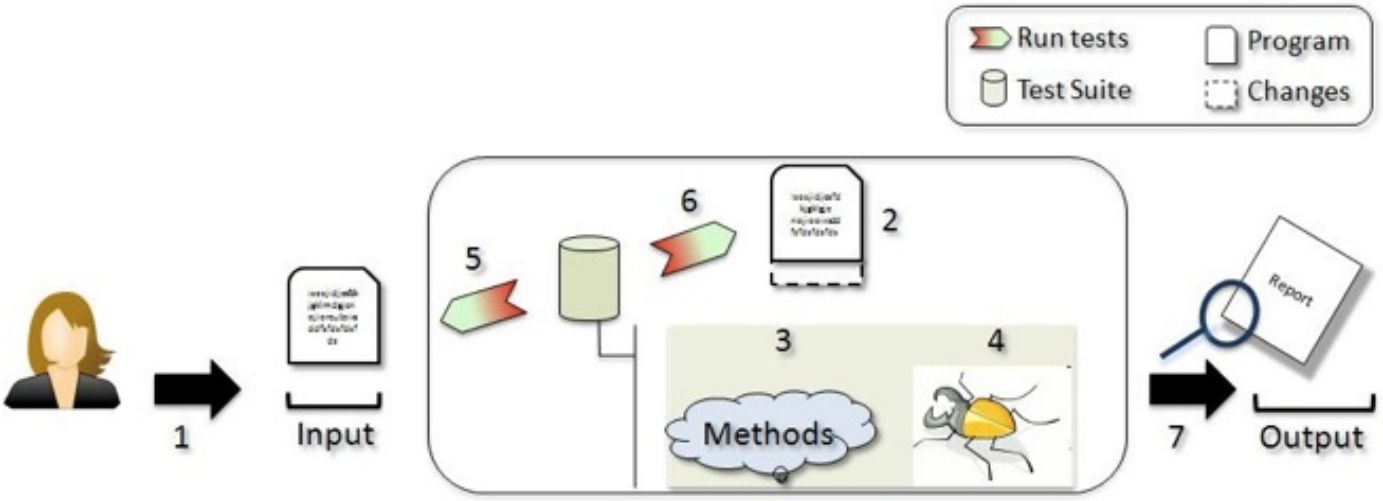

Figure 1. Our Refactoring Technique

4. EXPERIMENTAL RESULTS & CONTRIBUTIONS

In order to evaluate and improve our work, we automatized our technique and applied it on transformed programs implemented using AspectJ. The experimental evaluation can be divided into two categories:

- *Defective Transformations*: transformations applied by some refactoring tools but that cause behavioral changes which are not detected;
- *Refactorings Applied in Real Case Studies*: transformations intended to be refactorings which were presented in the literature and applied by developers in real programs.

Defective Transformations

Based on the catalogue proposed by Ekman et al. [Ekman et al. 2008], we built a catalogue of 12 non-trivial AO transformations, to which we applied transformations intended to be refactorings, but that introduce behavioral changes. We evaluated the most common refactorings (Rename Method, Rename Class, Extract Method, Pull Up Method, Inline Method, Push Down Method and Encapsulate Field) [Hill et al. 2008] applying them using Eclipse and Netbeans in aspect-oriented programs with AspectJ.

The subjects are composed of a small set of classes and different pointcut definitions and advice declarations. For instance, we applied our technique to the example presented in Section 1. It generates tests for the source version of the program and, when we run the generated test suite against the target version, some tests failed.

Overall, our evaluation with defective transformations was successful in detecting, in less than 1 minute, 91% of the behavioral changes. Our technique also identifies behavioral changes in programs using reflection. Some of our limitations, are that our current implementation cannot detect methods which differ

in the standard output and also cannot be applied to concurrent programs.

Real Case Studies

We also evaluated our technique with five open-source projects from 3KLOC to more than 65KLOC. We found all the refactored versions of these aspect-oriented programs in the literature. The experiments aim to improve confidence that both versions of the programs (before and after refactoring) have the same observable behavior.

Table 1 presents some information about real programs which we used in evaluation. It is important to say that all refactorings were applied by experts and were published in conferences worldwide as behavior-preserving transformations.

Table 1. Real Case Studies Evaluated

Subject	Summary	LOC
Prevayler	Open-source library for object persistence	2,949
JAccounting	Web-system to order and payment control	6,355
JSpider	Web spider engine	9,873
JHotDraw	Framework for graphics	21,055
FreeMind	Mind-mapping software	65,489

Case studies annotated with JML - Prevayler, JAccounting and JSpider

The subjects Prevayler, JAccounting and JSpider are programs annotated with *Java Modeling Language* (JML) [Leavens et al. 2006], which is a behavioral interface specification language used to specify contracts, such as pre and post conditions and invariants with annotations. To compile a class annotated with JML, one can use the JML compiler (the *jmlc*) [Cheon and Leavens 2002]. From an annotated program, it generates a Java program with exceptions to check the specified conditions. Two other compilers for JML were also proposed: the *ajmlc* [Rebelo et al. 2008], which generates code in AspectJ with the checking of conditions modularized into aspects; and another compiler which is the same as the *ajmlc* but that was refactored for optimization [Rebelo et al. 2009].

All three experiments (Prevayler, JAccounting and JSpider) were made available by the authors of *ajmlc* to analyze if all compilers really do generate programs with the same observable behavior. However, with our technique, in about 2 minutes, we analyzed the programs and automatically generated tests for them, finding some errors. Table 2 shows a summary of our results, with the number of tests that were automatically generated by our technique and how many failed. It can be noted that all the experiments showed problems. For instance, for the subject JAccounting containing more than 28KLOC, our technique analyzed it, generated more than 3,000 tests and, in less than 2 minutes, it was possible to find a behavioral change that the authors of the refactoring did not notice. The same occurred with the other subjects.

Analyzing the code, based on the tests which failed, we could find that the problem was in the code which was generated from the second compiler (*ajmlc*). It generates aspects which incorrectly check invariants before constructors.

Table 2. Results of JML Case Studies

Subject	Tests	Failures	Time
Prevayler	125	28	< 2 mins
JAccounting	3,506	39	< 2 mins
JSpider	861	42	< 2 mins

JHotDraw

Another experiment performed was with JHotDraw (http://www.jhotdraw.org). It is a java graphical user interface framework for graphics and has more about 22KLOC and more than 200 classes and interfaces. The original code is written in Java and was refactored for AspectJ in order to modularize the exception handling. The refactoring was performed by manual steps with pair programming and code revision.

When we applied our technique to both versions of the program, it generated more than 18 hundred tests, of which 12 tests have failed. So, with our technique, we could find a behavioral change. The tests pointed out that one version is throwing an exception which did not exist in the other version. The problem arises because we have some classes that implement Serializable and have some attributes that do not implement it. So, developers introduced a problem with class serialization.

Freemind

Finally, we performed an evaluation using Freemind (http://freemind.sourceforge.net), a mind-mapping software written in Java, containing 65KLOC and 475 classes. Its original version was refactored to AspectJ in order to extract a software product line. The refactored program has 472 classes and 7 aspects.

In 90 seconds, on a Intel Pentium Dual Core 2.5 GHz processor with 1 GB of RAM, our technique analyzed the both versions of the program and generated 3,502 tests which aim at exercising 140 LOC of difference in both versions. It has been successful in detecting one behavioral change in the refactored program.

CONCLUSIONS

In this work, we propose a practical approach for generating tests in order to find behavioral changes in AO transformations, which is useful during refactoring activities. We evaluated our work with five refactored open-source projects found in the literature and also with defective transformations catalogued for AspectJ.

All refactorings which were applied to real case studies were made by experts with the help of refactoring tools, code revision and pair programming. Nevertheless, the refactored programs presented different behavior when compared with their original versions. These behavioral changes were not noticed by the authors of transformations when refactoring. They occur because the complexity of transformations in AOP is bigger than OOP, since aspect-aware refactoring should be concerned with program control flow, wildcards and others. With our technique, one can analyze the modified program, generate test cases which exercise the affected code and in few minutes increase confidence that program behavior was preserved.

ACKNOWLEDGMENTS

I would like to thank Rohit Gheyi for his guidance. I would also like to thank the referees of ACM Student Research Competition at OOPSLA'09, Tiago Massoni, Daniel Wong and Anderson Ledo for their comments and assistance. This research was supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq, grant 573964/2008-4.

REFERENCES

- Binkley, D., Ceccato, M., Harman, M., Ricca, F., Tonella, P., Coll, L., and Baltimore, M.(2005). Automated refactoring of object oriented code into aspects. In ICSM ’05, pages 27-36.
- Cheon, Y. and Leavens, G. T. (2002). A runtime assertion checker for the java modeling language (jml). In Proceedings of SERP ’02, Las Vegas, pages 322–328. CSREA Press.
- Cole, L. and Borba, P. (2005). Deriving refactorings for AspectJ. In Aspect-Oriented Software Development Conference (AOSD) ’05, pages 123–134.
- Ekman, T., Ettinger, R., Schafer, M., and Verbaere, M. (2008). Refactoring bugs in Eclipse, IDEA and Visual Studio.
- Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. Addison-Wesley.
- Hanenberg, S., Oberschulte, C., and Unland, R. (2003). Refactoring of aspect-oriented software. In Net.ObjectDays, pages 19–35.
- Hannemann, J. (2006). Role-based refactoring of crosscutting concerns. PhD thesis.
- Murphy-Hill, E. and Black, A. P. (2008). Refactoring tools: Fitness for purpose. IEEE Software, 25(5):38–44.
- Leavens, G., Baker, A., and Ruby, C. (2006). Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes.
- Monteiro, M. P. and Fernandes, J. (2005). Towards a catalog of aspect-oriented refactorings. In AOSD ’05, pages 111–122.
- Opdyke, W. (1992). Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign.
- Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. (2007). Feedback-directed random test generation. In ICSE ’07, pages 75–84.
- Rebêlo, H., Soares, S., Lima, R., Ferreira, L., and Cornélio, M. (2008). Implementing java modeling language contracts with aspectj. In SAC ’08, pages 228–233.
- Rebêlo, H., Lima, R., Cornélio, M., Leavens, G. T., Mota, A., and Oliveira, C. (2009). Optimizing JML features compilation in ajmlc using aspect-oriented refactorings. In SBLP ’09, pages 117–130.
- Schäfer, M., Ekman, T., and de Moor, O. (2009). Challenge proposal: Verification of refactorings. In Programming Languages meets Program Verification, pages 67–72.
- Wloka, J., Hirschfeld, R., and Hansel, J. (2008). Tool-supported refactoring of aspect-oriented programs. In AOSD ’08, pages 132–143.