Making aspect-oriented refactoring safer

Gustavo Soares, Diego Cavalcanti, Rohit Gheyi

Federal University of Campina Grande, Brazil
{gsoares, diegot, rohit}@dsc.ufcg.edu.br

Abstract. Developers may refactor part of the object-oriented (OO) code into aspects in order to improve modularity. However, since most refactoring tools have a limited or no support for aspect-oriented (AO) constructs, developers have to apply manual steps. Even the aspect-aware refactorings contain bugs. Developers need a better support for making AO refactorings safer. In this paper, we propose a tool for evaluating whether AspectJ transformations preserve behavior. We evaluate it in 8 non-behavior-preserving transformations applied by Eclipse. Our tool detected all of them. Additionally, we compared the OO version and its refactored AO version of two real case studies. Our tool found a non-behavior-preserving transformation. We also analyzed 23 design patterns implemented in Java and AspectJ. Our tool identified that the AO and OO versions of the State pattern are not equivalent. Finally we evaluated two JML compilers that generate AO code. Our tool identified a bug in one of them.

1 Introduction

Aspect-oriented (AO) programming aims at increasing modularity by allowing the separation of crosscutting concerns [9], such as persistence and exception handling. AspectJ [10] is a general purpose aspect-oriented extension to the Java language. Developers may apply refactorings [14] to extract part of the object-oriented (OO) code into aspects.

Existing Integrated Development Environments (IDEs), such as Eclipse and Netbeans, offer limited or no support to refactor AO programs. For instance, the Eclipse 3.6 IDE with AJDT 1.2.0 only supports the aspect-aware refactorings: *Renaming*, and *Pull Out and Pull In Intertype Declaration*. However, they contain bugs (Section 2). Moreover, a number of useful refactorings [13] implemented by Eclipse, such as *Extract Method*, do not consider aspects. Therefore, developers usually have to manually refactor aspect-oriented programs, which is an error-prone and a time consuming activity. In general, they use test suites to guarantee behavior-preservation. They need a better tool support for making aspect-oriented refactorings safer.

Inspired by the Fowler's catalog [4], Monteiro and Fernandes [12] proposed 27 refactorings that can be used to introduce aspects and improve the design of AO programs. Cole and Borba [2] formally specify aspect-oriented behavior-preserving transformations, and use them for deriving aspect-oriented refactor-

ings. Wloka et al. [25] propose a tool support for extending currently OO refactoring implementations for considering aspects. Specifying and implementing refactorings is difficult. For instance, most of the current Java refactoring implementations do not check all preconditions allowing non-behavior-preserving transformations [21]. In fact, for complex languages such as Java, proving refactorings with respect to a formal semantics is considered a challenge [19]. This problem is even worse with the presence of aspects. We need a practical way to evaluate whether a transformation preserves behavior. In this way, developers can use it to improve confidence that the transformations applied by refactoring tools preserve behavior.

In our previous work [21,20], we proposed a tool (SafeRefactor) for detecting behavioral changes in OO transformations. SafeRefactor analyzes a transformation and generates unit tests specific for detecting behavioral changes. We evaluated it in OO transformations with up to 100KLOC. Our tool was useful for detecting a number of bugs in Eclipse and Netbeans.

In this paper, we make small changes in SafeRefactor in order to extend it to the AO context (Section 3). We evaluated SafeRefactor in 37 programs and their AO counterpart. They are supposed to be equivalent. First, we evaluated in 8 defective refactorings performed by Eclipse with AJDT 2.1.0 that change program's behavior in the presence of aspects. SafeRefactor detected all of them. Then, we used our tool to evaluate four transformations performed in two real case studies [23] (20 and 23 KLOC) to modularize exception handling in aspects. SafeRefactor identified a behavioral change in one of the transformations. We also evaluated whether 23 design patterns implemented in Java and AspectJ [7] are equivalent. SafeRefactor found that the implementations of the State pattern are not equivalent. Finally, we tested two JML compilers implemented using AspectJ [17,18]. As test inputs, we use two JML programs (6 and 9 KLOC), and SafeRefactor compares the behavior of the compiled programs. SafeRefactor found that the compiled programs have different behavior.

In summary, the main contribution of this paper is the following:

– An evaluation of SafeRefactor in 37 programs and their AO counterpart (Section 4).

2 Motivating Example

In this section, we present a defective refactoring performed by Eclipse 3.6 with AJDT 2.1.0 that introduces a behavioral change.

Consider the `A` class, its subclass `B`, and the `AspectB` aspect presented in Listing 1.1. The `B` class declares methods `k` and `test`. Moreover, `AspectA` declares the `n` method in `A` through an inter-type declaration. The `test` method yields `10`. By using Eclipse 3.6 with AJDT 2.1.0 to apply the Rename Inter-type Declaration refactoring to `A.n` changing its name to `A.k`, it yields the program presented in Listing 1.2. Eclipse changed the inter-type's name and updated its references. However, this transformation introduces a behavioral change: the

`test` method yields `20` (target version) instead of `10` (original version). After the transformation, it calls `B.k` instead of the `A.k` method.

Listing 1.1. Original version

```
class A {}
class B extends A {
  int k() {
    return 20;
  }
  int test() {
    return n();
  }
}
aspect AspectA {
  int A.n() {
    return 10;
  }
}
```

Listing 1.2. Target version

```
class A {}
class B extends A {
  int k() {
    return 20;
  }
  int test() {
    return k();
  }
}
aspect AspectA {
  int A.k() {
    return 10;
  }
}
```

Fig. 1. Applying the Rename Inter-type Declaration refactoring of Eclipse leads to a behavioral change.

The previous problem is small, likely to be detected by a developer or a test suite. However, the current tool support for applying AO refactorings is limited. Moreover, there are only a small number of aspect-aware refactorings implemented. Usually, developers have to manually apply transformations and check whether they are sound. However, this activity is time-consuming and error-prone. The Java semantics is not simple [6]. For example, a simple transformation changing the access modifier may have an impact on a number of Java constructs [22]. By including AspectJ constructs such as, pointcuts, advices, and inter-type declarations, this task can be even harder. In this paper, we show that SafeRefactor can be useful for making aspect-oriented refactoring safer.

3 SafeRefactor

In this section, we present an overview of SafeRefactor. It is a tool useful for detecting behavioral changes in transformations applied to sequential Java/AspectJ programs [21].

The tool analyzes a transformation and generates a number of tests suited for detecting behavioral changes. The analysis consists of identifying the common methods, that is, methods with same signature before and after the transformation. Next, SafeRefactor generates a test suite for the methods previously identified within a given time limit. Since the tool focuses on identifying common methods, it executes the same test suite before and after the transformation.

SafeRefactor uses Randoop [16], a Java unit test generator, to perform the test case generation. Randoop randomly generates tests for a set of classes and methods given a time limit or a maximum number of tests. Finally, the tool executes the tests before and after the transformation, and evaluates the results: if they are different, the tool reports a behavioral change, and yields the unit tests that reveal it. Otherwise, we improve confidence that the transformation is behavior-preserving. Figure 2 illustrates this process.

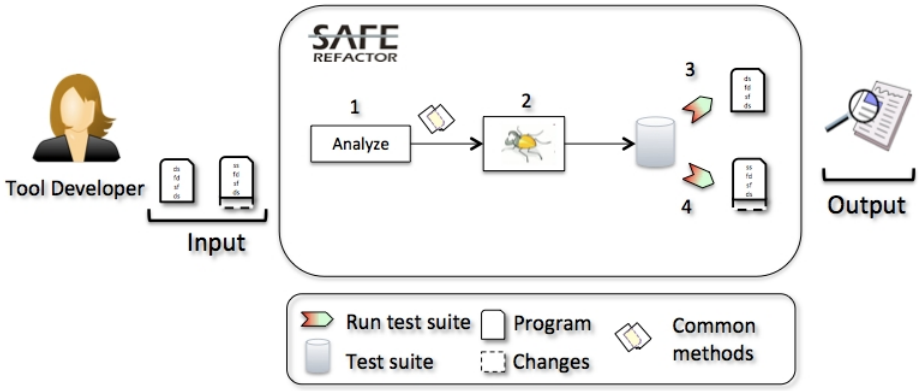

Fig. 2. SafeRefactor's technique: 1) It identifies the methods with same signature before and after the transformation; 2) It generates a test suite for the identified methods using Randoop; 3) It runs the tests on the source program; 4) It runs the tests on the target program; 5) Finally, SafeRefactor evaluates the results: if they are different, the tool reports a behavioral change, otherwise, we increase confidence that the transformation preserves behavior.

Consider the transformation presented in Figure 1. SafeRefactor receives as input the programs shown in Listings 1.1 and 1.2. First, it identifies the methods with the same signature on both versions: `B.k` and `B.test`. Next, it generates in 1 second (time limit) 12 unit tests for these methods. Finally, it runs the test suite on both versions and evaluates the results. A number of tests (11) passed in the source program but they did not pass in the refactored program. Listing 1.3 shows one of the generated tests that reveals the behavioral change. The test passes in the source program since the value returned by `B.test` is `10`, but it fails in the target program since the value returned by `B.test` in this version is `20`. Therefore, SafeRefactor reports a behavioral change.

Listing 1.3. A unit test revealing a behavioral change in the transformation performed by Eclipse from Listing 1.1 to Listing 1.2.

```
public void test() {
  B var0 = new B();
```

```
    long var1 = var0.test();
    assertTrue(var1 == 10);
  }
```

The first step of SafeRefactor consists of identifying the common methods. We modified the original version of SafeRefactor for allowing it to identify methods added in the classes by aspect inter-type declarations. We identify them by analyzing the class bytecode.

4 Evaluation

We evaluated our technique in 8 defective refactorings applied by Eclipse, 2 real case studies, 23 design patterns and 2 JML compilers[1]. They are described in Sections 4.2-4.5. Section 4.1 describes the experimental setup.

4.1 Experimental Setup

We run the experiment on a dual-processor 2.2 GHz laptop with 4 GB RAM and running Mac OS 10.6.7. We used the command line interface of SafeRefactor. It receives three parameters: source and target program paths, and time limit to generate tests. We used the default time limit of 2s to generate tests in the transformations described in Sections 4.2 and 4.4. This time limit is enough to test transformations applied to small programs. On the other hand, we used a time limit of 90s to evaluate the transformations in Sections 4.3 and 4.5.

4.2 Defective Refactorings

Subject Characterization

By applying the Eclipse refactorings in small toy examples created by us, we manually identified 8 transformations applied by Eclipse IDE 3.6 that introduce behavioral changes in AO programs. Eclipse is a popular Java IDE with a number of automated refactorings and offers refactoring support for AspectJ.

Table 1 describes the transformations. Each subject contains a small set of classes, pointcuts, advices, and inter-type declarations. For instance, the example shown in Figure 1 is the Subject 5 of our evaluation. Since AJDT 2.1.0 delivered in 2010, the Rename refactorings of Eclipse are aspect-aware. In Subjects 1-5, we apply transformations performed by these aspect-aware refactoring implementations. On the other hand, in Subjects 6-8, we apply different kinds of useful OO refactorings (Push Down Method, Pull Up Method, and Inline Method) performed by Eclipse that are unaware of aspects. In practice, since refactoring tools have a limited support for AO refactorings, developers may have to manually perform a transformation or use an OO refactoring implementation to automate part of the transformation, and manually check whether it preserves behavior.

[1] All experiment data are available at: http://dsc.ufcg.edu.br/~spg/papers.html

Table 1. A catalog of transformations performed by Eclipse that introduce behavioral changes in the presence of aspects; Column Test: number of tests generated by SafeRefactor; Column Fail: number of tests that have different results after the transformation; Result: the result of the SafeRefactor's evaluation.

Subject	Refactoring	Test	Fail	Result
1	Rename Class	63	1	Behavior Change
2	Rename Method	83	24	Behavior Change
3	Rename Field	167	1	Behavior Change
4	Rename Intertype Declaration	171	160	Behavior Change
5	Rename Intertype Declaration	119	118	Behavior Change
6	Push Down Method	200	188	Behavior Change
7	Pull Up Method	192	180	Behavior Change
8	Inline Method	219	24	Behavior Change

Results

SafeRefactor detected behavior changes in all transformations in less than 8s. Table 1 contains the number of generated tests (Column Tests) and the number of tests that detects the behavioral change (Column Fail) for each subject. Column Result indicates whether SafeRefactor identified a behavior change.

In some aspect-aware refactorings (Subjects 2 and 3), Eclipse did not update pointcuts, leading to behavioral changes. Pointcuts may use wildcards that may impose additional challenges when checking preconditions. The behavioral changes in Subjects 4 and 5 are due to OO features, such as overloading and overriding. In the presence of aspects, this problem is even worse.

4.3 Real Case Studies

Subject Characterization

Taveira et al. [23] present two approaches to modularize exception handling mechanism. They change an OO version into equivalent ones: OO' (a class modularizes it) and AO (an aspect modularizes it), as depicted by Figure 4.3. Eight programmers working in pairs performed the changes. They relied on refactoring tools, pair review, and unit tests to assure behavioral preservation.

They refactored JHotDraw and CheckStylePlugin to use the approach proposed. We use SafeRefactor to evaluate the OO to AO versions, and the OO' to AO versions (see Figure 4.3) of JHotDraw and CheckStylePlugin. The OO, OO' and AO versions must be equivalent. Table 2 contains the four transformations evaluated by SafeRefactor. It shows the program's name, its size, and the pair of versions evaluated. In our previous work [21], we use SafeRefactor to evaluate the OO to OO' refactoring applied to JHotDraw and CheckStylePlugin. We found a behavioral-change in the refactoring applied to JHotDraw.

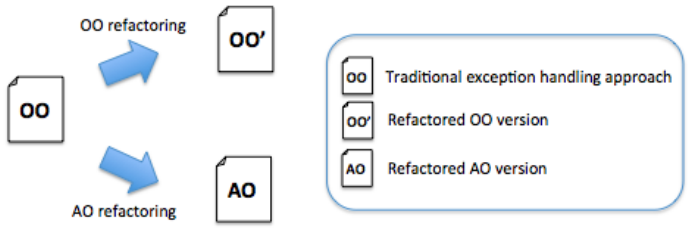

Fig. 3. Two alternative refactorings to modularize exception handling code.

Table 2. Refactoring real applications; Column KLOC: nonblank, non-comment lines of code; Column Test: number of tests generated by SafeRefactor; Column Fail: number of tests that have different results after the transformation; Result: the result of the SafeRefactor's evaluation.

Subject	Program	KLOC	Compared Versions	Test	Fail	Total Time (s)	Result
9	JHotDraw	23	OO and AO	893	0	132	-
10	JHotDraw	23	OO' and AO	585	120	113	Behavior Change
11	CheckStylePlugin	20	OO and AO	3423	0	129	-
12	CheckStylePlugin	20	OO' and AO	3546	0	117	-

Results

SafeRefactor did not detected behavioral changes in the OO and AO versions in Subjects 9 and 11 (Table 2). In Subject 12, we also did not find a behavioral change between OO' and AO versions of CheckStylePlugin. We improve confidence that these transformations are sound. Table 2 contains the number of generated tests (Column Tests) and the number of tests detecting the behavioral change (Column Fail) for each subject. Column Result indicates whether SafeRefactor identified a behavior change. The symbol - indicates that no behavior change was found.

In our previous work [21], we found a behavioral change in the OO and OO' versions of JHotDraw. In Subject 10, we evaluated the OO' and AO versions, and SafeRefactor also detected this behavioral change. To perform the OO refactoring, developers extracted the code inside the `try`, `catch`, and `finally` blocks to methods in specific classes that handle exceptions. Some classes that implement `Serializable` were refactored.

```
class A implements Serializable {
  Object clone() {
    try { ... }
    catch (IOException e) { ... }
  }
}
```

Developers changed the `clone` method and introduced the handler attribute to handle exceptions. However, they forgot to serialize this new attribute.

```
class A implements Serializable{
  ExceptionHander handler; ...
  Object clone() {
    try { ... }
    catch (IOException e) {
      handler.handle(e);
    }
  }
}
class ExceptionHandler { ... }
```

Thus, when the `clone` method try to serialize the object, an exception is thrown. Therefore, a bug was introduced in the code. On the other hand, in the AO version, developers extracted the exception handling code to aspects. Since it was not needed to introduce new fields in the classes to handle exceptions, this problem did not happen in the AO version. They used tools and a test suite do guarantee behavior preservation. However, we do not have a good tool support for refactoring AO code. It is simple to apply a small transformation and introduce a behavioral change.

4.4 Design Patterns

Subject Characterization

Hannemann and Kiczales [7] implemented 23 design patterns [5] in Java. The same patterns were also implemented in AspectJ. They compared them with respect to locality, reusability, composability, and (un)pluggability. Table 3 summarizes the patterns implemented by them. We use SafeRefactor to evaluate whether their OO and AO implementations are equivalent.

Results

We identified behavioral changes in 4 out of 23 implementations of the design patterns [7] (Table 3). They implemented OO and AO versions of the queue data structure to illustrate the State pattern (Subject 32). This pattern allows an object to behave differently according to its internal state. They implemented the `Queue` class to represent a queue and the abstract class `State` representing the queue states (`Empty`, `Normal`, and `Full`), as depicted by Figure 4.4. Each queue must contain at most three elements.

In the OO version, the state transitions are performed in each class representing a possible state. For instance, the following part of code shows the `insert` method from the `Empty` class, which changes the queue's state to normal and add an element.

```
public boolean insert(Queue queue, Object arg) {
  Normal nextState = new Normal();
  queue.setState(nextState);
```

Table 3. The GoF design patterns implemented in Java and AspectJ.

Subject	Design Pattern	Test	Fail	Total Time (s)	Result
13	Abstract Factory	48	0	10	-
14	Adapter	70	0	7	-
15	Bridge	107	0	8	-
16	Builder	178	0	8	-
17	Chain of Responsibility	8	0	9	-
18	Command	6	0	9	-
19	Composite	234	0	9	-
20	Decorator	84	0	7	-
21	Facade	104	0	8	-
22	Factory Method	59	0	10	-
23	Flyweight	72	0	8	-
24	Interpreter	14	0	8	-
25	Iterator	70	0	9	-
26	Mediator	5	0	0	-
27	Memento	93	0	7	-
28	Observer	60	0	8	-
29	Prototype	131	76	8	Behavior Change
30	Proxy	117	0	8	-
31	Singleton	1	0	7	-
32	State	44	26	8	Behavior Change
33	Strategy	8	0	7	-
34	Template Method	138	54	8	Behavior Change
35	Visitor	27	1	7	Behavior Change

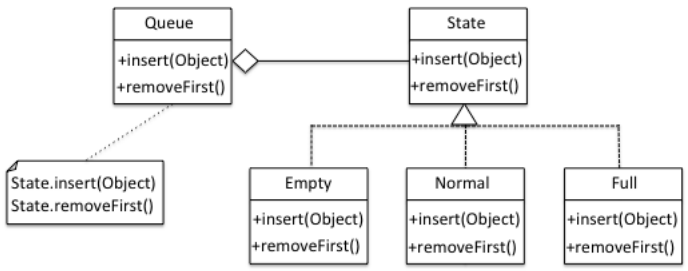

Fig. 4. The class diagram of a Queue using the State design pattern.

```
    return nextState.insert(context, arg);
  }
```

On the other hand, in the AO version, they implemented the state transitions in an aspect. The aspect declares the state objects (empty, normal, full), and an advice makes the state transition after the invocation of the `insert` method.

```
public aspect QueueStateAspect {
  protected Empty empty  = new Empty();
  protected Normal normal = new Normal();
  protected Full full = new Full(); ...
  after(Queue queue, State qs, Object arg):
      call(boolean State+.insert(Object)) && ... {
    if (qs == empty) {
      normal.insert(arg);
      queue.setState(normal);
    } ...
  }
}
```

SafeRefactor detected a behavioral change. Listing 1.4 shows a test case generated by SafeRefactor that reveals a behavioral change. It instantiates the `q1` queue and adds one element to it. Next, it instantiates another queue `q2` and add three elements. All elements were correctly inserted in the OO version. However, the last element could not be inserted into the queue in the AO version. The `r4` variable yields false. It states that the queue is full (it contains three elements).

Listing 1.4. A unit test revealing a behavioral change in the State pattern.

```
public void test() {
  Queue q1 = new Queue();
  boolean r1 = q1.insert("element1");
  Queue q2 = new Queue();
  boolean r2 = q2.insert("element1");
  boolean r3 = q2.insert("element2");
  boolean r4 = q2.insert("element3");
  assertTrue(r1 == true);
  assertTrue(r2 == true);
  assertTrue(r3 == true);
  assertTrue(r4 == true);
}
```

Aspects are singleton by default in AspectJ [10]. Notice that the fields of the `QueueStateAspect` aspect are only instantiated when the aspect is created. Therefore, the same state is shared by all queues. `Normal` contains an array for storing three elements. When we insert an element in `q1`, it is inserted in this array. However, when we create `q2`, this array is not cleared. Therefore, we can only include two elements in `q2`. To avoid this problem, they could have

instantiated an aspect for every new queue instance. AspectJ allows per-object aspects by using the `perthis` and `pertarget` keywords [10].

We also found simple behavioral changes in three design patterns (Subjects 29, 34 and 35). Some methods yield different String messages.

4.5 JML Compiler

Subject Characterization

The Java Modeling Language (JML) [11] is a behavioral interface specification language used to specify contracts, such as pre and post conditions and invariants with annotations. The JML compiler (*jmlc*) reads a Java program annotated with JML and produces an instrumented bytecode with additional code to check the program correctness against restrictions imposed by the JML specification.

Rêbelo et al. [17] propose a JML compiler (*ajmlc*) implemented using AspectJ to avoid using reflection, which was used in jmlc. In this way, they could use JML with Java ME applications, which do not support reflection. Later, they proposed an optimized version of this compiler (*ajmlc optimized*) [18]. They optimized the bytecode size and running time.

We evaluated the JML compilers implemented in AspectJ (ajmlc and ajmlc optimized) using SafeRefactor. We use two Java programs annotated with JML (JAccounting and JSpider) as test inputs for the compilers (Table 4). For each input, SafeRefactor compares the behavior of the programs yielded by these compilers. The programs compiled by ajmlc and ajmlc optimized must be equivalent.

Table 4. Evaluation of JML compilers.

Subject	Program	KLOC	Test	Fail	Total Time (s)	Result
36	JSpider	9	204	32	127	Behavior Change
37	JAccount	6	643	15	118	Behavior Change

Results

SafeRefactor detected behavioral changes in Subjects 36 and 37. The programs compiled using *ajmlc* and *ajmlc optimized* are not equivalent.

A class invariant should be checked before and after a method call. However, it must only be checked after an object is created. By analyzing the tests reported by SafeRefactor, we detected that, *ajmlc* checks invariants before each constructor. This leaded to false invariant violation warnings. For example, consider the following class specifying a person.

```
public class Person  {
  private /*@ spec_public @*/ int height;
  //@invariant height > 0;
  //@pre i > 0;
  public Person(int i) {
    this.height = i;
  } ...
}
```

The `Person` class contains the `height` attribute, and a constructor that sets the height of each person. An invariant states that each person must have a height greater than 0. Moreover, the `Person` constructor has an precondition specifying that the parameter `i` must be greater than 0. Now, suppose we would like to instantiate this class.

```
Person gustavo = new Person(178);
```

The previous code compiled by *ajmlc optimized* is normally executed. However, it throws a warning due to postcondition violation when compiled by *ajmlc*. By analyzing the code generated by the compilers to check the constructor precondition, we notice the *ajmlc* implements this check in an intertype declaration.

```
before (Person p, int i): execution(Person.new()) {
  boolean b = p.checkPrePerson(i); ...
}
boolean Person.checkPrePerson(int i) {
  return (i > 0);
}
```

The `checkPrePerson` method is invoked by an advice before the execution of the constructor. Notice that this method belongs to `Person`. Therefore, by calling it, the invariants of this class will also be checked. However, since the constructor was not initialized so far, the `height` attribute is still `0` leading to an invariant warning.

On the other hand, the *ajmlc optimized* changes the previous checking code by applying the *Inline method intertype within before-execution* refactoring [18]. Part of the resulting code is shown next.

```
before (Person p, int i): execution(Person.new()) {
  boolean b = (i > 0);  ...
}
```

Notice that the inter-type declaration was removed. Therefore, ajmlc contains a bug.

SafeRefactor also detected a behavioral change during a postcondition evaluation of a JAccount method. In a test case generated by SafeRefactor, the code compiled with *ajmlc optimized* throws the `JMLEvaluationError` exception, as expected. However, the code compiled with *ajmlc* throws

`JMLInternalExceptionalPostconditionError`. These exceptions have different meaning. The former occurs when an exception is thrown during the postcondition evaluation, such as `NullPointerException`. The latter notifies an internal exceptional postcondition violations.

4.6 Threats to Validity

Construct validity. Behavioral changes in three design patterns (Subjects 29, 34 and 35) seem to be related not to bugs but to subtle differences in the OO and AO implementations such as the message returned by a method.

Internal validity. SafeRefactor cannot detect behavioral changes in the standard output (System.out.println) messages and exception messages. Moreover, the time limit used for generation tests may have influences in the detection of behavioral changes. Therefore there may be unrevealed behavioral changes in the evaluated subjects.

External validity. Although SafeRefactor generates tests only for common methods, it can be used to evaluated transformation that change method signatures. For instance, consider a rename method from `A.m(..)` to `A.n(..)`. The set of methods that Step 1 identifies doesn't include them. A similar situation occurs when renaming a class. We can't compare the renamed method's behavior directly, but SafeRefactor compare them indirectly if a method in common calls them.

5 Related Work

The term refactoring was coined by Opdyke [15,14] as a behavior-preserving program transformation that improves some quality (reusability, maintainability) of the resulting code. Opdyke [14] proposes a number of refactorings for C++ and specifies conditions to guarantee behavior preservation. However, there was no formal proof of the correctness of these conditions. Later on, Tokuda and Batory [24] found some faults in these refactorings.

Monteiro and Fernandes [12] proposed a catalog of 27 aspect-oriented refactorings [4]. These refactorings aim at introducing aspects and improve the design of them. They can be used as guide for proposing tool supported refactorings. However, they do not prove them sound. We can apply their refactorings and use our tool to improve confidence that the transformation is correct.

Cole and Borba [2] formally specify aspect-oriented programming laws (each law defines a bidirectional semantics-preserving transformation) for AspectJ. By composing them, they derive AspectJ refactorings. Each law formally states preconditions. They proved one of them sound with respect to a formal semantics for a subset of Java and AspectJ [3]. They can be very useful for implementing aspect-aware refactoring tools.

Wloka et al. [25] propose a tool support for extending currently OO refactoring implementations for considering aspects. They developed an impact analysis tool for detecting change effects on pointcuts to generate pointcut updates.

Binkley et al. [1] present a human guided automated approach to refactor OO programs to the AO. Hannemann et al. [8] introduce a role-based refactoring approach to help programmers modularize crosscutting concerns in aspects. These work contribute for improving tool support for refactoring aspect-oriented programs. Our work is complementary to them. We propose a more practical approach for detecting behavioral changes in AO transformations.

Some work have refactor OO programs to AO to investigate benefits of aspects. Hannemann and Kiczales [7] present 23 design patterns [5] in OO and AO. Taveira et al. [23] modularize exception handling in OO and AO code. The study shown that AOP promotes reuse of exception handling code. We used SafeRefactor to evaluate transformations applied in these case studies.

6 Conclusions

Our earlier work presents our technique and its implementation (SafeRefactor) for making OO program refactorings safer [21,20]. Here, we make small changes to SafeRefactor in order to extend it to the AO context. We evaluated SafeRefactor in 37 subjects. SafeRefactor was able to detect a number of non-behavior-preserving transformations [7,23,18]. The transformations presented in Section 4 were applied by refactoring tools or by developers that have a strong background in Java and AspectJ. However, the Java/AspectJ semantics is nontrivial, which imposes challenges in checking and performing refactorings. For instance, pointcuts may use wildcards making difficult to check preconditions, and a small transformation can have an impact on a number of different parts of the program. Therefore, it is not simple to apply them without a good tool support.

As future work, we intend to improve our static analysis for checking the impact of the change. It must take into consideration aspect constructs, such as pointcuts and inter-type declarations. In this way, we can improve performance and generate tests only for the methods affected by the transformation.

Acknowledgment

We would like to thank Paulo Borba, Tiago Massoni, Sérgio Soares, Fernando Castor, Leopoldo Mota, Márcio Ribeiro and Henrique Rêbelo. We gratefully thank the anonymous referees for useful suggestions. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES)[2], funded by CNPq grants 573964/2008-4, 477336/2009-4 and 304470/2010-4.

References

1. Binkley, D., Ceccato, M., Harman, M., Ricca, F., Tonella, P.: Automated refactoring of object oriented code into aspects. In: ICSM. pp. 27–36 (2005)

[2] http://www.ines.org.br

2. Cole, L., Borba, P.: Deriving refactorings for AspectJ. In: AOSD. pp. 123–134 (2005)
3. Cole, L., Borba, P., Mota, A.: Proving aspect-oriented programming laws. In: FOAL. pp. 1–10 (2005)
4. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (2005)
6. Gosling, J., Joy, B., Jr, G.L.S., Bracha, G.: The Java Language Specification. Sun microsystems (2005)
7. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: OOPSLA. pp. 161–173 (2002)
8. Hannemann, J., Murphy, G.C., Kiczales, G.: Role-based refactoring of crosscutting concerns. In: AOSD. pp. 135–146 (2005)
9. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP. pp. 220–242 (1997)
10. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co. (2003)
11. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Software Engineering Notes 31, 1–38 (2006)
12. Monteiro, M., Fernandes, J.: Towards a catalog of aspect-oriented refactorings. In: Proceedings of AOSD '05. pp. 111–122. ACM, New York, NY, USA (2005)
13. Murphy, G.C., Kersten, M., Findlater, L.: How are Java software developers using the Eclipse IDE? IEEE Software 23(4), 76–83 (2006)
14. Opdyke, W.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
15. Opdyke, W., Johnson, R.: Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In: SOOPA. pp. 145–160 (1990)
16. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ICSE. pp. 75–84 (2007)
17. Rebêlo, H., Soares, S., Lima, R., Ferreira, L., Cornélio, M.: Implementing Java modeling language contracts with AspectJ. In: SAC. pp. 228–233 (2008)
18. Rebêlo, H., Lima, R., Cornelio, M.L., Leavens, G.T., Mota, A.C., Oliveira, C.: Optimizing JML features compilation in ajmlc using aspect-oriented refactorings. In: SBLP. pp. 117–130 (2009)
19. Schäfer, M., Ekman, T., de Moor, O.: Challenge proposal: Verification of refactorings. In: PLPV. pp. 67–72 (2009)
20. Soares, G., Gheyi, R., Massoni, T., Cornélio, M., Cavalcanti, D.: Generating unit tests for checking refactoring safety. In: SBLP. pp. 159–172 (2009)
21. Soares, G., Gheyi, R., Serey, D., Massoni, T.: Making program refactoring safer. IEEE Software 27, 52–57 (2010)
22. Steimann, F., Thies, A.: From public to private to absent: Refactoring Java programs under constrained accessibility. In: ECOOP. pp. 419–443 (2009)
23. Taveira, J.C., Queiroz, C., Lima, R., Saraiva, J., Soares, S., Oliveira, H., Temudo, N., Araújo, A., Amorim, J., Castor, F., Barreiros, E.: Assessing intra-application exception handling reuse with aspects. In: SBES. pp. 22–31 (2009)
24. Tokuda, L., Batory, D.: Evolving object-oriented designs with refactorings. ASE 8(1), 89–120 (2001)
25. Wloka, J., Hirschfeld, R., Hänsel, J.: Tool-supported refactoring of aspect-oriented programs. In: AOSD. pp. 132–143 (2008)