A classification of invasive patterns in AOP

Freddy Munoz, Benoit Baudry, Olivier Barais

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A classification of invasive patterns in AOP

Freddy Munoz — Benoit Baudry — Olivier Barais

N° 6501

March 2008

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6501--FR+ENG

A classification of invasive patterns in AOP

Freddy Munoz, Benoit Baudry, Olivier Barais

Thème COM — Systèmes communicants
Équipes-Projets Triskell

Rapport de recherche n° 6501 — March 2008 — 12 pages

Abstract: Aspect-Oriented Programming (AOP) improves modularity by encapsulating crosscutting concerns into aspects. Some mechanisms to compose aspects allow invasiveness as a mean to integrate concerns. Invasiveness means that AOP languages have unrestricted access to program properties. Such kind of languages are interesting because they allow performing complex operations and better introduce functionalities. In this report we present a classification of invasive patterns in AOP. This classification characterizes the aspects invasive behavior and allows developers to abstract about the aspect incidence over the program they crosscut.

Key-words: Aspect-oriented programming, Classification system, Invasive AOP.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Résumé : Pas de résumé
Mots-clés : Pas de motclef

1 Introduction

Aspect-oriented Programming (AOP) is a paradigm that enhances current approaches to modularizing software. AOP enables separation of concerns that crosscut the implementation of a system. This is done by encapsulating crosscutting concerns into single units called *aspects*. An aspect itself is composed of several units that realize the crosscutting behavior. Aspects also provide pointing elements that designate well defined points in the program execution or structure. These are the points where the program executes the crosscutting behavior. Different approaches to AOP have been proposed [5, 6, 10, 9, 7, 1]. Each of these approaches provide a different mechanism to compose aspects with the base program. These composition mechanisms range from simple program augmentation to more complex operations such as behavior replacement.

Invasive composition mechanism allow developers to manipulate almost any structure and behavior of the base program. *Invasive AOP* approaches realize invasive composition mechanism. *Invasiveness* is the ability of invasive AOP to manipulate the program structures and behavior. Invasive AOP provides several strategies to manipulate the base program. These strategies range from less invasive such as the augmentation of a procedure execution to more invasive ones such as the replacement of a procedure execution.

This report presents a classification system for invasive AOP. This classification system focuses on the invasive behavior of aspects over the base program. We construct this classification system by studying the invasive structures of AspectJ [3], however, we think that it can be adapted to other invasive languages.

The reminder of this report is organized as follows. In section 2 we study the AspectJ structures in order to derive our classification of invasive AOP. In section 3 we introduce our classification of invasive aspects. In section 4 we discuss related work. Finally section 5 concludes.

2 Classifying Invasive AOP

Each programming paradigm promotes its own way to encapsulate concerns. For example, the object-oriented paradigm encapsulates concerns in classes, fields and methods. It also defines a protection level for each one of them. This ensures that properties will be accessed according to their permission.

Invasiveness breaks the encapsulation that each paradigm promotes and provides a mean to access protected properties. For example, aspects can access classes, methods and fields whatever their permission. Developers can fruitfully use invasiveness. For example, they can use it to check the preservation of system properties [11]. However, developers can use invasiveness to invalidate some desirable system properties, intentionally or unintentionally.

The duality in the usage of invasiveness motivates us to propose a classification system for invasive AOP. Our classification system provides the following benefits.

- It enables the characterization of aspects with precise invasive behavior (invasive pattern).

- It enables developers to reason about aspects and discriminate when they could represent a risk to the base program.

We build the classification system by studying the invasive structures available in AspectJ. Once identified the different possibilities it provides, we proceed to derive our classification system.

2.1 Running example

To illustrate our characterization approach, in listing 1 we present the skeleton implementation of an ordered list class (`MyArrayList`). All the attributes in this list are private to the class. Elements added using the `add` method (line 8) are stores in the `elements` array (line 2). Elements are retrieved by using the `get` method (line 11). The specific position of an element in the `elements` array is obtained by using the `indexOf` method (line 12). An element is removed from the list by using the `remove` method (lines 13-14), this method removes an element by using its position or the original element. The first and last element in the list are retrieved by using the methods `first` and `last` (lines 15,16). All the elements in the list can be removed by using the `clear` method (line 9).

```
1  public class MyArrayList {
2          private Object[] elements;
3          private int initialSize=0;
4          private int growFactor=5;
5          private int size=0;
6          public MyArrayList(int grow,int initialSizei){...}
7          public MyArrayList(){..}
8          public boolean add(Object o) {...}
9          public void clear() {...}
10         public boolean contains(Object o) {...}
11         public Object get(int index) {...}
12         public int indexOf(Object o) {...}
13         public boolean remove(Object o) {...}
14         public Object remove(int index) {...}
15         public Object first() {...}
16         public Object last() {...}
17         public int size() {...}
18 }
```

Listing 1: Skeleton implementation of an ordered list example

Listing 2 presents the code of our example aspect. This aspect (`MyAspect`) realizes a set of advices an Inter-type declarations that affect in different ways the `MyArrayList` class. The declaration in line 3 introduces the parent interface `Comparable` into the `MyArrayList` class hierarchy. The inter-type in line 5 adds the method `compareTo` to the `MyArrayList` class. Analogously, the inter-type of line 7 adds the field `maxSize`. The advice in lines 9-12 execute a logging facility just before the execution of the `add` method. The advice in lines 14-19 print a message into the standard output before and after the execution of the `add` method. The advice in lines 21-24 replaces the execution of the `first` method and executes the `last` method instead. The advice in lines 26-32 executes the `add` method only when non null values are passed as parameters. The advice in

lines 34-42 adds each element of a collection into the list. The advice in lines 44-48 replaces the parameter object (object to be inserted) by an string, then the string is inserted instead of the object. The advice in lines 48-52 reads the field current size and print it to the standard output. Finally, the advice in lines 54-57 decreases the value of `currentSize` by 1.

```
1  public privileged aspect MyAspect{
2
3          declare parents: MyArrayList implements Comparable;
4
5          public boolean MyArrayList.compareTo(Object o){...}
6
7          public int MyArrayList.maxSize=100;
8
9          before(MyArrayList list,Object o):
10                 execution(* MyArrayList+.add(Object+)) && args(o) && this(list){
11                 logger.debug("Adding object "+o.toString());
12         }
13
14         void around((MyArrayList list,Object o):execution(* MyArrayList+.add(Object)) &&
15                 args(obj) && target(list){
16                 System.out.println("adding object:"+obj);
17                 proceed(obj,list);
18                 System.out.println("object added ");
19         }
20
21         boolean around(MyArrayList list):
22                 execution(* MyArrayList+.first()) && this(list){
23                 return list.last();
24         }
25
26         boolean around(MyArrayList list,Object o):
27                 execution(* MyArrayList+.add(Object+)) && args(o) && this(list){
28                 if(o!=null){
29                   return proceed(list,o);
30                 }
31                 return false;
32         }
33
34         boolean around(MyArrayList list,Collection c):
35                 call(* MyArrayList+.add(Collection+)) && args(c) && this(list){
36                 for(Object o:c){
37                    proceed(list,o);
38                 }
39                 return true;
40         }
41
42         boolean around(MyList list,Object o):
43                 call(* MyArrayList+.add(Object+)) && args(o) && this(list){
44                 o = new String("object"+ o.toString());
```

```
45              return proceed(list,o);
46          }
47
48          after(MyArrayList list):
49              execution(* MyArrayList+.add(..)) && this(list){
50              int size=list.currentSize;
51              logger.debug("current size: "+size);
52          }
53
54          before(MyArrayList list):
55              execution(* MyArrayList+.add(..)) && this(list){
56              list.currentSize=list.currentSize-1;
57          }
58
59          boolean around(MyArrayList list,int i):
60              call(* MyArrayList+.get(int)) && args(i) && this(list){
61              Object o=proceed(list,i);
62              return o.toString();
63          }
64  }
```

Listing 2: MyAspect example

2.2 Invasive AOP: The case of AspectJ

AspectJ [3] is the most prominent realization of invasive aspects. It realizes the crosscutting behavior on *advices* and designating the places where this behavior must be woven with pointcut expressions. Pointcuts are regular expressions that match well defined point in the structure (static) and execution (dynamic) of the program.

In AspectJ, aspects are composed of advices, pointcuts, inter-type declarations and inter-type parent declarations. Advices are used to modify the program flow and write or read fields. An advice can be declared inside a privileged aspect, which means that it is enabled to ignore the object-oriented access policy. Inter-type declarations are used to introduce methods and fields into a target class. Inter-type parent declarations are declarations of inheritance that modify the class hierarchy.

AspectJ allows developers to implement various behaviors on advices. We summarize them by using the actions advices may perform in the following list.

- The advice does not modify the behavior of the methods it crosscuts. (Listing 2, lines 9-12, 14-19)
- The advice replaces the behavior of the methods it crosscuts, the original methods are never invoked. (Listing 2, lines 21-24)
- The advice conditionally replaces the behavior of the methods it crosscuts. (Listing 2, lines 26-32)
- The advice replaces the behavior of the methods it crosscuts. It invokes two or more time the original method. (Listing 2, lines 34-40)

- The advice invokes a method declared outside its declaring aspect. (Listing 2, line 11)
- The advice changes the argument values of the methods it crosscuts. (Listing 2, lines 42-46)
- The advice replaces the result of the method execution. (Listing 2, lines 59-63
- The advice reads or writes object fields values. (Listing 2, lines 50, 56)

AspectJ allows developers to modify the program structure. We summarize the modifications developers can implement on aspects in the following list. It is worth to remark that these modifications are performed at the aspect level.

- The aspect inserts new fields into a target class by using an inter-type declaration. (Listing 2, line 7)
- The aspect inserts new methods into a target class by using an inter-type declaration. (Listing 2, line 5)
- The aspect modifies the class hierarchy by using an inter-type parent declaration. (Listing 2, line 3)

These list of interventions range from simple augmentation to more invasive ones like complete replacing the target method or write a protected field. We motivate our classification on these factual intervention we proceed to identify different invasive behavior that we call *invasiveness patterns*.

3 Classification System

We can identify three axes for our classification. The first corresponds to how developers can use advices to affect the program control flow. The second corresponds to how developers can use advices to affect the data flow, changing fields, methods parameters and returning values. The third corresponds to how developers can use aspects to modify the program structure.

3.0.1 Control Flow

Classifies advices according to the behavioral interaction between them and the methods it crosscuts.

Augmentation : After crosscutting, the body of the method is always executed. The advice augments the behavior of the method it crosscuts with new behavior that does not interfere with the original behavior. The advices in Listing 2, lines 9-12, 14-19 realize the invasiveness pattern *Augmentation*.

Replacement : After crosscutting, the body of the method is never executed. The advice completely replaces the behavior of the method it crosscuts with new behavior. This kind of advices eliminate a part of the base code. The advice in Listing 2, lines 21-24 realizes the invasiveness pattern *Replacement*. The advice in lines 59-63 is an special case of *Replacement*, this because it executes the original behavior to later replace the obtained result.

Conditional replacement : After crosscutting, the body of the method is not always executed. The advice conditionally invokes the body of the method and potentially replaces its behavior with new behavior. Examples of this kind of advices are advices realizing transaction, access control, etc. Listing 2, lines 26-32 realizes the invasiveness pattern *Conditional replacement*.

Multiple : After crosscutting, the body of the method is executed more than once. The advice invokes two or more time the body of the method it crosscuts generating potentially new behavior. The advice in Listing 2, lines 34-40 realizes the invasiveness pattern *Multiple*.

Crossing : After crosscutting, the advice invokes the body of a method (or several methods) that it does not crosscut. The advice have a dependency to the class owning the invoked method(s). The advice in Listing 2, lines 9-12 realizes the invasiveness pattern *Crossing*.

3.0.2 Data Access

Classifies advices according to the access they perform to object fields and method arguments.

Write : After crosscutting, the advice writes an object field. This access breaks the protection declared for the field and can modify the behavior of the underlying computation. The advice in Listing 2, lines 54-57 realizes the invasiveness pattern *Write*.

Read : After crosscutting, the advice reads an object field. This access breaks the protection declared for the field and can potentially expose sensitive data. The advice in Listing 2, lines 48-52 realizes the invasiveness pattern *Read*.

Argument passing : After crosscutting, the advice modifies the arguments of the method it crosscuts and then invokes the body of the method. The body of the method always executes at least once. The advice in Listing 2, lines 42-46 realizes the invasiveness pattern *Argument passing*.

3.0.3 Structural

Classifies aspects according to the modification performed they perform to the existing structure of a class.

Hierarchy : The aspect modifies the declared class hierarchy. For example, the aspect adds a new parent interface to an existing one. An example of this is the declaration in Listing 2, line 3.

Field addition : The aspect adds new fields to an existing class declaration. These fields depending on their protection can be acceded by referencing an object instance of the affected class. An example of this is the declaration in Listing 2, line 7.

Operation addition : The aspect adds new methods to an exiting class declaration. These methods depending on their protection can be acceded by referencing an object instance of the affected class. An example of this is the declaration in Listing 2, line 5.

The groups we have presented describe the aspect invasive behavior in different dimensions, and most of them are complementary. For example, an advice cannot perform data access without executing and therefore a Write advice can also realize and Augmentation advice. However, some classification categories are incompatible, *Augmentation*, *Replacement* and *Conditional Replacement* cannot co-exist in the same advice.

Note that this classification is still incomplete. It does not address cases like the exceptions raised by advices. However, it allows us to characterize the behavioral interaction between advices and methods, major structure modifications and data access.

4 Related work

In [8] categories of direct and indirect interactions between aspects and methods are identified. Direct interaction is whether an advice interferes with the execution of a method, whereas indirect is whether advices and methods may read/write the same fields. This classification is similar to ours, however, it addresses a different dimension. We identify invasiveness patterns instead of direct/indirect interactions.

Katz [4] recognizes the fact that aspects can be harmful to the base code and the need of specification on aspect-oriented applications. Our approach agrees with his ideas and likewise we propose a mean to write such specifications. Furthermore, he describes three groups of advices according to their properties. *Spectative* aspects, which do not influence the underlying computation, *Regulatory* aspects, which change the control flow but do not affect existing fields, and *Invasive* aspects, which affect existing fields. This classification is similar to ours, however, our characterization of is more fine grained. The two first correspond to our behavioral classification and the last to our data access classification.

Clifton and Leavens propose *Spectators* and *Assistants* [2]. Spectators are advices that do not affect the control flow of the advised method and do not affect existing fields. Assistants can change the control flow of the advised method and affect existing fields. *Spectator* are similar to our classification category *Augmentation* and *Read* in the sense that they do not interfere with the mainline computation or write fields. All other classification categories are equivalent to *Assistants*. Nevertheless, we have achieved a more fine granularity level in our classification.

5 Conclusions and Future work

In this report we have presented a characterization of invasive behavior. Such a characterization is encoded as a classification of invasive aspect oriented programing. The component of this classification represent invasiveness patterns, i.e. the ways in which AOP can be invasive.

The characterization of invasive aspects allow developers to recognize and reason about the potential risks introduced by aspects into the base program. We think that this is the first step to a bigger specification framework that will

make developers more confident in AOP. In future work we will explore the creation of an aspects - base program specification framework.

References

[1] Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. Adapting virtual machine techniques for seamless aspect support. In ACM, editor, *ACM Sigplan International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA) 2006*, 2006.

[2] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In Ron Cytron and Gary T. Leavens, editors, *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, pages 33–44, March 2002.

[3] http://www.aspectj.org. Aspectj.

[4] Shmuel Katz. Diagnosis of harmful aspects using regression verification. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages*, pages 1–6, March 2004.

[5] Gregor Kiczales. Aspect oriented programming. *ACM SIGPLAN Notices*, 32(10):162–162, October 1997. No paper in volume, but table of contents includes an entry for this invited talk.

[6] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM.

[7] Renaud Pawlak. Spoon: annotation-driven program transformation — the aop case. In *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*, New York, NY, USA, 2005. ACM.

[8] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for interactions in aspect-oriented programs. In *Foundations of Software Engineering (FSE)*, pages 147–158. ACM, October 2004.

[9] Davy Suvée, Bruno De Fraine, and Wim Vanderperren. A symmetric and unified approach towards combining aspect-oriented and component-based software development. In Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, and Kurt C. Wallnau, editors, *CBSE*, volume 4063 of *Lecture Notes in Computer Science*, pages 114–122. Springer, 2006.

[10] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM.

[11] Dean Wampler. Contract4J for design by contract in Java: Design pattern-like protocols and aspect interfaces. In Yvonne Coady, David H. Lorenz, Olaf Spinczyk, and Eric Wohlstadter, editors, *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 27–30, Bonn, Germany, March 20 2006. Published as University of Virginia Computer Science Technical Report CS–2006–01.

Contents

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Futurs : Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399