Improving Maintenance in AOP Through an Interaction Specification Framework

Freddy Munoz, Benoit Baudry, Olivier Barais

Improving Maintenance in AOP Through an Interaction Specification Framework

Freddy Munoz, Benoit Baudry
INRIA
Campus de Beaulieu
F-35042 Rennes Cedex, France
{fmunoz,bbaudry}@irisa.fr

Olivier Barais
Université de Rennes 1
Campus de Beaulieu
F-35042 Rennes Cedex, France
barais@irisa.fr

Abstract

The invasiveness of aspects is beneficial to modularize crosscutting concerns that require the modification of the data or control flow. However, it introduces subtle errors that are hard to locate and fix in case of evolution. In this paper we illustrate this issue by evolving a program implemented using aspects. Interaction issues, between aspects and the program, emerge from this evolution. We locate them through manual inspection and test execution. This tedious process motivates the need for an abstract specification of intended interactions. To tackle this issue, we propose a framework for specifying the types of invasiveness pattern that are allowed of forbidden in the program. We have also implemented a tool that automatically checks whether the specification is satisfied by the aspects.

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 realizing the crosscutting behavior, these units are called *Advices*. 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. Generally the pointers are called *point-cut* and the execution points *Join-points*. Different approaches to AOP have been proposed [8, 10, 13, 3]. Each of these approaches provide a different mechanism to compose aspects with the base program (main concern). These composition mechanisms range from simple program augmentation to more complex operations such as behavior replacement.

Invasive AOP approaches use composition mechanisms that allow developers to manipulate almost any structure of the base program. This ability to manipulate the base program structures is called *invasiveness*. 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. An *invasiveness pattern* is the characterization of invasive behavior, i.e. the strategy or a combination of them to manipulate the base program.

Invasive aspects are useful to introduce functionalities that otherwise must be hardcoded into the base program. For example, a system transaction concern is implemented using an invasive aspect because it requires to stop executing the intercepted behavior each time a transaction fails. However, invasive aspects can also do harm to the base program. When they introduce the functionalities they are designed for, they can also introduce side effects, hence, generating unexpected interactions.

The work presented in this paper is divided in two parts. In the first part we illustrate the issues that can arise when evolving an aspect-oriented program that is built with aspects (some of them invasive). This is an special case of the *AOSD-Evolution paradox* [17], which results to be aggravated in presence of invasive aspects. We consider a distributed chat application as a case study. First, we illustrate how invasive aspects are useful to implement cross-cutting features in this example. Then we evolve the system to enable authentication in the application. While regression testing the application errors emerge. We then propose an iterative process involving manual inspection and test execution in order to locate the source of the problem. It allows us to track back the issues to unexpected interactions between aspect and the program. From this study and the debugging process we learn that invasive aspects can introduce faults that emerge only when evolving the system. We also learn that these faults are difficult to locate and that debugging requires a complex and error-prone process. Moreover, the crosscutting nature of aspect makes difficult to reason about their impact in interaction with the base program. This study demonstrates the need to reason about expected interactions, and to control the usage of invasive aspects.

In the second part of this work we propose a framework for specifying the expected interactions in the base program as well as the way in which aspects can be invasive. The specification on aspects is based on a classification proposed in previous work [11, 12]. This classification identifies different patterns according to which AspectJ aspects can be invasive. Based on this specification, the base program declares which type of invasiveness it allows or forbids. We have developed a tool support for this framework. This tool analyzes aspects to infer which invasive pattern they encapsulate. It can also statically check that the aspects conform to the specification of the base program. Based on this approach, we revisit the chat application and illustrate that specifying interactions is useful to early detect when invasive aspect can perform harmful. This experience expresses that it is worth specifying the interactions between aspect and base program. Besides, having specification reduces the time to locate and fix issues consequence of unexpected interaction.

The contributions of this work are summarized as follows:

- An illustration through a rigorous inspection process on a case study of the *AOSD-Evolution paradox* in presence of invasive aspects.
- A framework for the characterization of invasiveness patterns on aspects and the specification of expected invasiveness patterns in the base program.

This paper is organized as follows. Section 2 explain the chat application case study and a process to detect problems introduced by invasive aspects. Section 3 explains our specification framework. Section 4 describes the implementation of the specification framework. Section 5 revisits the case study using the specification framework. Section 6 presents the related work and finally section 7 concludes.

2. Motivating Case Study

In this section we illustrate the AOSD-evolution paradox [17] in presence of invasive aspects. Our goal is to show that invasive aspects offer efficient mechanisms to implement cross-cutting concerns, but that they can also introduce complex errors in case of evolution.

To illustrate these issues, we present an example implemented in Java and AspectJ. The example is a chat application implemented with 5 aspects. We run system-level test cases on the application to validate the initial version. Then, we evolve the application adding authentication capabilities. While testing the new version, we have detected errors.

We precisely discuss the analysis we perform to trace the source of the error back to a wrong interaction between aspects and the base program. Based on these observations, we motivate our approach to assist the validation and verification of aspect-oriented programs.

2.1. A chat application

A chat application is a program that allows users to communicate with each other in real time. Our chat is composed of two parts: a client and a server. The server handles client, manages the communication between them and ensures their uniqueness. The client transmits messages to the server that are dispatched to other clients. The global behavior of a *chat application* can be described as follows. Initially, the server is waiting for clients. The establishment of communications is called association. The cease of communication is called disassociation. Before associating a client, the server checks the uniqueness of the client's nickname. Clients using existing nicknames are not associated. Once associated, the clients can send messages. Such messages are encrypted and decrypted by the clients. Each client's chat session is recorded in a log file. The server also stores each session into a log file. A graphical user interface (GUI) controls the client and server behavior.

2.2. Initial version

We have implemented a chat in Java^{TM}. From the requirements documents, we separated the core concerns form the cross-cutting concerns. Figure 1 shows the class diagram for the core concerns of the chat application.

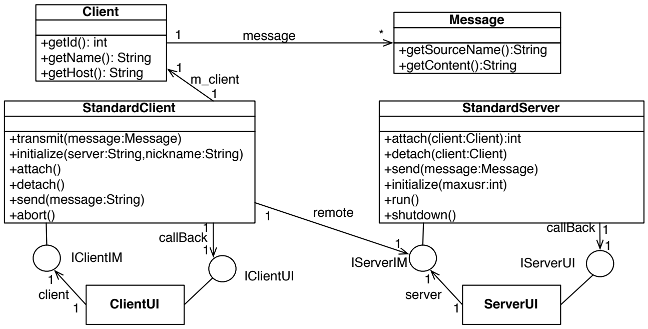


Figure 1. *Chat application* **class diagram**

`IClientIM` and `IServerIM` are the remote interfaces for the client and the server. The `StandardClient` class realizes the client interface. The methods `attach()` and `detach()` associate and dissociate the client to a server. The method `transmit(Message)` notifies the GUI about the arrival of a new message. Finally the method `abort()` aborts the execution of the client. The `StandardServer` class realizes the server interface. As in the client, the methods `attach(Client)` and `detach(Client)` associate and dissociate a client. The `send(Message)` method dispatches the messages to other clients associated to the server. The class `Client` is a container for the client's information. The class `Message` supports the messaging mechanism between client and server. Client and Server communicate using the Java RMI distribution mechanism [16]. The client and the server GUI are implemented with the Standard Widget Toolkit (SWT).

2.3. Crosscutting concerns

We have identified the following crosscutting concerns and implemented them with AspectJ.

- ***Message encryption*** encrypts and decrypts the incoming/outgoing messages. Encryption/decryption occurs just before the execution of the methods `send` and `transmit`. It replaces the method arguments with the encrypted/decrypted version of the message.
- ***Message logging*** logs the incoming/outgoing messages for each user. Its behavior executes before transmitting a message and after receiving a message (before the encryption and after the decryption).
- ***Error handling*** captures the communication exceptions and raises an alert indicating communication problems. Its behavior executes after an exception of type remote is thrown.
- ***Server logging*** logs the server activity in a file. Its behavior executes before and after all the methods of the `IServerIM` interface. This is, it observes the execution of each method of the server.
- ***Nickname uniqueness*** checks the existence of only one nickname in the server. Its behavior executes just before the server association methods (`attach` and `detach`). In the case of association, it checks the existence of the client's nickname on the server. If the nickname exists, it throws an exception. Otherwise, it adds the name to a list and executes the association normally. In the case of dissociation, it removes the client's nickname from a list.

```
1   public aspect EncryptionAspect{
2    pointcut encryptMessage(String string):
3       execution(void IClientIM.send(String)) && args(string);
4    pointcut reencryptMessage(Message messg):
5       execution(void ∗.retransmit(Message)) && args(messg);
6    pointcut decryptMessage(Message messg):
7       execution(void IClientIM.transmit(Message)) && args(messg);
8    void around(String arg) : encryptMessage(arg){
9       arg = encrypt(arg);
10      proceed(arg);
11   }
12   void around(Message message) : reencryptMessage(message){
13      message.sContents=encrypt(message.sContents());
14      proceed(message);
15   }
16   void around(Message message) : decryptMessage(message){
17       message.sContents=decrypt(message.sContents());
18      proceed(message);
19   }
20  }
```

Listing 1. Implementation of the encryption concern

Listing 1 shows the implementation of the *Message encryption* concern. It is clear from the code that the three advices of this aspect are changing the argument values of the intercepted methods (lines 9-10, 13-14, 17-18). The original message is replaced by the encrypted/decrypted version and then it is re-injected to the original method (`proceed` call). This concern can only be implemented by using invasive aspects, otherwise it must be hard-coded in the base program.

```
1   public aspect UniqueNameAspect issingleton() {
2    public pointcut ensureUniqueness(IServerIM serv,Client client):
3       execution(∗ IServerIM.attach(Client)) &&
4       target(serv) && args(client);
5    public pointcut removeFromList(IServerIM serv,Client client):
6       execution(∗ IServerIM.detach(Client)) &&
7       target(serv) && args(client);
8    private static ArrayList nameList=new ArrayList();
9    int around(IServerIM serv,Client client)
10      throws UsedNameException:ensureUniqueness(serv,client){
11      int retValue=−1;
12      if(!nameList.contains(client.getSName())){
13         nameList.add(client.getSName());
14         retValue=proceed(serv,client);
15      }
16      else{ throw new UsedNameException();}
17      return retValue;
18   }
19   after(IServerIM serv,Client client): removeFromList(serv,client) {
20      nameList.remove(client.getSName());
21   }
22  }
```

Listing 2. Implementation of the unique nickname concern

Listing 2 shows the implementation of the *Nickname uniqueness* concern. This aspect manages a list of the currently associated clients nicknames (`nameList` line 8). If the actual client nickname does not exist in the list (line 12) then it is added to the list (line 13) and the intercepted method is executed. Otherwise, the method is never executed and an exception is raised. This aspect conditionally replaces the execution of the methods it intercepts. Analogously to the *Message encryption* concern, it can only be implemented by using invasive aspects.

These examples illustrates that invasive aspects help implementing the crosscutting concerns that modify the flow or data in the program.

2.4. Validating the initial version

We test the initial version with 7 system-level test scenarios. Each scenario validates a different dimensions of the system. These dimensions are summarized as follows:

1. The association mechanism between client and server.
2. The association mechanism supports multiple clients.
3. Clients can send/receive messages.
4. The server distributes the messages among clients.
5. The server detects clients named with an existing nickname.
6. The server association mechanism removes a used nickname when disassociating a client.

7. The error handling mechanism handles the exceptions.

All the test scenarios pass on the initial version composed of the core concern and 5 aspects.

2.5. Evolving the chat application

The initial version of the chat application allows any user to associate with a server. Here, we add an authentication mechanism to ensure that only the registered users are able to associate with a server. We also want to ensure that the clients of this new version are compatible with the old version. As a consequence, a server without authentication must be able to associate an authenticated client. Authenticated servers must refuse the association of unauthenticated clients. The chosen authentication protocol proceeds as follows: the client provides the nickname and password data to the server. The server checks the nickname, password pair internally. If the pair is authentic, then the server will associate the client, otherwise the client is not associated.

In order to implement the evolution, we have added two classes: `AuthenticatedSever` that extends `StandardServer` and overrides the method `attach(Client)`; `AuthenticatedClient` that extends `StandardClient` and overrides the methods `transmit(Message)` and `receive(Message)`. Additionally, we have performed minor changes in the class Client and the client GUI adding support for password. Since this evolution is only adding new behavior, the behavior for the initial version should be kept. The crosscutting concerns present in the initial version remains in this evolution with no further implementation changes.

2.6. Validating the new version

We use the previous test scenarios for regression testing. To do so, we replace the standard client/server with the authenticated one. We also add 5 scenarios to validate the authentication mechanism as well as the compatibility between the two versions. The dimensions addressed by these scenarios are summarized as follows:

8. The authentication mechanism detects invalid nicknames and passwords.
9. The server detects void nicknames or password (or both).
10. The client association mechanism is compatible with the standard server.
11. The authenticated server is incompatible with the standard client.
12. The authenticated client messaging mechanism is able to send messages to standard clients.

The results of the executing tests 1 to 12 are summarized in table 1.

Table 1. Test results after evolution

Test	1	2	3	4	5	6	7	8	9	10	11	12
	x	x	x	x	x	x	x	✓	✓	✓	✓	x

2.7. Reasoning about the problems

Looking at the results of the test scenario execution it is not easy to see where the problems are located. However, a rigorous manual analysis will help detecting the problems. We deal with the authenticated association issues and later the compatibility issues.

The failure of the two first test cases tells that the client cannot associate with the server or the server cannot authenticate/associate the clients. The failure of tests 3, 4, 5, 6 and 7 are consequence of the failure of the test case 1 and 2. This is because the preconditions cannot be fulfilled: there is no connected client. The success of tests 8 and 9 provides no information about the association mechanism. The success of the first test is fundamental to obtain more information about tests 3 to 7. This guides our analysis to examine the association and the authentication mechanism.

After rigorously inspecting the base program we found no errors, however, we need to take into account that the chat runs with aspects. Before examining the aspects code, we try to run the server without aspects that could interfere with the association/authentication mechanism. We remove only the invasive aspects because they are the only ones that can change the final behavior of the application. Moreover, the test scenarios that fail (except for the test case 5) evaluate only the functionalities implemented in the base program.

Table 2 shows the test results after removing the *Nickname uniqueness* aspect. Now test cases 1 to 4 pass, however, the test case 5 fails because it depends on the aspect. Removing the aspect helped to localize the problem (the *Nickname uniqueness* aspect is interfering with the association mechanism). Nevertheless, we still ignore the specific localization of the failure and its cause.

After rigorously inspecting the aspect code and the places it affects, we finally localize the specific cause of the problem. The problem occurs because the aspect captures a wrong join point, the execution of the `attach` method in the `StandardServer` class. The point-cut descriptor captures all the executions of the `attach` method in the server. This, while the body of the advice realizing the crosscutting concern is designed to be executed just once at each join point. This means, once from the beginning to the end of the method execution.

Table 2. Test results after removing the *Nickname uniqueness* aspect

Test	1	2	3	4	5	6	7	8	9	10	11	12
	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	x

Figure 2 depicts schematically the association

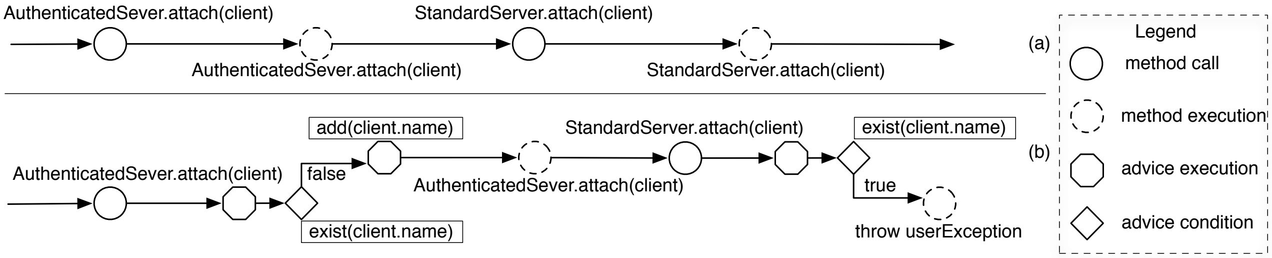

AuthenticatedSever.attach(client)
AuthenticatedSever.attach(client)
StandardServer.attach(client)
StandardServer.attach(client)
(a)
add(client.name)
StandardServer.attach(client)
exist(client.name)
AuthenticatedSever.attach(client)
false
AuthenticatedSever.attach(client)
true
exist(client.name)
throw userException
(b)
Legend
method call
method execution
advice execution
advice condition

Figure 2. Schematic view of the association flow. (a) association without the *Nickname uniqueness* aspect. (b) association with the *Nickname uniqueness* aspect.

flow without (a) and with (b) the *Nickname uniqueness* aspect. In the case with the aspect (b), it captures the first call that performs the authentication (`AuthenticationServer.attach(client)`). Then, the aspect finds that the client nickname is not in the registry (`exist(client.name)=false`) and adds its name into the registry (`add(client.name)`). The aspect also captures the second call that performs the association (`StandardServer.attach(client)`). It is here where the aspect finds that the name exists in the registry and throws an exception. This problem is hard to detect because it can be a combination of a failure in the point-cut descriptor and a limited implementation of the advice. To solve this problem we could modify the advice implementation or the point-cut descriptor. However, we think that the advice implementation realizes in a proper manner the crosscutting concern. Therefore, we modify the point-cut description.

Once we had detected and fixed the association problem, we explore the compatibility problems. The failure of test 12 is due to a compatibility problem when sending/receiving messages. Following the procedure we used to localize cause of the association issue we localize the case of the compatibility issue. The problem is localized in the *Encryption* aspect and is analogous to the association problem. The advice captures the execution of the messaging methods twice, therefore, it encrypts/decrypts the messages twice. On the other hand, the standard client encrypts/decrypts the messages only once. We solve this problem by changing the point-cut descriptor to match only one execution each time.

2.8. Discussion

Through this experiment we have shown that despite the features provided by invasive aspects, they can hamper the software evolution. The main issue is that it is very hard to reason about the aspects impact on the final application, and it is very hard to trace them as the source of the problems to aspects.

The successive execution of a set of test scenarios and the manual inspection of code helped us tracing the problems to aspects. However, this process is tedious, time consuming and error prone. Besides, there is no generic formula to trace this kind of problems. In the chat application, the problems were localized by removing the aspect. This was possible because the test scenarios we used were testing the base program functionalities. Nevertheless, it is not always possible to simply remove the aspects.

Actually there is a missing element in the AOP support. The tedious process we performed tells us that there is a need to abstract from code to reason about the interactions between aspects and the base program. To tackle this issue, we propose a framework for specifying the (1) the invasiveness patterns that aspects realize, and (2) the invasiveness patterns expected on the base program. This means specifying the interaction between aspects and base program. Such specification assist developers to localize and solve problems due to faulty invasive aspects.

3. Specifying aspects-base program interaction

The specification of interactions between aspects and base program consists of two parts. In the first, we characterize aspects with specific invasiveness patterns (*Aspect specifications*). In the second, we specify the invasiveness patterns the base program allows from aspects.

For the first we propose to use our previous work [12] on classifying invasive aspects. For the second we propose to specify assertions that allow/forbid invasiveness patterns to interact with the base program elements.

The goal of these specifications is to obtain information about the potential unexpected interactions that invasive aspects can produce. Such information will help developers to reason about the harmfulness of aspects. Therefore, it assists developer to track potential faults (introduced by invasive aspects).

3.1. Aspect specification

In [11, 12] we present a classification of invasive aspects. Such classification is the result of an analysis of the invasive mechanisms that AspectJ [6] provides. It allows us to identify specific invasiveness patterns and therefore abstract from code. Such abstraction helps reasoning about the interaction of aspects and the base program. We use this classification to characterize aspects with invasiveness patterns.

In AspectJ, aspects crosscut the base program at two levels. At class level (aspect) modifying the program structure and at method level (advice) manipulating the method's behavior. Our classification addresses these two levels. In the following, we list the classification elements with a brief description. Aspect invasiveness patterns are marked with ‡, and advice invasiveness patterns are marked with †.

- † *Augmentation*: After crosscutting, the body of the intercepted 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. Examples of this kind of advices are those realizing logging, monitoring, traceability, etc.
- † *Replacement*: After crosscutting, the body of the intercepted 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 program.
- † *Conditional replacement*: After crosscutting, the body of the intercepted 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.
- † *Multiple*: After crosscutting, the body of the intercepted method is executed more than once. The advice invokes two or more time the body of the method it crosscuts generating potentially new behavior.
- † *Crossing*: After crosscutting, the advice invokes the body of a method (or several methods) that it does not intercepts. The advice have a dependency to the class owning the invoked method(s).
- † *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.
- † *Read*: After crosscutting, the advice reads an object field. This access breaks the protection declared for the field and can potentially expose sensitive data.
- † *Argument passing*: After crosscutting, the advice modifies the argument values of the method it crosscuts and then invokes the body of the method. The body of the method always executes at least once.
- ‡ *Hierarchy*: The aspect modifies the declared class hierarchy. For example, the aspect adds a new parent interface to an existing one.
- ‡ *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.
- ‡ *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.

All the advices of the *Encryption* aspects (Listing 1) are classified *Argument passing*. This because they modify the argument values of the methods they intercept. The advices of the *Nickname uniqueness* aspect (Listing 2) are classified *Conditional Replacement* (lines 9-18) and *Augmentation* (lines 19-21). The first conditionally replaces the execution of the methods it intercepts, the second is orthogonal to the methods it intercepts.

3.2. Core specification

We specify the base program (core) by asserting the patterns of invasiveness allowed/forbidden to interact with it. Such specification emanates from the base program designers and declares an expected interaction.

Aspects crosscut the base program at the level of classes (modifying the class structure), methods (modifying the declared behavior) and fields (accessing the data contained in object fields). This motivates us to attach specifications to each one of these elements. By default (implicit specification) only the patterns *Augmentation*, *Crossing*, *Read* are allowed to advise the base program. This is because a priori these classes do not alter the program flow, data or structure, hence, they are less harmful than the others.

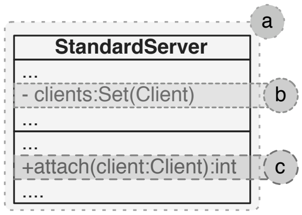


Figure 3. Base program specification covering

Specification for classes: Specify the invasiveness patterns allowed englobing two levels. The first is related to the class structure modifications like the addition of a field. The second is related with methods and fields declared in the class. A class is specified with an allowed invasiveness patterns that applies to its fields and methods. This specification can allow any invasiveness pattern forbidden by default, and forbid any invasiveness pattern allowed by default. This specification corresponds to (a) in figure 3 covering all the class definition.

Specification for fields: Specify the invasiveness patterns that a field allows in terms of how advices access it. This specification can allow *Write* and forbid *Read*. This specification corresponds to (b) in figure 3 covering only a field definition.

Specification for methods: Specify the invasiveness patterns allowed on a specific method. It can allow *Replacement*, *Conditional replacement*, *Multiple*, *Write*, *Argument passing*. It can forbid *Augmentation*, *Crossing*, *Read*. This specification corresponds to (c) in figure 3 covering only a method definition.

It is worth mentioning that a class specification can be reused in the case of the inheritance. For example, The class `B` extends class `A`. If `C` is the specification of `A`, and `B` is not explicitly specified, then `C` is the specification of `B`.

In the case of conflicts between the specifications of fields, methods and classes we propose the following: The specification of fields is always used if it exists. If a method is specified (explicitly), its specification is used instead of the global.

3.3. Specification matching

We compare the core and aspect specifications in order to detect when aspects or advices violate the core specifications. An aspect or an advice violates a core specification when it realizes invasiveness patterns that the core specification forbid. This gives us information about the invasive aspects harmfulness. Such information is used to assist the developer to reason about the impact of aspects on the composed program.

We detect violations of specification in the following way: At the aspect level, for each aspect we obtain the classes it targets adding fields, methods or modifying the hierarchy. Then, we compare the specification of forbidden patterns on each class with the specification of the aspect. At the advice level, for each advice we obtain the methods it advises. Then, we compare the specification of forbidden patterns on each method with the specification of the advice. For the advices accessing fields, the matching is analogous to the previous.

4. A specification framework for interactions

We have implemented a tool for matching specification as well a language to express the base program specifications called ABIS (Aspect-Base Interaction Specification)[1]ABIS is built on top of the AJDT eclipse plug-in and is completely integrated with eclipse. After a short presentation of the global structure of ABIS we detail how each aspect and advice is automatically classified. Then we describe how to specify the base program with the expected invasiveness patterns.

Figure 4 presents the ABIS' organization. We extend the AspectJ `AST Visitor` (1) in order to create a simplified (SAST) version of the Abstract Syntax Tree (AST). ABIS obtains information about the structure of the program (aspects and base program) from AJDT and builds a model of the program structure (2). This model contains the relations between aspects and the base program (advised and introduced element relations). An automatic classification process inspects the SAST and classifies each aspect and advice according to its invasiveness pattern (3). Then the model and the classified advices are checked following the previously presented matching process (4). If specification violations are found, then they are reported to the eclipse GUI (5).

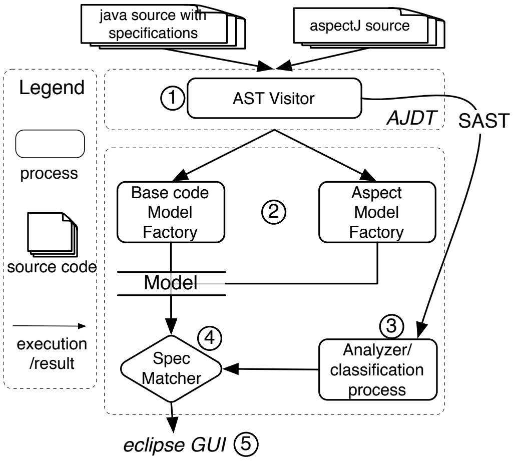


Figure 4. ABIS structure diagram

4.1. Automatic classification of aspects

ABIS is able to automatically identify invasiveness patterns in aspects and advices according to their de-facto properties. Then, by using the identified patterns, aspects and advices are classified according to the classification presented in section 3.1.

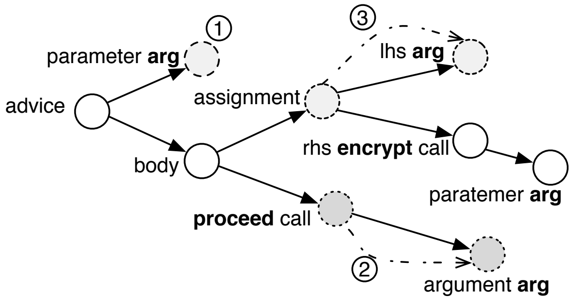


Figure 5. SAST of *Encryption* aspect, Listing 1, lines 9-12

Figure 5, shows the SAST of the first advice of the *Encryption* aspect (Listing 1, lines 9-12). The root node of the SAST corresponds to the advice declaration. From the root node, we find the parameter declaration and the advice body node. The children of the body node are the statements declared on the advice. The node `assignment` corresponds to the assignment in line 10 of Listing 1. The node `proceed call` represents the call to the `proceed(arg)` method (line 11 of Listing 1) (it executes the intercepted method) and its children are the arguments it receives.

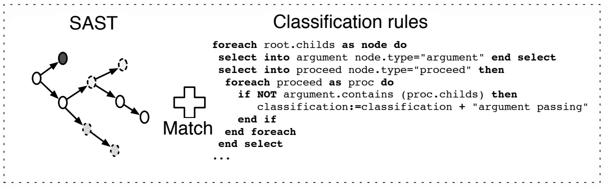
The classification algorithm applied to the *Encryption* aspect is the following:

1. Initially, select all the advice argument nodes (step 1, `argument arg`).

[1] Available at `http://contract4aj.gforge.inria.fr`

2. Traverse the SAST searching for the nodes representing a call to the proceed method (proceed node).
   (a) If all the arguments nodes reference the advice parameter nodes, then select it (step 2, `proceed(arg)`).
   (b) If the proceed arguments are different from the advice parameters, then classify the advice as *Parameter passing*.
3. Starting from the last selected proceed node, select the assignment nodes on top of it. If the left hand side (lhs) of an assignment references one of the advice arguments (step 3, `arg=Encrypt(arg)`), then classify the advice as *Parameter passing*.

This algorithm represents the set of rules used to identify the *Parameter passing* invasiveness pattern. In general, invasiveness patterns are identified by applying a set of rule to the advice SAST.



```
foreach root.childs as node do
 select into argument node.type="argument" end select
 select into proceed node.type="proceed" then
  foreach proceed as proc do
    if NOT argument.contains (proc.childs) then
       classification:=classification + "argument passing"
    end if
  end foreach
 end select
...
```

Figure 6. Automatic advice classification

Figure 6 depicts the automatic classification process or advices. The process proceeds as follows: once obtained the advices SASTs a set of identification rules are applied to them. The results of this inquiry are the invasiveness patterns that the inspected SAST realizes. For example, an identification rule can check whether the left hand side of an assignment node is a reference to an object field, and then detected the invasiveness pattern *Write*.

Aspects invasive patterns are detected according to the aspect structural declarations. For example, if an aspect declares Inter-Type field, then the detected invasive pattern is *Field Addition*.

4.2. Writing specification in the base program

The base program specification is represented as meta-information by using Java 5 annotations [2]. The parameterized annotation `@Spec([allow=..|forbid=..])` specifies the expected invasiveness pattern in the base program. It can be attached to classes, fields and methods, and the possible values for its parameters vary as described in section 3.2.

The parameters `allow` and `forbid` are exclusive, i.e. only one can be used in each specification. The `allow` parameter indicates the invasiveness patterns allowed to interact with the base program. The default policy is to allow the non-invasiveness patterns *Augmentation*, *Crossing* and *Read*. Analogously the `forbid` parameter indicates the invasiveness patterns forbidden to interact with the base program.

```
1  @Spec(allow={"ConditionalReplacement"})
2  public synchronized int attach(Client client)
3        throws RemoteException {
4     ...
5  }
```

Listing 3. `attach` method specified with an annotation

Listing 3 shows the `attach` method of the standard server specified with an annotation (line 2). Finally, this method allows advices realizing the following patterns to advise it: *Augmentation*, *Crossing*, *Conditional replacement* and *Read*. All the other patterns are forbidden.

4.3. Contribution of the ABIS framework

ABIS statically computes and gives information, at compile-time, about the specification violation. This information is a useful and valuable:

1. Feedback for developers in the process of writing advices a specifying the base program.
2. For verifying an aspect-oriented program when aspects and the base program are developed separately.
3. For verifying an aspect-oriented program when aspects or the base program evolve.

The compile-time feature of the tool is also a drawback. The current implementation is unable to detect and check dynamic join points (for example using the `if` keyword in AspectJ).

5. Case study revisited

In this section we revisit the case study presented in section 2 by specifying the interaction between aspects and base program. Then, we use these specifications to spot and solve the problems that arise after the base program evolution.

5.1. Specifying the initial version

We annotate the initial version of the chat application specifying the allowed invasiveness pattern. The annotations we added allow the invasiveness patterns that interact safely with the base program. Therefore, only the advices realizing the specified patterns can advise the base program.

```
public class StandardClientImpl implements IClientIM{
  @Spec(allow={"ArgumentPassing"})
  public synchronized void transmit(Message message) ...
  @Spec(allow={"ArgumentPassing"})
  public void send(String sContents){ ... }
  @Spec(allow={"ArgumentPassing"})
  private void retransmit(Message message){ ... }
  ...
}
```

Listing 4. `StandarClient` **with annotations**

Listing 3 and 4 present the fragment of the specified methods. The methods `send`, `transmit` and `retransmit` were specified to allow the invasiveness pattern *Argument passing*. Aspect were automatically specified by ABIS. Thanks to these specifications, ABIS informs that no aspect is violating the base program specifications. Furthermore, the addition of annotations is transparent for the tests we execute, hence, the test results are not affected.

5.2. Evolution with specification, problems detection and solving

After specifying the initial version of the chat application, we evolve it as explained in section 2.5. As a result of this evolution, ABIS reports that some aspects are violating the base program specifications. The reports indicate that the aspects `EncryptionAspect` and `UniqueName` are violating the specifications on `AuthenticatedClient` (`attach` method) and `AuthenticatedServer` (`send` and `transmit` methods) respectively. This means that potentially unexpected behavior can emerge from the weavage of these invasive aspects advising new join points.

Using this information we trace unexpected interactions to the violator advices (problems presented in section 2.7). Knowing the violating pattern helps us reasoning about the causes of the unexpected interaction, hence, reasoning about the source of the problem. For example, the violating pattern *Conditional Replacement* tells us that something is wrong because it is possible that a method, which must always execute, sometimes will not be executed.

The further correction of the problems is a developer decision, however, the violated specification may help making the solution. In this case, we have applied the same procedure used in section 2.7, i.e. change the point-cut descriptors. Once applied the corrections, ABIS reports that no specification is violated and aspects are valid in relation with the base program specifications.

5.3. Discussion

Specifying interactions gives feedback about the harmfulness of aspects. This information assists developers in the process of creating an aspect-oriented program and ensuring that aspects perform as expected. It also helps developers to be conscious about the aspects they write and enforces the reasoning about the interaction between aspects and base program.

The violation of an specification indicates that a developer may review the code of the violator aspects and reason about their impact on the base program. Besides, specifying interactions reduce the effort and time required to locate faulty aspects because there is no need to successively execute a set of tests. We thinks that all these reasons justify the effort and time involved in specifying (annotating) the base program.

The Open Closed Principle [9] describes that a module/-class should be open for extension but closed for modification. Of all the principles of object oriented design, this is the most important. It means that we should write modules so that they can be extended without requiring them to be modified. In other words, we must be able to change the modules behavior without changing their code. Our framework is completely coherent with this principle. Aspects offer a new way to extend the behavior of classes without modifying their code (classes are opened). Therefore, developers have to anticipate these potential extensions and specify which kind of behavior modification (invasiveness pattern) is allowed.

6. Related work

The characterization of aspects has already been explored. In [14] 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. Moreover, in our work the identification of invasive patterns is only a portion of a whole specification framework. In [7] aspects are characterized among *Spectative*, *Regulatory* and *Invasive* aspects according to their invasiveness. This classification is similar to ours, however, our characterization of is more fine grained.

Several approach have been proposed to control the interactions between aspects and the base program. *Spectators* and *Assistants* [4] proposes to control interactions by specifying the invasive aspects that can advise the base program. It classifies aspect among *Spectators* (non invasive advices) and *Assistants* (invasive advices). Then, the base program explicitly demands the assistance of assistant aspects (identified by their names). We propose the same type of specifications, but we address a finest granularity and abstraction by referring to invasiveness patterns instead of specific invasive aspects.

Open Modules [1] is a system that focuses on the exposure of specific join-points. This approach hides all the join-points, then each module declares the join-point it will expose. *Open Modules* goes in a similar direction than our work. However, we perform this task in a very different way. *Open Modules* exposes join-points without distinguishing the aspects advising them, whereas we expose joint-point to aspects realizing specific invasiveness patterns. Moreover, our approach is intended to verify and validate aspect-oriented programs and assist developers when creating aspects.

XPI [5] are interfaces that mediate between aspects and the base program. They establish a set of design rules to implement aspects and the base program in such a way that the evolution is coordinated through the XPI. This approach restraint the manners in which developers may write programs. Instead, to check whether advices may be harmful to the base program, then advising developers to check the potential unexpected interaction introduced by invasive aspects.

An approach to assist developers is proposed in [15]. Through an analysis that compares the changes in the set of matched join-points for two different version of a program it reveals unexpected changes in the matching behavior of point-cuts. This analysis serves to assist developers finding bugs introduced by broken point-cuts. This work is close to ours. However, the dimensions in which this is accomplished are very different. Our approach advises developer about the potential undesired interactions introduced by invasive aspects instead of broken point-cuts. Moreover, our approach require the specification of the base program, which we think enforces the reasoning about the interactions with aspects.

7. Conclusions

The evolution of aspect-oriented programs is a complicated issue because necessary interactions of one version can introduce issues after evolution. This occurs as a consequence of the *AOSD-Evolution paradox* and invasive aspects. Through the evolution of an aspect-oriented chat application we have shown that tracing problems to unexpected interactions is a long and tedious process. Such a process involves rigorous manual inspection of code and the execution of several test scenarios.

This paper tackles this problem by specifying the interactions between aspect and the base program. Aspects are specified with the invasiveness patterns they realize, and the base program with assertions allowing or forbidding invasiveness patterns. The violations of these specifications are used to alert developers about the risk introduced by unexpected interactions. This assists developers reviewing the harmful code and to reason about its interaction with the base program. By specifying and evolving the chat application we have shown that specifying interactions reduces the time and effort necessary to locate problems introduced by unexpected interactions.

The specification of aspect-oriented programs improves their maintenance and evolvability. It also increases the confidence that developers have on aspect. This because developers can ensure that critical parts of the base program will not be modified unexpectedly by future addition of aspects.

8. Acknowledgments

This work was partially supported by the european project *DiVA* (EU FP7 STREP).

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In A. P. Black, editor, *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer, 2005.

[2] J. Bloch. A metadata facility for the java programming language. Technical report jsr 175, Sun Microsystems, www.jcp.org, 2002.

[3] C. Bockisch, M. Arnold, T. Dinkelaker, and M. 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.

[4] C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In R. Cytron and G. T. Leavens, editors, *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, pages 33–44, Mar. 2002.

[5] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, pages 51–60, Jan./Feb. 2006.

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

[7] S. Katz. Diagnosis of harmful aspects using regression verification. In C. Clifton, R. Lämmel, and G. T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages*, pages 1–6, Mar. 2004.

[8] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[9] B. Meyer. *Object-Oriented Software Construction*. Number ISBN 0136290493. Prentice Hall, first edition, 1988.

[10] M. Mezini and K. 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.

[11] F. Munoz, O. Barais, and B. Baudry. Vigilant usage of aspects. *Workshop on Aspects, Dependencies and Interactions. ECOOP 2007, Berlin, Germany*, July 2007.

[12] F. Munoz, B. Baudry, and O. Barais. A classification of invasive patterns in aop. Research report inria-00266555, IRISA Research Center, http://hal.inria.fr/inria-00266555/en/, March 2008.

[13] R. 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.

[14] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for interactions in aspect-oriented programs. In *Foundations of Software Engineering (FSE)*, pages 147–158. ACM, Oct. 2004.

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

[16] Sun Microsystems. *Java Remote Method Invocation Specification*, Nov. 1996.

[17] T. Tourwé, J. Brichau, and K. Gybels. On the existence of the AOSD-evolution paradox. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, Mar. 2003.