JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 7, No. 6, July–August 2008

Overcoming comprehension barriers in the AspectJ programming language

Venera Arnaoudova
Laleh Mousavi Eshkevari
Elaheh Safari Sharifabadi
Constantinos Constantinides
Department of Computer Science and Software Engineering,
Concordia University,
1455, De Maisonneuve Blvd. West,
Montréal, Québec, H3G 1M8, Canada

It has now been over a decade since the introduction of Aspect-Oriented Programming (AOP). As the AspectJ programming language (being one of the notable technologies of AOP) gains acceptance in industry and academia, its comprehensibility property is an important factor in determining an eventual wide acceptance by practitioners in development and maintenance as well as by educators who aim at introducing AOP into their curricula. Our objective is to improve program comprehension by identifying and addressing potential pitfalls in code which tend to make comprehension not intuitive. In those subtle places, we observe the behavior of the program to see the degree to which it matches the expected results. In cases where a conflict occurs, we provide a reasoning to point out where it would originate from, and a resolution to the conflict where applicable.

1 INTRODUCTION

Separation of concerns and its associated benefits such as good code modularity tend to be one of the objectives of many programming paradigms and languages. One such paradigm is Aspect-Oriented Programming (AOP) which was introduced to the community over a decade ago [Kiczales et al., 1997]. Aspect-Oriented Programming is currently supported by a number of technologies, perhaps the most notable of which is AspectJ [Kiczales et al., 2001], an aspect-oriented extension to the Java language. With a significant collection of supporting tools and an increasing community of practitioners and developers, the AspectJ language has influenced the design dimensions of several other general-purpose aspect-oriented languages, and has provided the community with a de facto common vocabulary based on its own linguistic constructs.

Over the last decade we have also experienced the worldwide adoption of AOP approaches in institutes of higher education[1]. In our institution we have included AspectJ in our upper-level undergraduate and graduate curriculum over the last four academic years. Relevant courses include the assignment of small to medium-scale projects involving the adoption of AspectJ for program development as well as for program maintenance. For the latter, one common

[1]A list of academic and industrial institutions is maintained at the following website (last accessed: June 5, 2008): http://dev.eclipse.org/viewcvs/indextech.cgi/aspectj-home/teaching.html.

task includes program comprehension. Another task includes reengineering of object-oriented (Java) programs into an aspect-oriented (AspectJ) context.

Of particular interest to us is comprehension, especially during maintenance where it constitutes the initial and vital step for any task and tends to consume a significant proportion of time. While teaching AspectJ we have realized that in some cases understanding program semantics may not be obvious for students. The purpose of this article is, therefore, to understand why this difficulty exists, to contribute to the comprehension of AspectJ programs based on our empirical work by exploring different cases[2] and to encourage people to adopt AspectJ by showing that some of the unintuitive[3] behavior originates in the underlying language (Java) and the answers to some questions which seem to be AspectJ-related can be found in the Java language specification.

The rest of this article is organized as follows: In Section 2 we provide some necessary background and in Section 3 we discuss the problem and motivation behind this research. In Section 4 we describe the subject population and the settings for our experiments. In Section 5 we discuss a number of cases, where our investigation has shown that comprehension of AspectJ programs tends to become difficult. For each case we initially describe the intent, followed by an intuitive solution based on a survey we have conducted with a group of students. Next, we implement and observe the behavior of the programs; in cases where the behavior differs from the expectations, we provide a reasoning why this is so and discuss guidelines on how one can achieve the intended behavior. In Section 6 we discuss related work, and we conclude our discussion in Section 7.

2 BACKGROUND

The principle of separation of concerns [Parnas, 1972, Dijkstra, 1976] refers to the realization of system concepts into separate software units and it is a fundamental principle in software development. The associated benefits include better analysis and understanding of systems, high readability of modular code, high level of reuse, easy adaptability and good maintainability. Despite the success of object-orientation in the effort to achieve separation of concerns, certain properties cannot be directly mapped in a one-to-one fashion from the problem domain to the solution space, and thus cannot be localized in single modular units. Their implementation ends up cutting across the inheritance hierarchy of the system. Crosscutting concerns (or "aspects") include persistence, authentication, synchronization, contract checking and logging. The "crosscutting phenomenon" creates two implications: 1) The scattering of a concern over a number of modular units and 2) The tangling of code of several concerns in one unit. As a result, developers are faced with a number of problems including a low level of cohesion of modular units, strong coupling between them and difficult module comprehensibility, resulting in programs that are more error prone.

[2]All examples discussed are compiled with the ajc compiler (version 1.5.2) and the abc compiler (version 1.2.1).

[3]The definition of unintuitive here is based on a survey over 35 upper-level undergraduate and graduate students with experience in AOP, enrolled in Computer Science and Software Engineering programs at Concordia University.

Aspect-Oriented Programming [Kiczales et al., 1997, Elrad et al., 2001] explicitly addresses those concerns by introducing the notion of an aspect definition, which is a modular unit of decomposition. Currently there exist many approaches and technologies to support AOP. One notable technology is AspectJ [Kiczales et al., 2001], a general-purpose aspect-oriented language, which has influenced the design dimensions of several other aspect-oriented languages, and has provided the community with a common vocabulary based on its own linguistic constructs. There are currently two AspectJ compilers, namely ajc[4] and abc[5]. In the AspectJ model, an aspect is a new unit of modularity providing behavior to be inserted over functional components. This behavior is defined in method-like blocks called *advice* blocks. However, unlike a method, an advice block is never explicitly called. Instead, it is only implicitly invoked by an associated construct called a *pointcut* expression. A pointcut expression is a predicate over well-defined points in the execution of the program which are referred to as *join points*. When the program execution reaches a join point captured by a pointcut expression, the associated advice block is executed. Even though the specification and level of granularity of the join point model differ from one language to another, common join points in current language specifications include calls to, and execution of methods and constructors. Most aspect-oriented languages provide a level of granularity which specifies exactly when an advice block should be executed, such as executing before, after, or instead of the code defined at the associated join point. Furthermore, several advice blocks may apply to the same join point in which case advice precedence rules [Kienzle et al., 2003, Lorenz and Kojarski, 2006] are applied. In cases where these advice blocks are defined in the same aspect, precedence of advice execution depends on the type of advice. For `before` advice blocks, the one defined first has precedence over the one following it, that is, it will be executed first. For `after` advice blocks, the one defined last has precedence. For `after` advice having highest precedence means executed last. In cases where these advice blocks are defined in different aspects, precedence can be defined through the `declare precedence` construct. An aspect definition may also define state and behavior to be introduced into the core functionality. It may also declare a new parent type for an existing set of types.

3 PROBLEM AND MOTIVATION

In this article we explore some cases, in order to find an explanation why the actual behavior is as it is, and to find one possible solution for reaching the intended goal (where applicable). Our motivation is to decrease the gap between the semantics of the language and the understanding of program readers.

When implementing a solution of a problem in a specific programming language, we rely on its semantics in order to predict how the program will behave when it is executed. While teaching AspectJ in our institution we observed that many students have difficulties in providing the result of some AspectJ programs. We then decided to survey graduate and upper-level undergraduate students in order to understand the critical points in comprehension of AspectJ semantics. We prepared questionnaires containing programs under different cases and we asked the participants to provide answers to each program. In Section 4 we describe how the survey

[4]Available at http://www.eclipse.org/aspectj/

[5]Available at http://abc.comlab.ox.ac.uk

Category	Language	Professional	Good	Familiar	Number of participants
Category 1	Java AspectJ	Y Y			4
Category 2	Java AspectJ	Y	 Y		12
Category 3	Java AspectJ		Y	 Y	19

Table 1: Categories of participants based on their knowledge of Java and AspectJ.

was conducted and in Section 5 we discuss those cases where students had difficulties with.

4 EXPERIMENTS

In this section we describe the subject population for these experiments. The group of participants consists of undergraduate and graduate students some of whom are employed as industrial developers[6]. The participants are categorized in three groups as shown in Table 1. In the subsequent paragraphs we provide a more detailed description of each group.

Participants in Category 1: They are industrial Java programmers and academic AspectJ programmers. They are currently enrolled in a graduate (master/ PhD) program in Computer Science/ Software Engineering, engaged in AOP-related research. Four participants fell into this category.

Participants in Category 2: They are industrial Java programmers. They have good knowledge of AspectJ, since they have completed at least one course covering the principles of separation of concerns and AOP. They are currently enrolled in an undergraduate program in Computer Science (3-year program)/ Software Engineering (4-year program). Twelve participants fell into this category.

Participants in Category 3: These students do not have any industrial experience with Java even though they have enough background in object-oriented programing (90% of which is Java) in academia. They were not familiar with AOP/AspectJ but they were introduced to the principles of separation of concerns, AOP, and AspectJ during a one-term course (13 week; 2.5 hours per week of lectures and 50 minutes per week of tutorial). We can therefore consider their level as “being familiar with AspectJ.” Nineteen participants fell into this category.

The participants were asked to complete a printed questionnaire containing 19 questions by providing the output of the given programs. They were given “AspectJ Language Quick Reference” from [Colyer et al., 2004] and they were free to use any other documentation. The

[6]In [Penta et al., 2007] the authors discuss the identification of an appropriate subject population as still being an open issue.

constraints were as follows: First, the questionnaire should be completed individually and second, it should be completed as a dry run, i.e. with no assistance from a compiler. There was no time limitations for responding to the questionnaire.

5 CASES: INTENT AND BEHAVIOR

In this section we describe a number of cases where the provisions of the AspectJ language do not seem to be intuitive. The cases are classified into two categories, Java and AspectJ, based on the reason this unexpected behavior originates from.

Java

Since Java is the underlying language of AspectJ, we should not be surprised from the fact that it inherits design/implementation decisions from Java. Thus, for some cases, one should look for an explanation in the language specification of Java rather than AspectJ. Following are such examples.

Exact location of "before execution" of constructors

In this example we want to capture the exact location of the `before` constructor execution join point. We tend to think that before execution of a constructor is like before execution of a method, meaning before the body is explored. Thus, even if the first statement of a constructor is a call to another constructor, the first constructor to be called should start execution before any referenced constructors. Consider the following code:

```
public class C {
  ...
  C(int x){...}
  C(int x, String y){
    this(x);
    this.y = y;}}

public class Demo {
  public static void main(String[] args){
    C c = new C(5, "s");}}

public aspect Tracer {
  pointcut constructor():
    execution(C.new(..));
  before():constructor(){...}
  after():constructor(){...}}
```

Based on the above explanation one may expect the following result[7].

[7]Advice blocks in all cases display the signature of `thisJoinPoint`.

```
Before execution(C(int, String))
Before execution(C(int))
After execution(C(int))
After execution(C(int, String))
```

However, this is not the actual output of the program. In Java, the compiler treats methods and constructors differently. The execution of the body of a constructor starts after completing the following steps [Gosling et al., 2005]:

1. Evaluation of all arguments.
2. Invocation of another constructor in the same class (by going through steps 1 to 4, followed by the execution of its body), should an explicit call be made through the keyword `this`.
3. Invocation of a constructor in the superclass (by going through steps 1 to 4, followed by the execution of its body), should an implicit or explicit call be made to the superclass constructor through the keyword `super`.
4. Execution of the instance initializers and instance variable initializers for this class.

Consequently, AspectJ treats methods and constructors in the same manner as Java. Execution of a method is considered to be *"when the body of code for an actual method executes"*, whereas execution of a constructor is *"when the body of code for an actual constructor executes, after its* `this` *or* `super` *constructor calls"* [The AspectJ Team, 2006]. Thus the output of the above example (with both abc and ajc compilers) is

```
Before execution(C(int))
After execution(C(int))
Before execution(C(int, String))
After execution(C(int, String))
```

The above result can be interpreted as follows: the execution of the constructor being called starts after the completion of the execution of its inner constructor (see Figure 1). Two of the participants in Category 1 provided the actual output, while none of the participants in categories 2 or 3 did.

Counting the creation of objects

In this case our intention is to capture the number of created instances of a particular class. The question is "What exactly should be monitored?" and it may lead to more than one possibilities: monitor the number of calls to constructors; monitor the number of executions of constructors or, alternatively, count the number of classes that have been initialized. For this experiment participants were given all possible alternatives and they were asked to provide the output of each alternative. If the provided result corresponds to the actual number of objects created, we

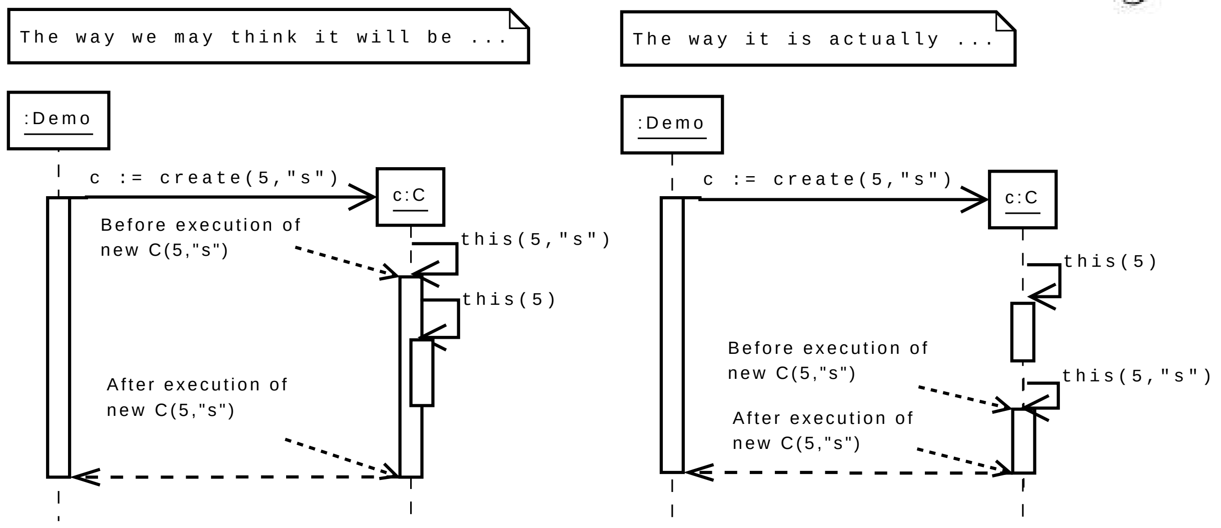
The way we may think it will be ...
:Demo
c := create(5,"s")
c:C
Before execution of new C(5,"s")
this(5,"s")
this(5)
After execution of new C(5,"s")
The way it is actually ...
:Demo
c := create(5,"s")
c:C
this(5)
Before execution of new C(5,"s")
this(5,"s")
After execution of new C(5,"s")

Figure 1: Sequence diagram - exact location of "before execution" of constructors.

suppose that the participants consider this alternative as a good possible solution. At the end, participants were asked to choose the best solution for the purpose of this case.

Consider the following definitions:

```
public class C1 {}

public class C2 extends C1{
  int x;
  String y;
  C2(int x){
    super();
    this.x = x;}
  C2(int x, String y){
    this(x);
    this.y = y;}}

public class Demo {
  public static void main(String[] args){
    C2 c = new C2(5, "s");}}
```

Thirty percent of the participants think that using `execution` pointcut on the `new` keyword is a good idea since we count the number of objects created. This results in the following aspect definition:

```
public aspect A {
  before():
    execution(*.new(..)) && !within(A){...}}
```

The above pointcut monitors the execution of any constructor outside aspect `A`, and results in three captured join points (with both abc and ajc compilers) because three constructors are executed (`C2(int,String)`, `C2(int)`, `C1()`).

One may argue that in order to be able to count the number of objects created with `execution`, the pointcut definition should be changed to:

```
before():
  execution(*.new(..)) && !within(A)
  && !cflowbelow(execution(*.new(..))){...}
```

The above advice block means that we exclude the execution of constructors that are already in the control flow of other constructors. When we asked the participants whether this would solve the problem fifty percent replied "yes." By running the program however, we still observe three captured join points (with both abc and ajc compilers). The reason for this behavior is that as explained in Section 5, the execution of a constructor starts after the execution of the nested constructors called with the keywords `this` and `super`. This implies that the inner most constructor will be executed first, and after its execution the constructor that called it, will start executing and so on. After returning from all nested constructors, the body of the initial constructor (in this case the one called in the `main` method in class `Demo`) will start executing. Thus the pointcut designator

```
!cflowbelow(execution(*.new(..)))
```

does not affect the captured join points, since there is no execution of a constructor inside the execution of another constructor. For example, in Figure 2 the constructor `C2(int,String)` is called in the `main` method of class `Demo`. As the first statement of this constructor is `this(5)`, the constructor `C2(int)` will be executed first. Since the first statement of `C2(int)` is `super()`, the constructor `C1()` will be executed first. After its execution, the rest of the body of constructor `C2(int)` will be executed, followed by the execution of the rest of the body of constructor `C2(int,String)`. Thus, the control flow below the execution of `C2(int,String)` does not contain any other constructor execution. This implies that keeping or deleting the previous pointcut designator does not affect the set of captured join points.

We then proposed an alternative solution, chosen by more than fifty percent of the participants, which is to use the `withincode` pointcut designator as follows:

```
execution(*.new(..)) && !within(A)
&& !withincode(*.new(..)){...}
```

The above pointcut captures every join point from the code defined outside the code of any constructor of any class (and not in aspect `A`). However adding `withincode` to an `execution` pointcut does not change the set of captured join point (with either abc or ajc compilers), because for `execution` pointcut, the enclosing code of the method is the method itself. Thus, adding

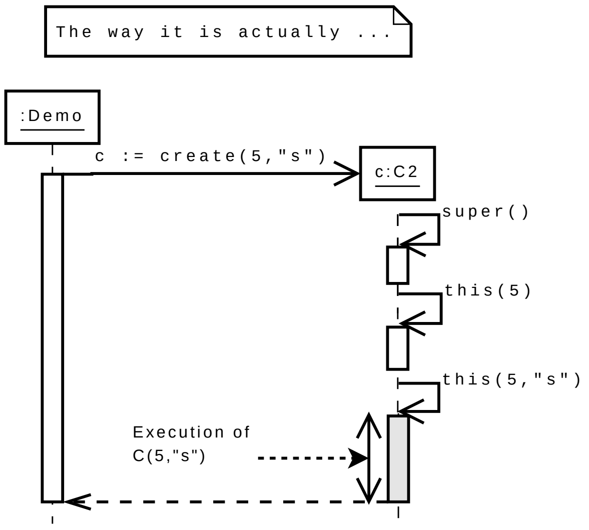

Figure 2: Sequence diagram - counting number of objects created.

```
!withincode(*.new(..))
```

to the join point designator `execution(*.new(..))` is interpreted as "`execution` of a method and not inside its body", or $p \wedge \neg p$ which is always false for any value of p. This implies no captured join points. Thus, capturing the execution of constructors is not a suitable solution for this purpose. More than fifty percent of the participants believed in another candidate solution to this question: the use of the `initialization` pointcut. This solution is invalid due to the underlying semantics of AspectJ. In Java, when an object is created its direct superclass is initialized first [Gosling et al., 2005]. The `initialization(*.new(..))` pointcut matches two join points and produces the following output (with both abc and ajc compilers):

```
initialization(C1())
initialization(C2(int, String))
```

Another solution may be the following:

```
public aspect A {
  before():
    call(*.new(..)) && !within(A){...}}
```

The above pointcut monitors any call to any constructor (outside the definition of aspect `A`), thus one may expect to have three captured join points:

1. In the main method of class Demo (because of the statement `C2 c = new C2(5, "s");`).
2. In the constructor `C2(int x, String y)` in class `C2` because we explicitly call another constructor of the class in it (`this(x);`).

3. In the constructor `C2(int x)` because we explicitly call the constructor of the superclass (`super();`).

In reality, we have only one captured join point (with both abc and ajc compilers). The captured join point corresponds to the call of the constructor in the `main` method of class `Demo`. Even if we explicitly call another constructor within the called constructor using `this` and `super`, these join points are not captured. We can then conclude that even if all three of them represent a call to a constructor, there is a difference between capturing the creation of an object with the keyword `this` or `super` on one hand, and with the keyword `new` on the other hand, when the pointcut is based on `call`.

Finally, we can conclude that in order to count the number of objects created, one should monitor the calls to class constructors, using `new` keyword, i.e.,

```
call(*.new(..)) && !within(A)
```

Unfortunately when the participants were asked to chose the most suitable solution for counting the number of objects created only one sixth of them chose the above pointcut.

AspectJ

For some other examples, one should search an explanation for the behavior of AspectJ programs in the design/implementation decisions behind the AspectJ language. In the following paragraphs we discuss some of these cases.

Library methods

Consider the following code:

```
public class Demo {
  public static void main(String[] args) {
    Vector v = new Vector();
    v.size();}}

public aspect A {
  pointcut callVectorSize():
    call(public int java.util.Vector.size());
  pointcut execVectorSize():
    execution(public int java.util.Vector.size());
  before():callVectorSize(){...}
  after():callVectorSize(){...}
  before():execVectorSize(){...}
  after():execVectorSize(){...}}
```

As we define four advice blocks, two third of the participants expected that call and execution of method `size()` of class `Vector` will be captured, and the following output will observed:

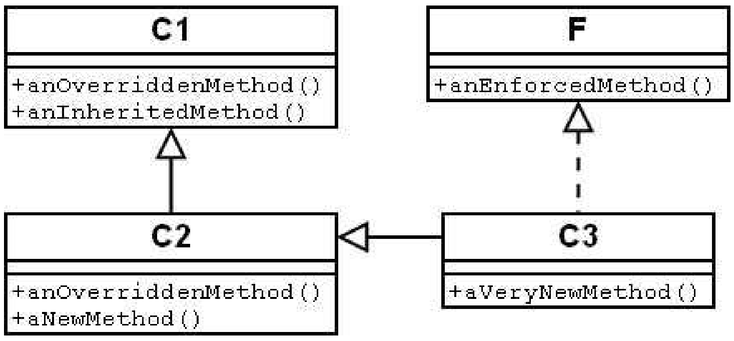

Figure 3: Class diagram - tracing methods.

```
callVectorSize before call advice
execVectorSize before execution advice
execVectorSize after execution advice
callVectorSize after call advice
```

The actual result, however (with both abc and ajc compilers) is:

```
callVectorSize before call advice
callVectorSize after call advice
```

One main difference between `call` and `execution` pointcuts is that `call` refers to the caller side (class `Demo` in this case) whereas `execution` refers to callee side (class `Vector` in this case) [Masuhara et al., 2002]. The question is then "Why execution of library methods is not captured?".

In order to explain the difference between expected and observed behavior, we need to consider how advice weaving is done based on static shadows.
Static shadow is a specific place in the source code or bytecode which corresponds to possible joint points. For each static shadow the back-end AspectJ compiler checks if a precompiled advice can match this shadow. Should this be the case, a call to the specific advice is injected into the bytecode [Hilsdale and Hugunin, 2004]. But if the weaver is not able to modify the bytecode, then the call to an advice can not be inserted. Based on *AspectJ Language Guide*, execution join point can be advised if the compiler controls the bytecode for the method or constructor body in question. Thus, in the previous example, because the weaver does not have control over the bytecode of the library methods, the execution of `java.util.Vector.size()` is not captured by the aspect. However, the call to `java.util.Vector.size()` is captured because a call to the advice is injected in class `Demo`.

Tracing methods

Our intent in this case is to capture any call and execution of any method (with no parameters) of a given class. Consider the class diagram in Figure 3.

We have defined pointcut `P_EXE_C1()` as `execution(* C1.*())`, and pointcut `P_CALL_C1()` as `call(* C1.*())`. In the same manner we defined six more pointcuts for the other classes and interface as follows: `P_EXE_C2()`, `P_CALL_C2()`, `P_EXE_C3()`, `P_CALL_C3()`, `P_EXE_F()`, `P_CALL_F()`. For each pointcut we have defined a `before` advice block. Consider class `Demo` defined as follows:

```
public class Demo {
  public static void main(String[] args) {
    C3 o3 = new C3();
    o3.anOverriddenMethod();}}
```

Based on the fact that only method `anOverriddenMethod()` is called on object `o3`, more than fifty percent of the participants think that only `P_EXE_C2()` and `P_CALL_C3()` pointcuts will capture the join point:

- `P_CALL_C3()` because the call is made to `o3` which is of type `C3`.
- `P_EXE_C2()` because `C2` is the first parent of `C3` which contains the method definition.

This results in the following output:

```
P_CALL_C3:
  call(void C3.anOverriddenMethod())
P_EXE_C2:
  execution(void C2.anOverriddenMethod())
```

However, after running the program we obtain the following (with both abc and ajc compilers):

```
P_CALL_C1:
  call(void C3.anOverriddenMethod())
P_CALL_C2:
  call(void C3.anOverriddenMethod())
P_CALL_C3:
  call(void C3.anOverriddenMethod())
P_EXE_C1:
  execution(void C2.anOverriddenMethod())
P_EXE_C2:
  execution(void C2.anOverriddenMethod())
```

This result is justified by the fact that for both `call` and `execution` pointcuts two factors must be taken into consideration: type and method declaration. However, there is a subtle difference between `call` and `execution` with regards to those characteristics:

- For a `call` pointcut to match a join point two criteria are important:
  - the **static type** of the object involved in the join point should be of type (or subtype) of the one involved in the pointcut,

  - signature of the method involved in the join point should exist in the **class (or superclass)** of the one monitored by the pointcut.

- For an `execution` pointcut to match a join point two criteria are important:
  - the **dynamic type** of the object involved in the join point should be of type (or subtype) of the one involved in the pointcut,
  - signature of the method involved in the join point should exist in the **class** monitored by the pointcut. Note that here superclasses are not checked.

Barzilay *et al.* discussed previously the patterns for matching `call` and `execution` pointcuts [Barzilay et al., 2004] but with an early version of AspectJ and are not applicable any more with the current version. The authors showed that for matching a `call` pointcut the method should be defined in the specific class (inheritance is not a sufficient condition). In the above guidelines however we explain that inheritance is now enough. Also, for matching an `execution` pointcut the authors explained that the method should be defined or overridden, whereas we show that currently a signature is enough.

Applying our guidelines on the above example, we observe the following:

1. `P_CALL_C1(): call(* C1.*())` pointcut captures `o3.anOverriddenMethod()` join point because
   - static type of `o3` is `C3`, which “is-a” `C1` and
   - the signature of `anOverriddenMethod()` exists in class `C1`.
2. `P_CALL_C2(): call(* C2.*())` pointcut captures `o3.anOverriddenMethod()` join point because
   - static type of `o3` is `C3`, which “is-a” `C2` and
   - the signature of `anOverriddenMethod()` exists in class `C2`.
3. `P_CALL_C3(): call(* C3.*())` pointcut captures `o3.anOverriddenMethod()` join point because
   - static type of `o3` is `C3`, and
   - the signature of `anOverriddenMethod()` exists in a superclass of `C3`, which is `C2`.
4. `P_EXE_C1(): execution(* C1.*())` pointcut captures `o3.anOverriddenMethod()` join point because
   - dynamic type of `o3` is `C3`, which “is-a” `C1` and
   - the signature of `anOverriddenMethod()` exists in `C1`.
5. `P_EXE_C2(): execution(* C2.*())` pointcut captures `o3.anOverriddenMethod()` join point because
   - dynamic type of `o3` is `C3`, which “is-a” `C2` and

Table 2: Results of execution, where `f` is declared as `F f = new C3()`.

Code in Demo	Call	Execution
`f.anEnforcedMethod()`	`P_CALL_F:` `call(void F.anEnforcedMethod())`	`P_EXE_C3:` `execution(void C3.anEnforcedMethod())` `P_EXE_F:` `execution(void C3.anEnforcedMethod())`

- the signature of `anOverriddenMethod()` exists in `C2`.

Applying our guidelines on the example in Table 2 (where `f` is defined as `F f = new C3()`), we can see that

1. `P_CALL_F(): call(* F.*())` pointcut captures `f.anEnforcedMethod()` join point because
   - static type of `f` is `F`, and
   - the signature of `anEnforcedMethod()` exists in interface `F`.
2. `P_EXE_C3(): execution(* C3.*())` pointcut captures `f.anEnforcedMethod()` join point because
   - dynamic type of `f` is `C3`, and
   - the signature of `anEnforcedMethod()` exists in class `C3`.
3. `P_EXE_F(): execution(* F.*())` pointcut captures `f.anEnforcedMethod()` join point because
   - dynamic type of `f` is `C3`, which “is-a” `F` and
   - the signature of `anEnforcedMethod()` exists in interface `F`.

More examples and results are summarized in Table 3 (where `o3` is declared as `C3 o3 = new C3()`) and Table 4 (where `o23` is declared as `C2 o23 = new C3()`).

Our initial intent was to capture one call and one execution in class `Demo` since only one call is made by the user and only one method body is executed. In order to reach this goal we propose one possible solution which is applicable for both `call` and `execution` pointcuts. Thus we define `P_CALL_C1` as:

```
call(* C1.*())
&& if(thisJoinPoint.getSignature().getDeclaringType() == C1.class)
```

And `P_EXE_C1` as:

```
execution(* C1.*())
&& if(thisJoinPoint.getSignature().getDeclaringType() == C1.class)
```

In the same manner we must define all other pointcuts previously addressed.

Table 3: Results of execution, where **`o3`** is declared as **`C3 o3 = new C3()`**.

Code in Demo	Call	Execution
`o3.aVeryNewMethod()`	`P_CALL_C3:` `call(void C3.aVeryNewMethod())`	`P_EXE_C3:` `execution(void C3.aVeryNewMethod())`
`o3.anEnforcedMethod()`	`P_CALL_C3:` `call(void C3.anEnforcedMethod())` `P_CALL_F:` `call(void C3.anEnforcedMethod())`	`P_EXE_C3:` `execution(void C3.anEnforcedMethod())` `P_EXE_F:` `execution(void C3.anEnforcedMethod())`
`o3.aNewMethod()`	`P_CALL_C2:` `call(void C3.aNewMethod())` `P_CALL_C3:` `call(void C3.aNewMethod())`	`P_EXE_C2:` `execution(void C2.aNewMethod())`
`o3.anOverriddenMethod()`	`P_CALL_C1:` `call(void C3.anOverriddenMethod())` `P_CALL_C2:` `call(void C3.anOverriddenMethod())` `P_CALL_C3:` `call(void C3.anOverriddenMethod())`	`P_EXE_C1:` `execution(void C2.anOverriddenMethod())` `P_EXE_C2:` `execution(void C2.anOverriddenMethod())`
`o3.anInheritedMethod()`	`P_CALL_C1:` `call(void C3.anInheritedMethod())` `P_CALL_C2:` `call(void C3.anInheritedMethod())` `P_CALL_C3:` `call(void C3.anInheritedMethod())`	`P_EXE_C1:` `execution(void C1.anInheritedMethod())`

Table 4: Results of execution, where **`o23`** is declared as **`C2 o23 = new C3()`**.

Code in Demo	Call	Execution
`o23.aNewMethod()`	`P_CALL_C2:` `call(void C2.aNewMethod())`	`P_EXE_C2:` `execution(void C2.aNewMethod())`
`o23.anOverriddenMethod()`	`P_CALL_C1:` `call(void C2.anOverriddenMethod())` `P_CALL_C2:` `call(void C2.anOverriddenMethod())`	`P_EXE_C1:` `execution(void C2.anOverriddenMethod())` `P_EXE_C2:` `execution(void C2.anOverriddenMethod())`
`o23.anInheritedMethod()`	`P_CALL_C1:` `call(void C2.anInheritedMethod())` `P_CALL_C2:` `call(void C2.anInheritedMethod())`	`P_EXE_C1:` `execution(void C1.anInheritedMethod())`

Access and modification of instance variables

In this case our goal is to capture accesses and modifications of instance variables. Consider the following definitions:

```
public class C1 {
  public int x;}

public class C2 extends C1 {}

public class Demo {
  public static void main(String[] args) {
    C2 o = new C2();
    o.x++;}}
```

We also define an aspect, which monitors the accesses and modifications of all instance variables in both classes:

```
public aspect FieldAccessModification {
  pointcut accessorC1():
    get(* C1.*);
  pointcut accessorC2():
    get(* C2.*);
  pointcut mutatorC1():
    set(* C1.*);
  pointcut mutatorC2():
    set(* C2.*);

  after(): accessorC1() {...}
  after(): accessorC2(){...}
  after(): mutatorC1(){...}
  after(): mutatorC2(){...}}
```

In Java, whenever a class instance is created, memory space is allocated for it with room for all the instance variables declared in the class and all instance variables declared in each superclass of the class, including those that may be hidden [Gosling et al., 2005]. Thus, these instance variables are accessed and modified when an instance of the class is manipulated. Based on this, one third of the participants expected two captured join points by running the above example - one for modification and one for access, which is:

```
Accessed(accessorC2): get(int C2.x)
Modified(mutatorC2): set(int C2.x)
```

In contrast, for a similar example the *AspectJ Language Guide* explains that for a `get` pointcut where a class does not directly declare a member, the join point matches each superclass up to and including the most specific supertype that does declare the member. This resumes in having two join points captured for the access pointcut. The guide also explains that the signatures for a `set` pointcut are derived in an identical manner, which in our case will result in having also two join points captured for modification pointcut.

The abc compiler: If the abc compiler follows the semantics defined in the *AspectJ Language Guide*, we will expect the following output:

```
Accessed(accessorC1): get(int C1.x)
Accessed(accessorC2): get(int C2.x)
Modified(mutatorC1): set(int C1.x)
Modified(mutatorC2): set(int C2.x)
```

This output was predicted by one third of the participants. The above result was verified by running the program[8]. In order to reach our initial intention, which is capturing the number of real modifications and accesses of the instance variable x, we need to modify the definitions of the pointcuts as follows:

```
public aspect FieldAccessModification{
  pointcut accessorC1():
    get(* C1.*)
    && if(thisJoinPoint.getSignature().getDeclaringType() == C1.class);
  pointcut accessorC2():
    get(* C2.*)
    && if(thisJoinPoint.getSignature().getDeclaringType() == C2.class);
  pointcut mutatorC1():
    set(* C1.*)
    && if(thisJoinPoint.getSignature().getDeclaringType() == C1.class);
  pointcut mutatorsC2():
    set(* C2.*)
    && if(thisJoinPoint.getSignature().getDeclaringType() == C2.class);
...}
```

Thus we observe our original expected behavior, which is:

```
Accessed(accessorC2): get(int C2.x)
Modified(mutatorC2): set(int C2.x)
```

This result corresponds to one access and one modification of instance variable x in class C2.

Currently, changing the type of the instance variable x to a user defined type, however, results in different behavior. We observe one modification of x in class C1 and one access of x in class C2[9]. The insufficient documentation regarding this behavior may be misleading and may decrease the comprehensibility of the language.

[8]For access and modification of all other primitive types we observe the same behavior.
[9]The aspect pointcuts do not yet contain the proposed solution.

The ajc compiler: The result with the ajc compiler differs from what is explained in *AspectJ Language Guide* and was expected only by one of the participants. We observe the following output after compiling the program with the ajc compiler:

```
Accessed(accessorC1): get(int C1.x)
Modified(mutatorC1): set(int C2.x)
Modified(mutatorC2): set(int C2.x)
```

It seems that for primitive types, reasoning about field access is based on the exact class where the field is defined. For field modification pointcut however, checking is based on the class where the field is defined or whose parent has the field definition. The above output seems to contradict what is specified in the *AspectJ Language Guide*.

The solution previously discussed in Paragraph The abc compiler is applicable when using the ajc compiler and the result is as following:

```
Accessed(accessorC1): get(int C1.x)
Modified(mutatorC2): set(int C2.x)
```

The above result is interpreted as one access of `x` in class `C1`, and one modification of `x` in class `C2`, which is different from the result previously discussed in Paragraph The abc compiler. Regarding the number of accesses and modifications, the result matches with our expectation. However, there seams to be a conflict between the two compilers regarding the place where the access is performed.

Currently, changing the type of the instance variable `x` to a user defined type, however, results in different behavior from the one observed with the abc compiler. We observe one modification and one access of `x` both in class `C1`.

Overall we think that field access and modification should be documented better in the literature in order to improve the understanding of program readers.

Purpose of initialization

Our intent is to observe the behavior of initialization pointcut in AspectJ. The result of all above mentioned cases is that AspectJ follows also Java specification. Therefore, our intuition in this case is also based on Java specification. In Java, a class is initialized by invoking static initializers and initializers for static fields of the class. Before the class is initialized, its direct superclass is initialized, but not the interfaces implemented by the class (if any). In the same manner, the superinterface of an interface does not need to be initialized when the subinterface is initialized [Gosling et al., 2005]. Consider the class diagram on Figure 4 and the following definitions:

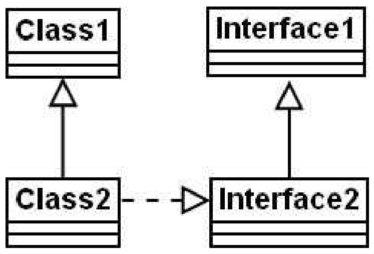

Figure 4: Class diagram - purpose of initialization.

```
public class Demo {
  public static void main(String[] args){
    Interface2 c= new Class2();}}

public aspect A {
  pointcut P_Init():
    initialization(*.new(..)) && !within(A);
  before():P_Init(){...}}
```

One third of the participants expected the following result based on the fact that the `Interface1` and `Interface2` will not be initialized (according to the above discussion):

```
initialization(Class1())
initialization(Class2())
```

This corresponds to initialization of superclass `Class1`, followed by initialization of `Class2` itself. Referring to the *AspectJ Language Guide* however, `initialization` join point captures call to constructor through the keyword `new`, and invocation of non-static initializers of superclasses and superinterfaces. There are few rules regarding the order of initialization within inheritance hierarchy chain:

- A supertype is initialized before the subtype.
- A superclass is initialized before superinterface.
- Initializers are executed once.

The ajc compiler In what seems to be in contradiction with the *AspectJ Language Guide*, after running the program with the ajc compiler, we observed the following result:

```
initialization(Class1())
initialization(Class2())
initialization(Interface2())
initialization(Interface1())
```

Note that `Class2` is initialized before the initialization of its superinterfaces. In addition, it seems that `Interface2` is initialized before `Interface1`. Both observations indicate that the above rules are not followed by the ajc compiler.

The abc compiler Similarly, the abc compiler provides the following output:

```
initialization(Class1())
initialization(Class2())
initialization(Interface1())
initialization(Interface2())
```

In this case the order of initialization of interfaces follows the above rules, but still `Class2` is initialized before the initialization of its superinterfaces which seems to contradict the *AspectJ Language Guide*.

Overall, it is important to note that initialization in Java and `initialization` pointcut in AspectJ are two different concepts, which means that `initialization` pointcut in AspectJ does not capture the initialization such as defined in Java. Therefore, in order to capture initialization as defined in Java, one should use `staticinitialization` pointcut.

6 RELATED WORK

In [Barzilay et al., 2004] the authors argue about certain “unintuitive aspects” of AspectJ. They observed certain unexpected results with AspectJ (version 1.1.1) regarding the semantics of `call` and `execution` pointcuts and proposed alternative semantics. Their observations and guidelines can not be applied to the current version of AspectJ (1.5.2). The authors showed that for matching a `call` pointcut the method should be defined in the specific class (inheritance is not a sufficient condition). In Section 5 however we explain that inheritance is now enough. Also, for matching an `execution` pointcut the authors explained that the method should be defined or overridden, whereas in Section 5 we show that currently a signature is enough.

In [Breu, 2005] the author introduces an aspect mining tool called DynAMiT, which uses AspectJ `execution` pointcut for trace generation. The authors mentions about the incapability to trace system methods. In Section 5 we provided a more detailed explanation for call and execution of system methods.

7 CONCLUSION

In this article we addressed the comprehensibility property of the AspectJ programming language as a notable representative of AOP technology, having gained an increasing acceptance both in industry as well as in academia. There seems to be two different issues that affect the comprehensibility property of AspectJ: The first is a conflict between the two different implementations, ajc and abc. The second is the conflict between what the two implementations seems to have intended and what program readers seem to expect. We identified cases where the provisions of the language do not seem to be intuitive or lack of detailed documentation. We provided a reasoning in cases where the behavior does not seem to be obvious for the participants which may result in lack of comprehension and also may affect the correctness of systems. Programmers should be careful when writing AspectJ programs and in some cases, should take into consideration the semantics of the underlying language - Java. Finally, we feel that by providing more clarity on some subtle cases, the understandability property as well as the usage of the AspectJ programming language can be increased considerably. We also feel that lessons learned from this experience can be beneficial to language designers and practitioners alike.

ACKNOWLEDGMENTS

We would like to thank Therapon Skotiniotis for his comments during the preparation of this article.

REFERENCES

[Barzilay et al., 2004] Barzilay, O., Feldman, Y. A., Tyszberowicz, S., and Yehudai, A. (2004). Call and Execution Semantics in AspectJ. In *Proceedings of the 3rd AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL)*.

[Breu, 2005] Breu, S. (2005). Extending Dynamic Aspect Mining with Static Information. In *Proceedings of the 5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*.

[Colyer et al., 2004] Colyer, A., Clement, A., Harley, G., and Webster, M. (2004). *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Addison-Wesley.

[Dijkstra, 1976] Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice Hall.

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

[Gosling et al., 2005] Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java Language Specification*. Addison Wesley, 3rd edition.

[Hilsdale and Hugunin, 2004] Hilsdale, E. and Hugunin, J. (2004). Advice Weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*.

[Kienzle et al., 2003] Kienzle, J., Yu, Y., and Xiong, J. (2003). On Composition and Reuse of Aspects. In *Proceedings of the 2nd AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL)*.

[Lorenz and Kojarski, 2006] Lorenz, D. H. and Kojarski, S. (2006). Feature Interaction in AspectJ 5. In *Proceedings of the 5th AOSD Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT)*.

[Masuhara et al., 2002] Masuhara, H., Kiczales, G., and Dutchyn, C. (2002). Compilation Semantics of Aspect-Oriented Programs. In *Proceedings of the 1st AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL)*.

[Parnas, 1972] Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058.

[Penta et al., 2007] Penta, M. D., Stirewalt, R. E. K., and Kraemer, E. (2007). Designing your Next Empirical Study on Program Comprehension. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC)*.

[The AspectJ Team, 2006] The AspectJ Team (2006). The AspectJ Language Guide.

ABOUT THE AUTHORS

Venera Arnaoudova is a graduate research assistant at the Department of Computer Science and Software Engineering of Concordia University, Canada. She can be reached at v_arnaou@encs.concordia.ca.

Laleh Mousavi Eshkevari is a PhD student and research assistant at the Department of Computer Science and Software Engineering of Concordia University, Canada. She can be reached at l_mousa@encs.concordia.ca.

Elaheh Safari Sharifabadi is a graduate research assistant at the Department of Computer Science and Software Engineering of Concordia University, Canada. She can be reached at e_safari@encs.concordia.ca.

Constantinos Constantinides is an Assistant Professor at the Department of Computer Science and Software Engineering of Concordia University, Canada. He holds an MS degree in Computer Science from the New York Institute of Technology and a PhD degree in Computer Science from the Illinois Institute of Technology. He can be reached at cc@encs.concordia.ca.