Aspect Instances and their Interactions

Therapon Skotiniotis Karl Lieberherr David H. Lorenz

College of Computer & Information Science
Northeastern University
360 Huntington Avenue 161 CN
Boston, Massachusetts 02115 USA
{skotthe,lieber,lorenz}@ccs.neu.edu

ABSTRACT

Programming languages and paradigms provide mechanisms by which we can express our own solutions to problems. The way by which one formulates a solution is affected by the medium with which the solution is expressed. Currently, the lack of polymorphism and the difficulty with which aspect instances can be accessed and used within AspectJ, force programmers to resolve to less elegant solutions, introducing code tangling in advice definitions, increasing code complexity and diminishing maintainability and robustness.

Through examples in AspectJ we discuss issues that relate to aspect instances, their interactions, and also the reflective capabilities of an AOSD language. We argue that any AOSD language should provide the means by which programmers can refer and manipulate aspect instances with the same ease and simplicity as with object instances in OO languages. Reflective capabilities inside AOSD languages should allow for introspection on their join point and pointcut mechanisms, allowing for the acquisition and manipulation of runtime information about these entities.

We hope that by exposing these issues for discussion will allow for a better assessment of these techniques as well as a community-wide decision as to what should comprise "good" or "bad" techniques based on "proper" or "improper" use of an AOSD technology.

1. INTRODUCTION

The open source AspectJ project has evolved into one of the main stream AOSD technologies. The evolution of the language has, and will continue, to provide a general purpose effective platform to accommodate the community's needs in the area of AOSD [9]. In the latest stable release of the language (version 1.0.6), explicit mechanisms for the creation and manipulation of multiple aspect instances, has been added. Having these new features allows for the exploration of bigger, more complex, AOSD systems. This paper addresses the questions:

- How can one interact with an instance of an aspect, when the aspect is `singleton` or when we have multiple instances.
- How does the design of one's aspects change, if at all, when you have to deal with multiple instances.
- What is the final impact on the programs complexity, modularity and understanding.

Through the use of the example presented in Section 1.1, we show that a programs complexity increases in situations where multiple instances of aspects are present. Furthermore, the current implementation of AspectJ lacks polymorphic features [4] when it comes to aspect instances, decreasing the impact of one's aspect design. Even though the above deficiencies are present, programmers can still provide workarounds using AspectJ's reflective capabilities. These solutions, however, come at a cost, increasing code complexity, maintainability and modularity. Section 2 presents each individual case and possible solutions exposing some of AspectJ's idiosyncrasies. Section 3 concludes the paper.

1.1 Program Setup

The example used for the rest of the discussion is an extension of the `Tracing` example found in the AspectJ Programmers Manual[1] [8]. The example consists of two dimensional shapes class hierarchy (Figure 1) [2] along with an abstract aspect `AbstractTrace` (this abstract aspect with few minor modifications is found in Listing 1). In order to address each aspect interaction case in isolation, a concrete aspect is implemented, defining the appropriate abstract pointcut `methods()`, as well as any extra advice and/or methods that might be needed, in each case.

Listing 1: Altered `AbstractTrace` aspect definition

```
public abstract aspect AbstractTrace {
 pointcut classes(): within(tracing.*) &&
   (!within(tracing.lib.*) ||
   !cflow(withincode(* tracing.lib.*(..)))
   || !within(java.lang..*) ||
   !cflow(withincode(* java.lang..*(..))));

 abstract pointcut methods();
 // same as [8] ...
}
```

[1]Basic familiarity with AspectJ, and with the examples that are found in [8], is assumed

[2]Which is an extension of the example program that is provided with an installation of AspectJ. The classes that are provided in the AspectJ distribution (version 1.0.6) are `TwoDShape` as an abstract class with `Circle` and `Square` as two distinct concrete subclasses

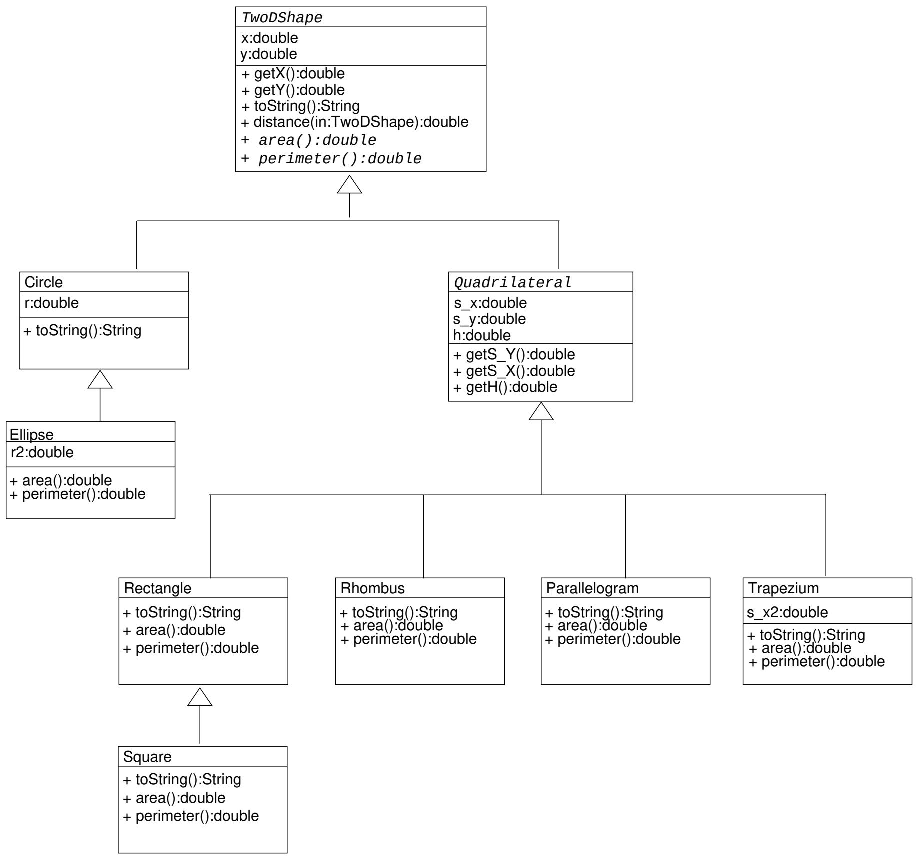

Figure 1: Class Diagram of the modified AspectJ's shape's example

2. EXTENDING THE TRACING EXAMPLE

In this section we first illustrate that the simple task of defining `methods()`, is not that simple at all. Subsection 2.1, exposes an idiosyncrasy found in AspectJ, specifically that of defining a pointcut to capture calls to a type pattern, and *only* that pattern. We provide the "idiom" that solves this problem and its use, along with AspectJ's reflective capabilities that allows one to capture calls on objects that have different *static* and *dynamic* types. Subsection 2.2 illustrates `singleton` aspect interactions and subsection 2.3 looks at interactions of multiple aspect instances.

2.1 Dealing with Type Patterns and Reflection

As a simple example, consider tracing all calls that occur within the `tracing` package.

Listing 2: The concrete `TraceMyClass` aspect definition

```
aspect TraceMyClasses extends AbstractTrace {
 pointcut methods(): call(* TwoDShape.*(..));
}
```

Observe that using the type pattern `TwoDShape`, the root of the class diagram, is enough to allow us to capture all calls that are made by all possible subtypes. There is no need to use `TwoDShape+` to denote the type pattern itself and all its subtypes.

In Java every class definition automatically introduces a new type into the system. In the class diagram found in Figure 1 an instance of `Square` introduces a new type $Square$ but is also a subtype of $Rectangle$. Subtype polymorphism allows us to use an instance of `Square` and refer to it as of type $Rectangle$.

To deal with subtype polymorphism AspectJ provides within the definition of pointcuts an explicit mechanism to denote subtypes (+). In our example, consider an aspect that needs to trace all calls that occur **only** on the type $Rectangle$ but not on any of its subtypes. At first one would be tempted to use a pointcut like

Listing 3: There is no distinction between `Rectangle` and `Rectangle+`

```
pointcut methods(): call(* Rectangle.*(..));
```

Unfortunately, there is no distinction between `Rectangle` and `Rectangle+`. Listing 3 will undesirably capture calls that are made to any subclass of `Rectangle`.

From a section entitled "Idioms" in the AspectJ manual one can find a pointcut definition for capturing calls to subtypes but not the root type with the definition, as illustrated in Listing 4.

Listing 4: Idiom definition found in the "Programmer Guide"

```
call(* (Rectangle+ && !Rectangle))
```

As a second attempt, one could try to negate the idiom in Listing 4. However, that results in all types other than $Rectangle$. Another attempt would be to alter the idiom in order to capture the root type and none of its subtypes, e.g.,

Listing 5: The `&&` operation yields no join points

```
call(* (Rectangle && !Rectangle+).*(..))
```

but the above pointcut definition results in **no** advice ever getting executed. It appears that since `Rectangle` and `Rectangle+` denote the same set of types in your program, the pointcut in Listing 5 is actually an empty set. The pointcut in Listing 4 seems the same as that in Listing 5, but bound differently on the same operators, denoting different sets of types.

Listing 6: Capturing calls `Rectangle`, but none of its subclasses

```
pointcut methods(): call(* Rectangle.*(..))
  && !call(* (Rectangle+ && !Rectangle).*(..));
```

Surprisingly enough the solution is the pointcut in Listing 6, in plain English, picks all calls to the type $Rectangle$ and, **no** calls to any subtypes of $Rectangle$. There seems to be a duality in the type pattern definitions inside a pointcut definition. It is "nice" at times that we could capture all subtypes by simply providing the root type of our class diagram. However, it comes with a price. The explicit use of the subtype pattern mechanism, denoted by +, when used inside a conjunction with the type pattern itself forces the type pattern to behave in the correct manner. That is, pick the type $Rectangle$ and **only** $Rectangle$. The same behavior can also be observed with the use of the `execution` pointcut primitive.

Another interesting case that has to do with types and the reflection mechanisms of AspectJ, arises when you would like to catch calls to instances that are used inside the source code under a different type than the type of the runtime instance. This is typically the case when an instance of an object o is instantiated with type τ and is then casted to a different type (supertype in the case of Java) τ'. In the following code fragment:

Listing 7: An example where dynamic and static types are not the same

```
Rectangle r = new Square();
r.area();
```

The **dynamic type** of `r` is $Square$ while its **static type** is $Rectangle$. If one would like to trace this types of calls in the code one solution could be:

Listing 8: Getting the types of source and target via AspectJ's reflective capabilities

```
public abstract aspect AbstractTrace {
 ...
before(): classes() && methods() &&
 if(thisJoinPoint.getSignature().getDeclaringType()
     .getName().compareTo(thisJoinPoint.getTarget()
         .getClass().getName()) != 0) {
  doTraceEntry(thisJoinPoint, false);
   System.out.println(thisJoinPoint.
    getSignature().getDeclaringType().getName());
   System.out.println(thisJoinPoint.
    getTarget().getClass().getName());
   }
}
```

The code fragment above explores some of the reflective capabilities that AspectJ provides. Attempting to express the above `if` pointcut definition using `target` and an appropriate type pattern (Listing 4) does not work. The idiom provided in the AspectJ manual behaves differently in this context and yields no joinpoints. More information could be obtained, like source location as well as the arguments of the call. One can use these pieces of information in order to provide finer execution control inside an application. Furthermore, we have found that access to the program's information through AspectJ's reflective API to be extremely helpful in debugging but also understanding a program's behavior. We believe that such information are vital for any AOSD language.

Reflective information along with the non-invasive addition of program behavior through advice make AOSD technologies good candidates for the enforcement/checking of design rules in Java programs, or even the generation of code that could, at runtime, perform checks for OCL [1] or even pre and post conditions (Design by Contract) [5, 7, 3]. Allowing for clear separation between the actual program and its constraints (e.g. contracts) in a modular and more reusable manner than from what is currently available by other technologies.

2.2 Aspect Interactions

Another interesting scenario is the one where you would like to have interactions between instances of aspects. These interactions encode operations as methods inside aspects and then reuse these methods. The alternative could be to create extra Java classes that will contain these methods instead of placing them inside aspect definitions. This alternative, however, introduces new classes and in some cases this approach might not be desirable[3]. By placing all of the code inside aspects allows for a more modular, and reusable design, limiting any further coupling inside aspects [2].

Listing 9: The `AbstractCollector` aspect, obtaining the information from `TraceCircle` and `TraceRectangle` aspects

```
aspect AspectCollector{
 AbstractCollection temp;
  pointcut main(): call(* main(..));
   after():
    main(){
   temp=TraceRectangle.aspectOf().getTargets();
   prettyPrint(temp, new String("Rectangle"));
   temp=TraceCircle.aspectOf().getTargets();
   prettyPrint(temp, new String("Circle"));
    }
 private void prettyPrint(
     AbstractCollection ele, String source){
  Iterator it = ele.iterator();
   while (it.hasNext()){
    String name = (String)it.next();
    System.out.println("Source:" +
      source + "Targets:" + name);
      }
    }
  }
```

For example, consider the case where we would like at the end of a programs execution to obtain the classes of all the calls whose dynamic type and static type are different. An aspect (`singleton`) is created for each non-abstract class that is also a superclass for some other class(es) in the program. For each of these classes we have, an aspect (`TraceRectangle` and `TraceCircle`) that captures calls and inspects the types for `source` and `target`.

[3] Consider the case where you would like to read in any Java program and create aspects and new classes to accommodate runtime checks of OCL or even Contract definitions. By creating new classes the programmer has to worry about class name clashes and ambiguity of class name resolutions between packages.

If the two types differ, then they are stored in an AbstractCollection that is a member of the instance of each concrete trace aspect. A third aspect (AspectCollector) after main() completes execution, obtains from the two aspects, the two AbstractCollections and displays the information that were collected during execution. The AbstractTrace definition will have to be appended with an extra abstract method (Listing 10).

Listing 10: Appending AbstractTrace with the new method definition

```
 ...
abstract AbstractCollection getTargets()
 ...
```

TraceRectangle, and similarly TraceCircle, will have to be altered as follows:

Listing 11: Using arguments in point cuts and getting the correct references to aspect instances

```
aspect TraceCircle extends AbstractTrace {
 AbstractCollection targets = new Vector();
 pointcut methods():
   !call(* (Circle+ && !Circle).*(..))
   && call(* Circle.*(..));

  before(): methods(){
   doTraceEntry(thisJoinPoint, false);
   }

 private void doTraceEntry(JoinPoint jp,
   boolean isConstructor) {
  Class source, target;
  source=jp.getSignature().getDeclaringType();
  target=jp.getTarget().getClass();
  if (source.getName().
     compareTo(target.getName()) != 0){
  if(!targets.contains(target.getName())){
   boolean temp=targets.add(target.getName());
  }
 }
 }
 public AbstractCollection getTargets(){
   return targets;
 }
}
```

In this manner all operations, and code that address the issue of tracing and collecting type information, reside inside aspect definitions leaving the original classes untouched. This achieves encapsulation of all related operations for the addressed concern to be encapsulated entirely inside aspect definitions achieving proper separation of concerns.

The mechanisms by which one can define instance methods (as well as class methods) for aspects is essentially identical to the way that OO languages provide these mechanisms for typical objects. The difference lies in the case of instance methods. There is no way by which one can "instantiate" an aspect in AspectJ. Thus one cannot create a binding to an instance within the source code and then, through that binding, call methods of the aspects instance. To bind an aspect instance, one must use aspectOf().

An alternative approach is to allow the programmer to instantiate an aspect, and by doing so, the aspect starts participating in the program's execution. This is a technique that has been implemented in AspectS [6]. Binding an aspect is therefore an instantiation within your code. Although the difference of these two approaches seems minute, the outcomes are not. The second approach, gives the programmer full control over the lifetime of an aspect, with the ability to enable or disable the aspect at specific points within the program's execution. AspectJ does not provide any control when it comes to enabling an aspect at specific points. Instead, one has to encode this using a flag and an if statement to wrap the advice code, or changing a pointcut definition with an if pointcut designator, tangling the concern throughout, possibly, many aspects that might require this feature. Also aspect-aspect interactions become, in essence, identical to object-object interactions without introducing code in your aspects to acquire references to other aspects. In this manner advice code is not "polluted" with calls to aspectOf().

2.3 Aspect Instances

A more interesting case arises when your aspects are not singleton as in Section 2.2 but are rather pertarget, percflow, perthis or percflowbelow. AspectJ provides a mechanism with which you could access these instances, provided that the access method is given a parameter of type Object(e.g. aspectOf(Object obj)). We can think of two cases

1. There is at most one aspect instance attached to some object instances.
2. There are more than one aspect instance, each instance being of a different aspect definition, and they are all attached on the same object instance(s).

For case 1 above consider the examples of TraceCircle and TraceRectangle but this time defined as below

Listing 12: Altered TraceCircle aspect definition to associate an instance of the aspect to each instance of a target object

```
aspect TraceCircle extends AbstractTrace
pertarget(call(* Circle.*(..))){
  AbstractCollection targets = new Vector();
  pointcut methods():!call
    (* (Circle+ && !Circle).*(..))
    && call(* Circle.*(..));
  ...
```

This will attach an instance of the aspect to each instance of Circle but not to any of its subclasses. The corresponding code changes, and effect, will also occur for TraceRectangle and Rectangle. AspectCollector though will have to change since the AspectJ compiler now complains since the method aspectOf() is no longer available in TraceRectangle and TraceCircle. Instead, you have to pass the right object as an argument to the call aspectOf(). The right object can be obtained from the pointcut itself, by binding the target object of a call within the pointcut and passing it to the advice code.

Following the ideas from OO Programming you would expect that since both of the concrete aspects extend the abstract aspect AbstractTrace that an addition to AspectCollector like

Listing 13: Getting a reference to an instance of an aspect

```
temp=AbstractTrace.asepectOf(obj).getTargets();
```

would deal with each case in the right way. That is since there is only one aspect instance and of different "type" the AspectJ compiler would call the correct method on the correct instance. However this is not really the case. The compiler complains for the above line of code, with the error that there is no such method defined for `AbstractTrace` [4]. A workaround can be achieved by wrapping your code with tests to check for the existence of a concrete aspect through the usage of `hasAspect()`. This approach might not work however if you do not have all possible aspect definitions that could, potential, be attached at a specific pointcut. It is even more cumbersome, if you would like to find out all aspects that are attached to a pointcut since you have to manually check and collect all such instances. The language itself provides no mechanism through which one could collect this information in a more modular manner.

Listing 14: `AbstractCollector` aspect definition dealing with multiple instances of aspects

```
aspect AspectCollector{
 AbstractCollection temp;
 pointcut circles(Circle cc):
   call(* Circle.*(..)) &&
   !call(* (Circle+ && !Circle).*(..))
   && target(cc);
 pointcut rectangles(Rectangle rr):
   call(* Rectangle.*(..)) &&
   !call(* (Rectangle+ && !Rectangle).*(..))
   && target(rr);

 after(Circle cc):
   circles(cc){
     temp = TraceCircle.
       aspectOf(cc).getTargets();
    prettyPrint(temp,
        new String("Circle"), cc);
   }
 after(Rectangle rr):
   rectangles(rr){
     temp = TraceRectangle.
       aspectOf(rr).getTargets();
    prettyPrint(temp,
        new String("Rectangle"), rr);
   }
 private void
   prettyPrint(AbstractCollection ele,
     String source, Object obj){
   Iterator it = ele.iterator();
   while (it.hasNext()){
     String name = (String)it.next();
     System.out.println("
         Source:" + source +
         "hashNumber:" + obj.hashCode()+
         "Targets:" + name);
   }}}
```

Looking at case 2, we might have more than one "type" of aspect instance attached to an object. In this case you either have to explicitly check for all the possible aspects that are attached to the instance and pick the one you are interested in. It becomes even harder, since the burden of how to decide which aspect is more appropriate for your needs essentially requires to evaluate `hasAspect()` for all aspects in the system, or possible aspects that might be found on a specific object instance, by guessing pointcut results. However, even this tedious solution to the problem is not always possible. Consider an application that reads in a Java program and generates aspect code (on the fly) to check for design constraints. The programmer does not have all possible aspect definitions before hand, nor can he evaluate possible aspect definitions that might appear at different pointcuts.

There should be better mechanisms to both refer but also use aspect instances and their methods. The lack of polymorphism when it comes to aspects in AspectJ makes the use of the language cumbersome when it comes to large and complex aspect oriented solutions. The work-arounds to these issues introduce nested `if` statements inside you advice, bringing back tangled code, this time inside advice definitions, along with all its disadvantages that AOSD was proposed to solve.

3. CONCLUSION

We have shown, through AspectJ examples, the shortcomings of an AOSD language which does not provide polymorphism for aspects or explicit aspect instantiation and referencing as part of the language. We further argue that, lack of these features causes programmers to come up with solutions that introduce tangled code inside advice definitions. As a result, increasing code complexity, decrease code maintainability and modularity, and making program extension/evolution tedious and error prone.

In trying to find possible solutions, we believe that the approach of aspect instantiation taken by AspectS [6] is more appropriate. AspectS allows the instantiation of an aspect (via `new`), the activation of an aspect (via `install`[5]). Obtaining a reference to an aspect instance in AspectS is a matter of a simple assignment at instantiation time, and deployment of the aspect becomes a simple call to its `install` method. Being able to refer and manipulate aspects as first class artifacts during the execution of a program, could be explored further to provide better mechanisms for a large portion of applications [3]. Addressing the issue of aspect polymorphism, A recent paper [4] presents the issues related with aspect polymorphism, in more detail and discusses possible solutions.

We therefore conclude that features like aspect instantiation, activation and aspect polymorphism, are essential features for any AOSD technology. Further analysis and research for finding appropriate solutions to these issues remains A recent paper[4] addresses the issue of aspect polymorphism in more detail, and discusses possible solutions.

We therefore conclude that features like, aspect instantiation, activation as well as aspect polymorphism, are essential features for any AOSD technology. Further analysis and research for finding appropriate solutions to these issues remains an open question for discussion.

4. ACKNOWLEDGMENTS

We would like to thank all of the students in COM3360 and COM3362 for bringing up questions and valuable example programs. We thank Paul Freeman for his input and ideas, and the members of the AspectJ's users mailing list.

5. REFERENCES

[1] *The Object Constraint Language*. http://www-3.ibm.com/software/ad/library/standards/ocl.html.

[4] A more detailed discussion on this issue, along with possible solutions, can be found at[4]

[5] Deactivation in AspectS is provided through `uninstall`

[2] Constantinos Constantinides and Youssef Hassoun. Visibility considerations and code reusability in aspectj. In *Aspect-Oriented Software Development of the SIG Object-Oriented Software Development, German Informatics Society*, Essen, March 4-5 2003 (To Appear).

[3] Constantinos A. Constantinides and Therapon Skotiniotis. Reasoning about a classification of crosscutting concerns in object-oriented systems. In *Second International Workshop on Aspect-Oriented Software Development of the SIG Object-Oriented Software Development*, Bonn, February 21-22 2002. German Informatics Society.

[4] Erik Ernst and David H. Lorenz. Aspects and polymorpshism in aspectj. In *International Conference on Aspect Oriented Software Development*, Boston, March 2003 (To Appear).

[5] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *Conferece on Object Oriented Programming, Systems, Languages and Applications*, number 11 in ACM SIGPLAN Notices, pages 1–15, Tampa Bay, Florida, October 2002.

[6] Robert Hirschfeld. Aspect-oriented programming with aspects. http://www-ia.tu-ilmenau.de/ hirsch/Projects/Squeak/AspectS/AspectS.html.

[7] Cristina Lopes and Martin Lippert. Design by contract with aspect-oriented programming, 1999. U.S. Patent No. 09/426,142. (Pending).

[8] The AspectJ Team. *The AspectJ Programmers Guide*. http://www.eclipse.org/aspectj/.

[9] The AspectJ Team. The AspectJ Web Site. http://www.eclipse.org/aspectj/.