A DSL to declare aspect execution order

Antoine Marot * †
Université Libre de Bruxelles (ULB)
amarot@ulb.ac.be

Roel Wuyts
IMEC Leuven and KU Leuven
wuytsr@imec.be

Abstract

Composing aspects is known to be problematic since unpredicted aspect interactions may appear and may lead to erroneous weaved programs. This paper focuses on one of these issues: the advice ordering around a join point. It views aspect composition issues as a crosscutting concern that should be handled by a composition aspect. It proposes a domain-specific declarative aspect composition language for composing aspects, and applies it on a number of examples.

Categories and Subject Descriptors D.1.m [*Programming Techniques*]: Aspect-Oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Aspects, Software Composition, Evolution

1. Introduction

The goal of aspect-oriented programming is separation of concerns, where each concern could be developed by an expert in the problem domain tackled by that concern. A domain specialist in transactional programming should be able to design a transactional aspect language that could then be composed in programs that need transactions. Initially these aspect languages were supposed to be domain-specific languages, as illustrated by for example the COOL language. Because of the complexity of designing weavers for more-or-less general purposes aspect languages, research on domain-specific aspect languages was taking the low road. Recently, as shown also by this workshop series, domain-specific aspect languages are on the rise again, for all kinds of domains.

This paper is concerned with the domain of aspect-oriented programming itself. It is a well-known AOP problem that composing multiple aspects, even if each aspect works correctly when being composed in isolation, is hard. We introduce a model where composition issues are considered as a crosscutting concern handled by a composition aspect implemented using our domain-specific aspect composition language. Our aspect composition language is declarative (to let composers focus on the actual composition) and allows to change composition orders at runtime (since this is needed in practice, especially when the composed aspects were developed in isolation).

Note that we submitted a paper similar to this one to the AOSD'08 workshop "6th Workshop on Software-engineering Properties of Languages and Aspect Technologies". We will attend both workshops with a very similar position on aspect composition, but with a different focus. This paper focuses on the domain-specificness of our aspect language. The other submission focuses on the non-functional properties (the '-ilities') of the language.

The rest of the paper is structured as follows. Section 2 looks in more detail at the problem of ordering advices. Section 3 describes related work, while Section 4 introduces our own solution. Section 5 gives examples to demonstrate how frequently occurring composition problems are addressed by the language. Section 6 discusses future work for this research. Section 7 concludes the paper.

2. Aspect Ordering Issues

When several advices execute at the same join point, it is important to consider the order in which they execute. Indeed, a wrong execution order can change the purpose of an aspect.

Take for example a workflow application with two aspects *CheckAccess* and *MonitorActions*. These aspects intercept every possible action a workflow user can take: *CheckAccess* executes the action only if the user is allowed to, while *MonitorActions* prints the name of an action when it is performed by a user. If advices of *MonitorActions* were to be executed before those of *CheckAccess*, every attempt to perform an action would produce a print (even if the action wasn't performed due to the access restriction). The purpose of *MonitorActions* therefore has changed, since it was only meant to print the names of actions that were performed.

This is an example which is very similar to the fragile pointcut problem that has been observed when aspects evolve: "The fragile pointcut problem occurs in aspect-oriented systems when pointcuts unintentionally capture or miss particular join points as a consequence of their fragility with respect to seemingly safe modifications to the base program." [3].

Since composition of aspects is a major problem for the semantic of weaved programs, an AOP language should always allow programmers to specify execution order. Moreover, to increase the reusability of aspects in different compositions, this specification should not be hardcoded within the concerned aspects.

3. Existing solutions

In this section, we'll briefly introduce some existing techniques to set execution orders around join points. We'll first present an example of an AOP program that obviously needs some ordering and then show how ordering can be handled with a number of existing solutions.

---

* A. Marot is a Research Fellow of the Fonds National de la Recherche Scientifique (F.R.S.–F.N.R.S)

† This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy.

Workshop DSAL'08 April 1, 2008 Brussels, Belgium.

3.1 Running Example

A company has developed a client-server application for file hosting that is widely used. The server is used to host files. Customers use a client application to upload files via the method *send()*. The server checks all received files for viruses via the method *virusCheck()*. When uploading files, the client software first encrypts the files. Then the files are compressed in order to speed up the upload. The current system is implemented in a monolithic fashion.

The company envisions that in the near future they will probably need to add support for other compression and security techniques, which is hard to do in their current implementation. Having heard about aspect-oriented programming they decide to use this technology when reworking their implementation. More specifically they want to implement the compression and decryption concerns using two aspects. The company also wants to improve the performance of uploading files. They noted that for the particular compression algorithm they use it would be better to first compress the files and then encrypt them. Of course they want the upgraded software to be compatible with the existing older clients.

Because the antivirus analysis does not support compressed nor encrypted files the program has to decrypt and uncompress them beforehand.

Both the compression and the encryption functionalities are crosscutting concerns and they are therefore implemented in the upgraded software by the two following ASPECTJ aspects :

```
public aspect SecureUpload {

 before(File f):
   call(* Client.send(..)) && args(f) {
        encrypt(f);
 }

 before(File f):
   call(* Server.virusCheck(..))
  && args(f) {
        decrypt(f);
 }
}
```

```
public aspect CompressUpload {

 before(File f):
   call(* Client.send(..)) && args(f) {
        zip(f);
 }

 before(File f):
   call(* Server.virusCheck(..))
  && args(f) {
        unzip(f);
 }
}
```

Figure 1 shows that two join points are shared between the *SecureUpload* and *CompressUpload* aspects. This mimics the two behaviours that are possible: we can send either *encrypted zip* files or *zipped encrypted* files. An execution order is therefore needed to choose between these two possibilities.

Note that the choice between these two is composition-specific and actually depends on the interplay between the compression and encryption algorithms. Some compression algorithms work by detecting file format specific information and adapting to it. In that case the composition should do compression before encryption. On the other hand other compression algorithms might yield better results when given encrypted files because they exhibit certain patterns that can be exploited by compression techniques.

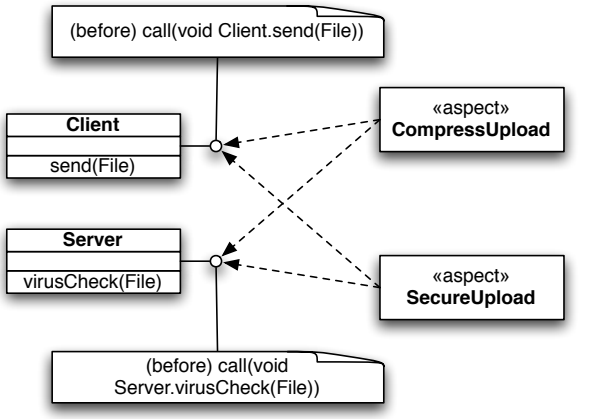

Figure 1. The client-server application including the two aspects.

An important point to make is that since the order can change in different compositions, even when the same aspects are used, information regarding the order should not be hardcoded in the individual aspects.

3.2 Related Work

The problem of ordering multiple aspects is not a new one, and there is a number of approaches and techniques that address this problem. This section reviews three techniques: *ordering aspects*, *JAsCo*, and *declarative aspect composition*.

Ordering aspects. One way to order advices is to let programmers express the order of aspects. If an aspect *A* is set to precede an aspect *B* then all advices of *A* will always be executed before any advice of *B*. This is the solution implemented in ASPECTJ[4] by the *declare precedence* statement. That technique is also used in [2] by the use of several operators between aspects. The major problem with this approach is that it makes it hard to specify advice-specific ordering (needed in our example). One could create an aspect for each advice that needs its order, but that solution modifies existing aspects and is awkward.

JAsCo. JAsCo[7] is an aspect-oriented extension of the Java language that combines AspectJ's expressive pointcuts with the Aspectual Components[5] approach of aspect independency. In JAsCo, the scope of an aspect is separated from its behavior. The crosscutting behavior is described in so-called *hooks*. A hook is composed of several advices and an abstract pointcut. *Connectors* are used to instantiate hooks and bind their abstract pointcuts to concrete ones in the program. Developers have to specify the advice sequence they want in the connector. Moreover *hook composition strategies* can be expressed that have the possibility to ignore hooks in particular composition contexts.

To implement our example using JAsCo, we would need at least four hooks, three connectors and one *connector composition strategy* (which is different from *hook* composition strategies). The connector composition strategies allows us to change dynamically the applying order of the connectors and thus of the hooks they contains. We need this since the advice ordering has to be altered if the file is sent by an older client version.

While JAsCo offers lots of flexibility it does so in a very low-level way. We expect a domain-specific approach to talk about aspect composition to be more high-level.

Declarative aspect composition A third approach is *declarative aspect composition* [6], based on declaring constraints between advices. The system then computes a possible ordering that respects all of these constraints, if possible.

There are two kinds of constraints that can be declared: *ordering* constraints and *control* constraints. An ordering constraint declares that one advice has to be executed before another one. A control constraint declares that an advice will be executed only if another one succeeded. Execution orders cannot be changed dynamically, meaning that in our example we cannot implement the backward compatibility with older versions that use compressed encrypted files instead of the newer encrypted compressed files. What we like is that a composer can focus on the actual intended composition.

4. Adding rules to aspects

This section presents our model to handle execution ordering of advices around join points. Our model is a declarative approach based on composition rules that declare constraints on the execution order of advices. Moreover rules have a priority, and the rule with the higher priority takes precedence in case of conflicting rules. The resulting model therefore, in a sense, combines the expressiveness of JASCO with the ease-of-use of the declarative aspect composition.

4.1 Composition Rules

Composition rules define constraints that the weaver will take into account while weaving the advices around join points. They are declared in aspects themselves: if an aspect needs to order its own advices, it can declare the appropriate rules.

In order to manage the constraints crosscutting the aspects, a new aspect has to be created that is dedicated to the aspect composition concern. This aspect will therefore declare rules over advices from different aspects.

Rules specify the context where they are active. If a join point meets this context, the rule is activated and will be taken into account on this join point.

The rest of this section presents the different constraints allowed in a rule, shows how a rule defines its active context and explains the extension mechanism useful to declare rule exceptions.

4.2 Constraints

There are four different constraints possible in rules:

1. $Prec(a, b)$: advice a has to be executed before advice b.
2. $First(a)$: advice a has to be the first executed advice.
3. $Last(a)$: advice a has to be the last executed advice.
4. $Ignore(a)$: advice a won't execute.

We can use the constraints to express our example on compression and decrypting. Assume that the advices for the example are named as follows: *zipFile*, *unzipFile*, *encryptFile* and *decryptFile*. We can then declare that *zipFile* precedes *encryptFile* while *decryptFile* precedes *unzipFile* with these two constraints: $Prec(zipFile, encryptFile)$, $Prec(decryptFile, unzipFile)$.

Doing this indeed declares the ordering we need. What is not yet solved however is the backwards compability with the older client version. To get such compatibility we need to be able to declare another rule that will be activated only in one particular context (when the received file is zipped instead of being encrypted). How to do this is shown in the next section.

4.3 Active context

Since ordering problems sometimes arise in particular situations only, the scope of a rule can be restricted to a precise context. This is done by parametrizing the rule with a pointcut. On each join point belonging to the pointcut, the rule will be applied.

In order to capture the advice interactions, we use a new pointcut parameter : $advices(a_1, a_2, ..., a_n)$. This parameter identifies each join point where all of the n advices interact.

This can be used to solve the remaining problem in our example, namely that, when the received file is zipped, we first have to unzip it and then decrypt it. That's what the following ASPECTJ-like aspect shows:

```
public aspect AspectComposition {

 public boolean isZipped(File f) {
   /* return true if the file is zipped */
 }

 declare rule sendFile:
  advices(zipFile,encryptFile) {
   Prec(zipFile, encryptFile);
 };

 declare rule receiveFile:
  advices(unzipFile,decryptFile) {
   Prec(decryptFile, unzipFile);
 };

 declare rule oldV:
  advices(unzipFile,decryptFile) &&
  if(isZipped(thisJoinPoint.getArgs()[0])){
   Prec(unzipFile, decryptFile);
 };
}
```

We are one step further to a full solution for our problem. One last part remains, and that is what happens when the server receives a compressed file. Indeed, the rules *receiveFile* and *oldV* declare opposite constraints and will be both activated. This is an example that shows that it should be possible to refine a rule with another rule, which is shown in the next section.

4.4 Extension mechanism

In order to allow developers to declare rule exceptions, it is possible to refine a rule by using another rule. This is done via an extension mechanism between rules.

An extending rule has a higher priority than the rules it extends. Therefore, when a conflict is found between two constraints in two different rules (one extending the other), the conflict is solved by choosing the constraint of the extending rule. The other constraint is just ignored.

Note that the rule extensibility is also useful to handle unpredicted changes since it allows to alter existing rules in order to handle new cases.

In our example, we can now declare that when the rules *receiveFile* and *oldV* interacts, the constraint of *oldV* is chosen.

```
declare rule oldV extends receiveFile:
 advices(unzipFile,decryptFile) &&
 if(isZipped(thisJoinPoint.getArgs()[0])){
  Prec(unzipFile, decryptFile);
};
```

5. Examples

This section presents two other examples using composition rules to improve both aspect expressivity and composability.

5.1 Using rules to (de)activate aspects dynamically

Let's imagine a software where two aspects are intended to react visually to certain events. Since the user doesn't want two visualizations for a single event, the following aspect enables a choice between both at launch. The example shows how rules are used to dynamically ignore execution of certain aspects.

```
public aspect ChooseVisualization {

  public boolean otherVis=false;

  before(String[] a):
     execution(* Main.main(String[]))
     && args(a) {
      /* if a contains the parameter to swap
         visualizations then otherVis is set
         to true */
  }

  declare rule defaultVis:
   advices(VisAspect1.*, VisAspect2.*)
   && if(!otherVis) {
      Ignore(VisAspect2.*);
   };

   declare rule otherVis:
     advices(VisAspect1.*, VisAspect2.*)
     && if(otherVis) {
       Ignore(VisAspect1.*);
   }
}
```

The single advice of this aspect intercepts the execution of the main method in order to set *otherVis* to *true* if the appropriate argument has been given at launch. The rule *defaultVis* declares that all visualizations implemented in advices of the aspect *VisAspect1* are chosen by default. But if the boolean *otherVis* has been set to *true* then the second rule is taken into account instead of the default one and the visualizations of *VisAspect2* are used.

Note that some aspect languages have specific statements to enable/disable aspects at runtime. But using rules for this enforces programmers to consider it as an extendable part of the composition strategy.

5.2 Using rules to capture aspect code around join points

Let's imagine now that we want to log the changes of state made by the aspects woven just before a particular type of join points (a call to *foo()* for instance). We can implement this by using a boolean: logging has to be done on all join points where the boolean equals *true*. We just need to specify that this boolean equals *true* only when the advices preceding a call to *foo()* are executed.

```
public aspect LogAspectCode {

  public boolean log=false;

  before() beforeFirst: call(* *.foo(..)) {
      log=true;
  }

  before() beforeLast: call(* *.foo(..)) {
      log=false;
  }

  before() logAdvice: set(* *.*) && if(log) {
      /* Log informations */
  }

  declare rule logBefore:
   advices(beforeFirst, beforeLast) {
      First(beforeFirst);
      Last(beforeLast);
   };
}
```

The aspect shown has three advices: *logAdvices* does the actual logging, *beforeFirst* sets the boolean to *true* and *beforeLast* to *false*. The rule declares that *beforeFirst* has to be the advice that is executed first while *beforeLast* has to be the last one. This ensures that all other advices executed just before a call to *foo()* will be executed between those two and thus in a context where the boolean equals *true* and where the logging is done.

In this example, the rule is declared in the same aspect as the advices it refers to instead of being declared in a composition aspect. This makes sense since the rule only serves the purpose of the aspect and not its integration in the composition.

6. Future Work

First of all we would like to finish our implementation using the extensible ASPECTBENCH COMPILER[1] and then perform a number of case studies on real examples. Last but not least we want to help composers of aspects spot composition problems and understand the semantics of the composition they are making.

7. Conclusion

Composing aspects is not a trivial task. We view this problem as a cross-cutting concern in need of its domain-specific language that enables composers to express compositions easily. We propose a declarative aspect composition language based on only three concepts (*composition rules*, *constraints* and *contexts*) that directly deal with composition.

Composing aspects is known to be problematic since unpredicted aspect interactions may appear and may lead to erroneous weaved programs. This paper focuses on one of these issues: the advice ordering around a join point. It views aspect composition issues as a crosscutting concern that should be handled by a composition aspect. It proposes a domain-specific declarative aspect composition language for composing aspects, and applies it on a number of examples.

Acknowledgments

We would like to thank Maja D'Hondt, Thomas Cleenewerck and the anonymous reviewers for their useful advice and references.

References

[1] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible aspectj compiler. In Peri Tarr, editor, *4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 87–98. ACM Press, 2005.

[2] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA, March 2004. ACM Press.

[3] Andy Kellens, Kris Gybels, Johan Brichau, and Kim Mens. A model-driven pointcut language for more robust pointcuts. In *Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT! 2006)*, 2006.

[4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4.

[5] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, Northeastern University, March 1999.

[6] Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Declarative aspect composition. In *2nd Software-Engineering Properties of Languages and Aspect Technologies Workshop*, 2004.

[7] Davy Suvee, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In ACM Press, editor, *Proceedings of international conference on aspect-oriented software development (AOSD)*, pages 21–29, Boston, USA, March 2003.