Declarative Composable Aspects

Johan Brichau*
Programming Technology Lab - Vrije Universiteit Brussel
johan.brichau@vub.ac.be

September 13, 2000

Abstract

Our position paper argues that a logic programming (i.e. declarative) approach to separation of concerns enables a composition technology that is much easier to handle than standard approaches. Concerns are 'declared' by writing logic facts. Combinations of concerns arise by writing logic rules that are to be seen as compositely declared concerns that consist of more elementary concerns. By using logic programs, the 'concern programmer' only has to focus on what the concerns are about and not about how they will be combined by the weaver. The latter is the job of the inference engine.

1 Introduction

A major problem of the current AOP-technology is that many concerns or aspects of a software application are dependent of each other. Clearly, this conflicts with the purpose of separation of concerns because aspects cannot be developed independently.

For example, if we write two aspects that wrap code around some methods in the component program, their code is not necessarily commutative. Therefore we need to decide which aspect to execute first (e.g. in case of a logging-aspect and a synchronization aspect: logging should also be part of the synchronized code). Because these concerns or aspects have to be integrated into the same program, we need additional information for a successful combination (e.g. in this case, the order of execution).

We can also think of more complex interactions, e.g.: in an application where objects travel freely across a network, we have separated out the concerns of distribution and persistence into aspects. The distribution aspect is concerned with the handling of the object's movement from host to host. The persistence aspect adds a method to every object to save it's state onto disk. However, because they are developed independently, the persistence aspect does not care about the object's current position in the network and hence, does not ensure the object is in a valid location to save its state onto the disk of the server (i.e. we first might want to move the object to the proper server). The handling of this interaction is not a responsibility of any of the two concerns. It clearly is a consequence of their integration, which should be handled by the aspect-programmer at 'aspect-combination time'.

Any technique to support separation of concerns has some sort of 'aspect language' to express how a certain aspect relates to the larger application. The statements of this aspect language are declarations that guide the weaver in his task of integrating the aspect's code into the larger application code. The nature of these

*Research Assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

statements is purely declarative. Therefore, the use of a logic (i.e. declarative) language as a general aspect language seems a straightforward choice. Moreover, because a logic language provides the programmer with a composition mechanism to combine multiple declarations (i.e. logic rules), this approach has the enormous benefit to be a viable technology to easily combine multiple aspects. In particular, if such aspects interact or conflict in some way, the use of a logic rule provides the aspect-programmer with additional control over their composition. As a result, the weaver only interacts with the 'combination aspect' which represents the combination of multiple aspects. The position we are defending in this paper is that a logic language is a viable choice as a general aspect language to allow a systematic study and usage of combinations of multiple, possibly conflicting or interacting, aspects.

2 A Declarative Aspect Language

In our approach, an aspect is a module (or a set) of logic declarations that instruct the weaver to insert code at specific points in the component program. Our aspect-weaver understands the following declarations to describe an aspect[1]:

- `introduceMethod(<classname>,{ <methodcode> })`
  introduce a method <methodcode> into class <classname>
- `introduceVariable(<classname>,{ <variable> })`
  introduce a variable <variable> into class <classname>
- `adviceBeforeMethod(<classname>,<methodname>,{ <advicecode> })`
  add <advicecode> before the method <methodname> in class <classname>
- `adviceAfterMethod(<classname>,<methodname>,{ <advicecode> })`
  add <advicecode> after the method <methodname> in class <classname>

Given a module of declarations, the aspect-weaver generates code for all declarations present in such a module. The weaver accomplishes this by launching logic queries to gather all declarations mentioned above. For example, the following module implements a simple aspect that logs the execution of some methods through the declaration of before- and after-advices [4].

```
adviceBeforeMethod(aClassA,someMethodA,{ System.out.println("Entering aClassA.someMethodA"); })
adviceAfterMethod(aClassA,someMethodA,{ System.out.println(" Leaving aClassA.someMethodA"); })
adviceBeforeMethod(aClassB,someMethodB,{ System.out.println("Entering aClassB.someMethodB"); })
adviceAfterMethod(aClassB,someMethodB,{ System.out.println(" Leaving aClassB.someMethodB"); })
adviceBeforeMethod(aClassC,someMethodC,{ System.out.println("Entering aClassC.someMethodB"); })
adviceAfterMethod(aClassC,someMethodC,{ System.out.println(" Leaving aClassC.someMethodB"); })
```

Clearly these declarations contain too much code duplication. The true power of using a logic programming language as an aspect language is in the use of logic rules [1]. For example, the following module implements the same aspect as above. Note that variables are identifiers starting with a question mark (e.g. `?class`).

```
adviceBeforeMethod(?class,?method,{ System.out.println(``Entering method ?class.?method''); })
     :- log(?class,?method)
adviceAfterMethod(?class,?method,{ System.out.println(``Leaving method ?class.?method''); })
     :- log(?class,?method)
log(aClassA,someMethodA)
log(aClassB,someMethodB)
```

[1] For the sake of the discussion, we have chosen logic declarations that clearly represent a subset of the possibilities of the aspectJ-weaver [4]. Eventually, the aspect language will contain more declarations.

The first two declarations are logic rules that declare 'advices' for each declaration of the form `log(?class,?method)`. It means that the 'advice declarations' are only valid if there are 'log declarations'. In this process, the logic inference engine binds the variables `?class` and `?method` to the constants specified in the 'log declarations'. Through this evaluation process, we obtain the same aspect declarations as before. This example also shows how to separate an aspect's implementation from its use: the first two rules are implemented by the aspect-programmer and the last two declarations are only introduced by the aspect-user when an aspect is to be woven into a particular component program. The real aspect code is located in the first two rules, while the last two declarations 'fill in' the parameters of where the aspect code should be inserted.

3 Composition Modules

The key to combining multiple aspects, and solving possible problematic interactions between them, is the combination of multiple modules using logic rules. This means that we create a new module that contains rules which address the declarations in other modules. Therefore, we introduce a scope-operator: '.' to refer to declarations located in other modules (which are referred to by their name). For example, the following rule simply makes all declarations of `foo(?x)` located in a module named 'anothermodule' available in the module where this rule is defined:

```
foo(?x) :- anothermodule.foo(?x)
```

Hence, given two aspects, aspectA and aspectB (implemented in logic modules with the same name), a simple composition module is depicted in figure 1.

```
introduceMethod(?class,?methodcode) :- aspectA.introduceMethod(?class,?methodcode)
introduceMethod(?class,?methodcode) :- aspectB.introduceMethod(?class,?methodcode)
introduceVariable(?class,?variable) :- aspectA.introduceVariable(?class,?variable)
introduceVariable(?class,?variable) :- aspectB.introduceVariable(?class,?variable)

adviceBeforeMethod(?class,?method,{ ?advicecodeA ?advicecodeB } ) :-
         aspectA.adviceBeforeMethod(?class,?method,?advicecodeA) and
         aspectB.adviceBeforeMethod(?class,?method,?advicecodeB)

adviceAfterMethod(?class,?method,{ ?advicecodeA ?advicecodeB } ) :-
         aspectA.adviceAfterMethod(?class,?method,?advicecodeA) and
         aspectB.adviceAfterMethod(?class,?method,?advicecodeB)

adviceBeforeMethod(?class,?method,{ ?advicecodeA } ) :-
        aspectA.adviceBeforeMethod(?class,?method,?advicecodeA) and
        not(aspectB.adviceBeforeMethod(?class,?method,?advicecodeB))

adviceBeforeMethod(?class,?method,{ ?advicecodeB } ) :-
        not(aspectA.adviceBeforeMethod(?class,?method,?advicecodeA)) and
        aspectB.adviceBeforeMethod(?class,?method,?advicecodeB)

adviceAfterMethod(?class,?method,{ ?advicecodeA } ) :-
        aspectA.adviceAfterMethod(?class,?method,?advicecodeA) and
        not(aspectB.adviceAfterMethod(?class,?method,?advicecodeB))

adviceAfterMethod(?class,?method,{ ?advicecodeB } ) :-
        not(aspectA.adviceAfterMethod(?class,?method,?advicecodeA)) and
        aspectB.adviceAfterMethod(?class,?method,?advicecodeB)
```

Figure 1: A simple composition module

In this module of logic rules, the aspect-programmer declares the combination

of aspectA and aspectB as a new aspect. The overall accomplishment of this 'combination aspect' is that the advices of both aspects are woven in a particular order (i.e. aspectA's advices are followed by aspectB's advices).

The first four rules declare that all introductions of both aspects remain present in this new aspect. The fifth and sixth rule state that all advices of aspectB are preceded by the advices of aspectA (if they act on the same method). To accomplish this, the advice-declarations of aspectA and aspectB are combined into a single advice-declaration in which the code for both advices is appended. The last four rules make sure that the advices of aspectA and aspectB which do not act on the same method, remain present in this 'combination aspect'.

Of course, the combination of multiple aspects can require more invasive changes to the original aspects' code. This will be the case for our example on the combination of distribution and persistence (we mentioned in the introduction). The following rule modifies the code added by the persistence aspect for each object that is affected by the distribution aspect. Of course, the disadvantage of this kind of change is that it requires the aspect-programmer to have a clear understanding of the aspect-code of both aspects.

```
introduceMethod(?class,?method, { bool store() {...move(...); ?storemethodbody } }) :-
       distriAspect.introduceMethod(?class,?movemethod) and
       persistAspect.introduceMethod(?class,{ bool store() ?storemethodbody})
```

Through this declaration, the overall result of the combination aspect is that the `store()` method, introduced by the persistence aspect, should first call a `move()` method. The logic rule specifies that if the class `?class` on which a `store()` method from the persistence aspect is introduced also gets a method introduced from the distribution aspect, then the combined aspect introduces a `store()` method in which we first call the `move()` method. This is done by appending '`...move(...);`' to the methodbody of the original `store()` method (i.e. as it is implemented in the persistence aspect), bound to the `?storemethodbody` variable.

4 Generic Composition Aspects

Eventually, we can think of a predefined layer of composition modules that 'preweave' all possible aspects together before they are injected into the component code. Those predefined modules are driven by extra declarations of the aspect-programmer to guide the composition process. E.g, consider the same 'combination aspect' as in figure 1 in which the 5th and 6th rule are replaced by the following:

```
adviceBeforeMethod(?class,?method,{ ?advicecodeA ?advicecodeB } ) :-
       conf.wrap(?aspectA,?aspectB) and
       ?aspectA.adviceBeforeMethod(?class,?method,?advicecodeA) and
       ?aspectB.adviceBeforeMethod(?class,?method,?advicecodeB)

adviceAfterMethod(?class,?method,{ ?advicecodeA ?advicecodeB } ) :-
       conf.wrap(?aspectA,?aspectB) and
       ?aspectA.adviceAfterMethod(?class,?method,?advicecodeA) and
       ?aspectB.adviceAfterMethod(?class,?method,?advicecodeB)
```

Now, the order in which 'advices' of aspectA and aspectB are wrapped around methods is parameterized. By adding the following declaration in a module named 'conf', the aspect-programmer states that an aspect named 'synchronization' has precedence over an aspect named 'logging':

```
wrap(synchronization,logging)
```

This 'wrap declaration' is then looked up by the advice declarations through the `conf.wrap(?aspectA,?aspectB)` clause, thus binding `?aspectA` to 'synchronization' and `?aspectB` to 'logging'[2].

Clearly, this combination aspect is more generic and can be reused in different applications. Now, the aspect-programmer can take such an off-the-shelf combination aspect and fill in the necessary parameters (i.e. declarations to guide the composition). We can think of building an entire layer of such predefined combination aspects to combine the different aspects in our program. Every combination aspect handles the interactions between several other aspects, which, in turn, can also be combinations of other aspects.

5 Position

In this paper we have argued that the use of a logic programming language as an aspect language allows the composition of multiple aspects in which the aspect-programmer is able to solve possible interactions. This approach allows to write aspects that can be reused in different applications because the interactions with other aspects are only handled at 'composition time'.

References

[1] Kris de Volder, *Aspect Oriented Logic Meta Programming*, Proceedings of Reflection '99, 1999.

[2] Roel Wuyts, *Declarative Reasoning about the Structure of Object-Oriented Systems*, Programming Technology Laboratory, Vrije Universiteit Brussel. Published in Proceedings of TOOLS-USA '98.

[3] Siobhan Clarke, William Harrison, Harold Ossher and Peri Tarr, *Subject-oriented Design: Towards improved allignment of requirements, design and code.* In Proceedings of OOPSLA '99, 1999.

[4] The AspectJ language specification. Xerox Corporation, Palo Alto, http://www.aspectj.org.

[5] G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier and J. Irwin. *Aspect-oriented Programming.* In Proceedings of ECOOP'97, 1997. Springer-Verlag.

[2] This would suggest that modules should be first class entities, however, in case of a variable as module name before the scope operator '.', this variable should not be bound to the module, but to the module name.