Aspect Mining Using Dynamic Analysis

Silvia Breu, Jens Krinke
Lehrstuhl Softwaresysteme
Universität Passau

May 5, 2003

1 Motivation

Concerns express a specific interest in some topic regarding a particular system of interest. *Separation of concerns* (originally invented by Dijkstra) is essential in the software development process: It is an important paradigm in software engineering to cope with the increasing number of special purpose concerns in today's applications. To deal with that increasing complexity, several new approaches like Composition Filters, Hyperslices and last but not least *Aspect-Oriented Programming* [3] (including programming languages like AspectJ) have been proposed. But what about legacy systems, where separation of concerns could only be applied in a restricted way within the object-oriented paradigm? It is possible to find *aspects* and to encapsulate them without changing software behavior, improving maintainability and re-usability, reducing tangled and scattered code. This is illustrated in the following sections.

2 Dynamic Analysis

This short report will present a first approach to dynamically analyze existing Java software to find aspects and crosscutting concerns. For this purpose, different general classes of aspects have been defined:

1. *Outside-Aspects* are patterns where a call of method `a` is always followed by a call of method `b`.
2. *Inside-Aspects* are patterns where a call of method `b` is always inside a call of method `a`.

These two main classes of aspects have been split further, each in two more subclasses:

Outside-Aspect can either be *before* or *after* a specific method call.

Inside-Aspect can either be *first-* or *last-in* a specific method call.

Realize that before- and after-set of aspects in a certain software system need not obligatory be equal, based on the assumption that we are looking for exhaustive sets. This holds for first-in- and last-in-set as well.
However, defining classes of aspects and their structure is not enough to find real aspects. To identify aspects in existing software systems, a possible approach is to generate program traces. The obtained traces are analyzed, using a tool written in Java. It searches for outside and inside aspect candidates with respect to a certain limitation:

```
 1  B.a() {
 2    C.d() {
 3      G.h() {...}
 4      H.g() {...}
 5    }
 6  }
 7  A.b() {...}
 8  B.a() {
 9    C.d() {...}
10  }
11  A.b() {...}
12  B.a() {
13    C.d() {
14      G.h() {...}
15      H.g() {...}
16    }
17    C.c() {...}
18  }
19  F.f() {
20    H.i() {...}
21  }
22  C.c() {...}
23  G.h() {...}
24  H.g() {...}
25  A.b() {...}
26  B.a() {
27    C.d() {...}
     :
     ...
30    F.f() {
31      H.k() {...}
32      H.i() {...}
33    }
34  }
35  D.e() {
36    C.d() {...}
     :
     ...
40    A.b() {...}
41    B.a() {
42      C.d() {...}
43    }
     :
     ...
50    G.h() {...}
51    E.f() {...}
52  }
```

Figure 1: Example trace

The pattern has to exist *always in the same composition.*

This constraint results from the following: A method call to `a` which is identified as being an outside aspect before a method call to `b` has always to be immediately before method calls to `b`, otherwise it is not a recurring pattern (aspect) in the system. The argumentation for outside after and inside first-/last-in aspects is analogous.
Figure 1 shows an example trace which shall be the basis to explain the dynamic analysis procedure and its result in more detail. Analyzing the given trace leads to the following aspect candidates:

- `G.h()` before `H.g()` (3x)
- `G.h()` before `E.f()` (1x)
- `B.a()` after `A.b()` (4x)
- `C.d()` first-in `B.a()` (5x)

- `C.d()` first-in `D.e()` (1x)
- `H.i()` last-in `F.f()` (2x)

The example shows that reversing 'before aspects' [1] does not necessarily lead to 'after aspects': `A.b()` is always followed by `B.a()` but it is not correct to say that before `B.a()` `A.b()` is always called—`B.a()` in line *1* is not preceded by `A.b()`. However, in general, we can say that iff both 'method `a` before method `b`' and 'method `b` after method `a`' applies, the number of occurrences of both patterns is the same—the aspects are *symmetric*.
Additionally, we can see that `C.d()` has no first-in or last-in aspect: Not every appearance of `C.d()` in the example trace has the same structure. In lines *2-5* as well as in lines *13-16* it would have `G.h()` as first-in and `H.g()` as last-in aspect, but in lines *9*, *27*, *36* and *42* it has no first-in and/or last-in aspect. Therefore, neither `G.h()` first-in `C.d()` nor `H.g()` last-in `C.d()` are added to the corresponding result sets. For similar reasons, `G.h()` after `C.c()` has not been detected because this pattern only holds for lines *22/23* but not for line *17*.
Applying a further analysis to the before-/after-/first-in- and last-in-set can now find crosscutting code candidates. Therefore, an additional constraint has to be checked:

> The found pattern has to occur in
> *more than one calling context*.

This means, that a method `a` is a first-in/last-in aspect only if it has this predicate concerning several different methods whereas a method `b` is a before/after aspect if the pattern appears more than once in the trace. Following this definition we can find several candidates for crosscutting concerns in the example:

- `G.h()` is a before aspect of `H.g()` and `E.f()`
- `B.a()` is an after aspect of `A.b()`
- `C.d()` is a first-in aspect of `B.a()` and `D.e()`

3 First Evaluation

For a first evaluation of the presented technique, a simple visualization tool for chopping and slicing (AnChoVis), written in Java, has been traced. The presented analysis of traces of AnChoVis runs revealed that there are method calls for a certain functionality: logging. Those calls always appear with the same pattern at identical places (as first-in and last-in aspect).
Another version of AnChoVis without the logging functionality scattered throughout the code has been created. The logging functionality has been implemented as simple aspect, written in AspectJ, and woven to the program. That resulted in a program version with the same behavior as before but with better understandability and maintainability of the code.

[1] In the remaining of the report only 'before', 'after', 'first-in' and 'last-in' are used to characterize the aspects as the nomenclature is clear.

4 Future Work

Up to now, the analysis presented in this short report is not fully assessed and explored but first evaluations are promising.
An open question is whether and how to merge identical first-in and last-in aspects. The current analysis does not provide any information if the aspect is one and the same call inside another method or if there are two calls of one and the same method with anything (statements, other method calls) in between.
More precise information could be gained by not just tracing method calls but also statements. However, this would make the analysis more complex—a tradeoff between higher gain and precision on the one hand, reduced speed on the other hand.
Another question is whether it is worthwhile finding more complex aspects: for example summarizing chains of aspects like `a` before `b`, `b` before `c`, and `c` before `d` to one single aspect `abc` before `d`. Solutions for these points may give additional information to the question how simple it is to encapsulate located aspects properly and correctly.
Anyhow, there are even more open questions which have to be explored:

- Do case studies with bigger software systems confirm the results we have so far?
- Are there other aspects except the already well-known aspects like logging etc.?

5 Related Work

There are two other approaches known to the author which present techniques to identify aspects. The work of Hannemann and Kiczales [2] is a query-based Aspect Mining Tool (AMT): The query specifies a type or a regular expression; every line of an analyzed system is checked against the query and highlighted if it matches. In the second approach, Aspect Browser [1], crosscuts are identified with textual-pattern matching and highlighted in the program files' windows with a specific color.

References

[1] William G. Griswold, Y. Kato, and J. J. Yuan. Aspect Browser: Tool Support for Managing Dispersed Aspects. Technical Report CS99-0640, Department of Computer Science and Engineering, University of California, San Diego, Dezember 1999.

[2] Jan Hannemann and Gregor Kiczales. Overcoming the Prevalent Decomposition of Legacy Code. In *Workshop on Advanced Separation of Concerns*, 2001.

[3] Georg Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.