Refactoring and Code Generation Tools for AspectJ

Paulo Borba* Sérgio Soares†
Informatics Center
Federal University of Pernambuco

October 29, 2002

1 Introduction

Code generation and refactoring tools have been quite useful for developing object-oriented applications [4, 2]. They increase development productivity by automating tedious, repetitive, and error-prone tasks. By reducing the number of programming errors, they also help to improve software quality.

Based on our experience developing AspectJ [5] applications, we believe that aspect-aware code generation and refactoring tools can bring similar benefits for the development of aspect-oriented applications as well. Although aspect-oriented languages such as AspectJ provide some of the power of metaprogramming constructs, we think this is not enough for dispensing with code manipulation tools.

In fact, code generation tools could generate part of the implementation of specific AspectJ patterns. For instance, we have noticed [6, 7] that the implementation of persistence and distribution aspects in several applications might follow the same structure, which could be automatically generated. The generation tools could also be used to instantiate specific AspectJ frameworks. This sometimes involves tedious coding that can only be avoided by generation tools or metaprogramming constructs more powerful than the ones currently supported by AspectJ.

Similarly, AspectJ-aware refactoring tools could be used for improving and maintaining existing AspectJ code. Again, this would be necessary when implementing distribution concerns as suggested elsewhere [6], since for every newly introduced remote method we should introduce an associated advice following an specific pattern. The advice should also be removed when the method is removed. User defined refactorings could simultaneously change both the pure Java and the AspectJ code, keeping the code consistent.

AspectJ refactoring tools could also help developers to restructure Java code in order to separately implement different concerns using AspectJ features. This is necessary, for example, when migrating Java applications to AspectJ. A well defined set of refactorings can guide the developer in this task and guarantee that the resulting AspectJ system preserves the behavior of the original system. Sometimes refactorings are also useful for exposing join points that should be intercepted by the aspect code.

*Supported in part by CNPq, grant 521994/96–9. Electronic mail: phmb@cin.ufpe.br. Telephone: +55 81 3271 8430, extension 4323. Fax: +55 81 3271 8438.

†Supported by CAPES. Also affiliated to Catholic University of Pernambuco. Electronic mail: scbs@cin.ufpe.br.

2 AspectJ Transformations

Our approach for developing refactoring and code generation tools for AspectJ is to consider program transformation as a unifying concept for code generation and refactoring [3]. A refactoring comprises several behavior preserving changes on the program, but does not add new functionalities. A generator, on the other hand, introduces new functionalities. With such unifying view, AspectJ transformations may create new code and modify existing ones as long as the semantics of the original program is preserved. For instance, the following simple program equivalence can be used to introduce a pointcut declaration, specifying a set of join points:

```
aspect A {}
```
$=$
```
aspect A {
  pointcut p(params) :
    events;
}
```

It can also be used to remove a useless pointcut declaration, when applied as a transformation from right to left.

In the example, the details of the introduced or removed pointcut should be given by instantiating the transformation variables, which appear in italics. However we could provide more information in the transformation if necessary, indicating the exact template of the code to be manipulated or generated. Moreover, besides variables, we could use more powerful metaprogramming constructs to specify the transformations [1].

We can similarly define transformations that manipulate both aspects and classes. This is the main point to consider when adapting our previous work [3, 1] on developing similar tools for Java. In fact, existing Java refactorings [4] are not valid when developing with AspectJ. They have to be adapted to consider the impact that the modifications on the Java code have on the aspects code. For example, when modifying the name of an intercepted field we have to modify the class that declares the field and the aspects that intercept accesses to that field.

Another example where adaptations are necessary is when inlining a method; we have to check if the call to that method is not intercepted by some advice, otherwise the refactoring would yield an invalid program. We could alternatively define an AspectJ-aware inline method refactoring in such a way that applying it to the code sketched on the left would yield the code sketched on the right:

```
class C {...
  int d() {
    return x*2;
  }
  int m(int i) {
    return d()*i;
  }
}
aspect A {...
  before:
    call(int C.d()) {
    System.out.print(...);
  }
}
```
$=$
```
class C {...
  int m(int i) {
    System.out.print(...);
    return (x*2)*i;
  }
}
aspect A {...}
```

where not only the call to `d` was inlined and the method declaration removed. Notice that the advice that intercepts calls to `d` was also "inlined" and its declaration removed too.

3 Coding Wizards

The transformations specify how the code should be refactored or generated, but they should not be visible to the programmer that uses the refactoring and generation tools for supporting coding activities. In order to accomplish that, we intend to provide coding wizards, which encapsulate transformations and the associated graphical user interfaces used to configure and provide parameters for these transformations. The programmers can then work directly with the wizards, not with the transformations. The wizards can be activated through IDEs menu options [3].

The wizards are defined by assembling transformations and graphical components reused or constructed by a tool customizer, which should also be responsible for defining the wizards. The components are Java Beans that comply to a specific interface. Transformations should be written in a language that extends AspectJ with metaprogramming constructs [1]. Such extension allows the tool customizer to program generic templates, and by using them he specifies how AspectJ code should be generated or modified, as illustrated before. This flexibility for creating and composing wizards based on an open architecture contrasts with typical code generation and refactoring tools, which support only a fixed set of built-in generators or refactorings.

4 Conclusions

Based on our practical experience with AspectJ, we believe that code generation and refactoring tools can play an important role in the development of aspect-oriented applications. This was noticed in several occasions during the development of two AspectJ applications: a health watcher system [6], having persistence and distribution aspects, and a J2ME dictionary application, having several aspects for adapting the application to different situations.

Corresponding Java tools can be adapted to AspectJ, and integrated to IDEs in a similar way. The major difficulty involved is that manipulations of the pure Java code have to consider the impact they might have on the aspects code, and vice-versa. In our approach, this means that the specified transformations should consider both the pure Java code and the associated aspects code. Special code analysis, such as the ones implemented in existing IDE plug-ins for AspectJ, should be implemented to support that.

References

[1] Fernando Castor and Paulo Borba. A language for specifying Java transformations. In *V Brazilian Symposium on Programmig Languages*, pages 236–251, Curitiba, Brazil, 23th–25th May 2001.

[2] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison–Wesley, 2000.

[3] Marcelo d'Amorim, Clóvis Nogueira, Gustavo Santos, Adeline Souza, and Paulo Borba. Integrating Code Generation and Refactoring. In *Workshop on Generative Programming, ECOOP02*, Malaga, Spain, June 2002. Springer Verlag.

[4] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[5] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[6] Tiago Massoni, Augusto Sampaio, and Paulo Borba. Progressive Implementation of Aspects. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems — OOPSLA'01*, Tampa Bay, USA, 14th-18th October 2001.

[7] Sérgio Soares and Paulo Borba. PaDA: A Pattern for Distribution Aspects. In *Second Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Itaipava, Rio de Janeiro, Brazil, August 2002.