Logic Meta Programming as a Tool for Separation of Concerns

Kris De Volder, Tom Tourwé *, Johan Brichau
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
E-mail: {kdvolder,tom.tourwe,jbrichau}@vub.ac.be
WWW: http://prog.vub.ac.be/

April 4, 2000

Abstract

An aspect system which only allows aspect code to be parameterized by the affected join point itself is not sufficiently expressive to capture some relatively simple aspects adequately. It is our position that a good AOP approach should be able to deal with aspect code that is parameterized by a relationship between code elements. We present a simple example to illustrate that code which needs to be inserted at a particular join point depends often not only on this join point itself, but also on a 'distantly related' code element.

We argue that logic meta programming is an interesting technology for AOP implementation, amongst other things because it serves as an excellent metaphor to express such 'join point relationships' and use them to drive code generation.

1 Introduction

An AOP system [KLM+97], such as AspectJ, which allows aspect code to be parameterized only by the affected join point itself is not sufficiently expressive to capture even some relatively simple aspects adequately. It is our position that a good AOP approach should be able to deal with aspect code that is parameterized by a relationship between code elements.

This paper presents a simple example to illustrate that often, code which needs to be inserted at a particular join point depends not only on this join point itself, but also on (one or more) 'distantly related' code elements. The example we present is the implementation of a *Visitor* design pattern [GHJV95]. To adequately express the common implementation structure underlying all visitors as a reusable aspect, we need to be able to capture the relationship between a visitor class and the classes it visits.

We illustrate the usefulness of logic meta programming (LMP) as a means to express relationships in a relatively straightforward way, and to use these relationships explicitly in parameterizing aspect code. Thus, aspects can be programmed in a significantly more generic way.

2 The Example Problem

The Visitor design pattern is meant to achieve more separation of concerns between how to traverse an object structure and what has to be done during the traversal process. To achieve

*Author financed with a doctoral grant from the Flemish Institute for the Improvement of the Scientific-Technological Research in Industry (IWT).

this separation, the Visitor groups all of the methods that actually do any work in a separate visitor class. However, the `accept` methods, that provide the structural backbone for the Visitor, still have to be scattered throughout the visited classes and their implementation becomes a cross-cutting concern of its own.

Expressing these structural changes to the visited classes as an aspect in AspectJ is relatively straightforward, for example:

```
abstract class Visitor {
   introduction AbstractTree {
      public abstract Object accept(Visitor v);
   }
   introduction Node {
      public Object accept(Visitor v) { return v.visitNode(this); }
   }
   introduction Leaf {
      public Object accept(Visitor v) { return v.visitLeaf(this); }
   }

   public abstract Object visitNode(Node node);

   public abstract Object visitLeaf(Leaf leaf);
}
```

Contrary to the plain Java implementation of a Visitor, this one achieves clean separation of concerns and keeps all of the Visitor's implementation details together. It uses AspectJ's `introduction` declarations to insert the necessary `accept` methods into the visited classes. Subsequently, concrete visitors for the `AbstractTree` hierarchy can be implemented as subclasses of this abstract `Visitor` class.

However, this solution is not yet entirely satisfactory. In fact there is a lot of manually repeated code in this implementation: the `accept` methods for `Leaf` and `Node` look very much alike, and would have been repeated for other visited classes if there were more. What is worse, the tedious work of implementing an abstract visitor will have to be repeated if a visitor for another set of classes is needed.

What we would like is to factor out the commonalities in the implementation of the `accept` method in this visitor, and by doing so, at the same time make the visitor more independent of the visited class structure.

3 Generic Visitors in AspectJ?

Since recently, AspectJ has been extended with some ideas from adaptive programming [Lie96, LLM99] and provides a way to decouple an aspect from its specific application context, by using abstract crosscut declarations. This allows to specify the join points (in this example the `Node, AbstractTree` and `Leaf` classes) separately from the actual aspect implementation. This makes the aspect more reusable.

Regrettably, this mechanism falls short of achieving a clean decoupling of the Visitor from its visited classes for a number of reasons. Most importantly, the aspect code can only be parameterized by the join point itself by using `thisJoinPoint`. This creates problems in the `visitNode` and `visitLeaf` methods, because although they are to be introduced into the Visitor, their definition relates also to the visited classes, both in their name and in their type signature. AspectJ does not provide a way to parameterize the definition of the visit methods with the visited nodes. Note that visitLeaf and visitNode in the example were not introduced using an introduction declaration, but even if they were, this would not help because `thisJoinPoint` would just refer to the Visitor class. The only way to correctly define

the methods is in a non generic way, on a case by case basis, referring explicitly to a particular visited class in each individual visit method. The fundamental problem is that the methods have to be introduced into one class, while at the same time, their precise definition depends also on another class.

In the next section we will show how to achieve a clean generic visitor implementation using an LMP approach. We will explain how it depends on the ability to express relationships between code elements and use these to generate aspect code.

4 Logic Meta Programming

The concept of logic meta programming [DV98, DV99] is straightforward: a base program is represented indirectly by means of a set of logic propositions. Since a logic program can be thought of as representing the set of logic propositions that can be proven from its facts and rules, these facts in turn can be thought of as indirectly representing a base-language program. The full power of the logic paradigm may thus be used to describe base-language programs indirectly. The relationship between the base program and its logic representation is made concrete under the form of a code generator: a program that queries the logic repository and outputs source code in the base language.

The mapping scheme between logic representation and base program may vary and determines what kind of information is reified and accessible to meta programs.

For example, consider the mapping which represents Java class declarations by means of facts, which state that the class has certain methods, instance variables or constructors.

The presence of a variable declaration in a class is represented by a fact of the form:

```
var(?Class,?VarType,?VarName).
```

A method declaration is asserted by a fact of the following form:

```
method(?Class,?ReturnType,?MethodName,?ArgList,{...method body...}).
```

Below is an example Java class declaration and its corresponding representation as a set of propositions.

```
class Stack {
  int pos = 0 ;
  ...
  public Object peek ( )  {
    return contents[pos];  }
  public Object pop ( )  {
    return contents[--pos];  }
  ... }
```

```
class("Stack").
var("Stack","int","pos").
...
method("Stack","Object","peek",[],
  {return...}).
method("Stack","Object","pop",[],
  {return...}).
...
```

4.1 Join 'relations' as logic programs

We propose to define join points in terms of logic programs. A join point will be characterized by a set of solutions to a logic query. As such, the join points are determined by the facts and rules in the database matching the query. The most straightforward way of identifying a set of join points is by explicitly enumerating them one by one by asserting facts. Of course, we can also use rules, which describe the properties the join points have in common.

Using this approach for defining sets of join points corresponds relatively closely to how AspectJ uses a pattern language to define 'crosscuts' as sets of join points. However, the logic language is significantly more expressive. One important fundamental difference is the ability to express not only sets of elements, but also relationships between elements. To emphasize

this difference between using sets or relations, we will talk about *join relations* rather than join points from here on down. Note that join relations are a natural generalization of sets of join points, since sets are merely 'unary' relations.

We have identified three join relations important for the Visitor example:

`visitor(?Visitor)` defines the set of all (abstract) visitor classes.

`rootVisitedNode(?Visitor, ?Tree)` defines the relation that determines, for each visitor, the root class of the hierarchy visited by it.

`visitedNode(?Visitor, ?Node)` defines the relation between each visitor class and the concrete classes it visits.

The generic Visitor implementation can be defined completely in terms of these three join relations. Thus, we make a clear separation between the clauses implementing the generic Visitor, and the clauses which concretely define the join relations needed to actually instantiate it. This functions more or less like abstract crosscut declarations in AspectJ. The concrete definitions of the join relations are provided separately from the aspect implementation.

The generic implementation of the visitor can be found below. Note how every part of the implementation is parameterized with logic variables, which are bound to concrete values by a query to one of the three join relations.

```
/* Declare the Visitor class (empty class declaration) */
class(?Visitor)
   :- visitor(?Visitor).

/* Declare an abstract accept method for the root of the Visited class
 * hierarchy */
abstractmethod(?Tree,"Object","accept",[ [?Visitor,"v"] ])
   :- rootVisitedNode(?Visitor,?Tree).

/* For every visitedNode called ?Node, declare the following... */
/* 1) an abstract visit<?Node> method in the Visitor */
abstractmethod(?Visitor,"Object",visit<?Node>,[ [?Node, "x"] ])
   :- visitedNode(?Visitor,?Node).
/* 2) an accept method in the visitedNode */
method(?Node,"Object","accept",[[?Visitor,"v"]],{ return v.visit<?Node>(this); })
   :- visitedNode(?Visitor,?Node).
```

Note that in this implementation of the Visitor aspect, the code duplication that occurred due to the similarity between the `Node` and `Leaf` classes has been factored out. Internal nodes and leaf nodes are treated alike and are both captured by the same join relation, `visitedNode`.

4.2 Instantiating the Visitor Aspect

The Visitor aspect implementation given above is defined abstractly in terms of three abstract join relations. In order to use this visitor and instantiate it in a concrete situation, we have to provide a concrete specification for the join relations. As explained before, we do this by implementing them using rules and facts. We thus have the possibility to write sophisticated logic programs in order to determine join relations. Of course, it is also possible to just enumerate the join relation's elements one by one simply using facts, or use a combination of facts and simple rules.

For our example, we need to express that the `AbstractTreeVisitor` is the visitor class, while the `AbstractTree` class is the root of the visited classes hierarchy and the `Node` and `Leaf` classes are the visited nodes. This can be done as follows:

```
#include "VisitorAspect.rub"
#include "AbstractTreeHierarchy.java"
visitor(AbstractTreeVisitor).
rootVisitedNode(AbstractTreeVisitor,AbstractTree).
visitedNode(AbstractTreeVisitor,Node).
visitedNode(AbstractTreeVisitor,Leaf).
```

Instead of explicitly enumerating all elements of a join relation one by one, we can characterize it by means of a rule, or in general, any logic program. For example, we can define the `visitedNode`s by stating that they are all the concrete subclasses of a given `rootVisitedNode`:

```
visitedNode(?Visitor,?Node) :-
    rootVisitedNode(?Visitor,?AbstractNode),
    subclass(?AbstractNode, ?Node),
    isConcrete(?Node)
```

Instantiating the Visitor pattern now boils down to just providing the appropriate `visitor` and `rootVisitedNode` facts, from which all other information can be derived automatically.

5 Conclusion

In this paper, we presented a simple example showing that code implementing a certain aspect often does not depend on a single join point, but on the relation between different code elements. We argued that aspect systems that do not allow such relationships to be made explicit, fail in defining truly reusable aspects that depend on these relationships. We therefore propose the concept of join relations, which is a natural generalization of sets of joint points. Using join relations rather than self contained sets allows more flexible parameterization of the aspect code and makes the aspects more reusable.

To illustrate the usefulness of characterizing join-points by means of relations rather then sets, we used a logic meta programming approach. Defining and using relations is the basis of the logic paradigm and therefore expressing and using join relations is intuitive and straightforward in the LMP approach. However, the idea of join relations has merit of its own and is independent of using LMP. We believe it should be possible to express some kind of join relationships in any AOP approach, in order to increase its ability to express generic aspects.

References

[DV98] Kris De Volder. *Type-Oriented Logic Meta Programming.* PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[DV99] Kris De Volder. Aspect-oriented logic meta programming. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer Verlag, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series, 1995.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer.

[Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software — The Demeter Method with Propagation Patterns.* PWS Publishing Company, 1996.

[LLM99] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.