Aspect-Oriented Programming with Aspectual Methods

Karl Lieberherr [1] Doug Orleans [1] Johan Ovlinger [1]

[1] College of Computer Science
Northeastern University
Boston, Massachusetts 02115-5000
Email: {lieber,dougo,johan}@ccs.neu.edu

February 15, 2001

Abstract

We use Gregor Kiczales' definition of an aspect: a modular unit of crosscutting implementation and present a useful kind of aspect-oriented programming in Java. By focussing on a limited, but commonly occuring set of aspects, we are able to provide the Java library DJ that allows us to write aspectual methods whose ad-hoc implementation cuts across several classes. An aspectual method is evaluated in the context of a class graph and it invokes a traversal-style method that traverses through objects following has-a links guided by a traversal specification. We present the key elements of the library and we connect aspectual methods with the AspectJ terminology. Several examples demonstrate the advantages and limitations of aspectual methods.

We hope that aspect-oriented programming will be better understood if it can be practiced, even in limited form, directly in Java. This paper shows one way how to achieve this through the DJ library.

1 Introduction

Just like object-oriented programming takes on several forms, from the pure everything-is-an-object philosophy of Smalltalk to the mixed approach of C++, aspect-oriented programming has several guises.

The aspect-oriented programming movement has several roots, such as reflection [12], open implementation [13], composition filters [1], subject-oriented programming [10], multi-dimensional separation of concerns [32], the GenVoca work on large-grain refinements [31] and the Demeter work [6] on separating behavioral and structural concerns but also on separating "systemic" concerns like synchronization [25, 21] and data transfer [22, 23, 21].

This paper describes how DJ [26], part of the Demeter family of programming tools since 1997, can be used to support a limited form of aspect oriented programming. Our focus, inherited from the Demeter project, is on controlling the coupling between structural and behavioral concerns of a program. Previously, we have explored how the coupling can be controlled in extensions of Java [28] and C++ [16, 17] and in other programming languages like CLOS [2] and Perl [11], while this paper focuses on the current implementation as a stand-alone Java runtime library and the restrictions and capabilities this affords us.

Compile time transformations allow a seamless integration of aspect-oriented features into the programming model, but do so at a cost of convenience – often the programmer is forced to "buy-in" to a complete programming environment, or at the very least to re-process any existing code to work with the new features. As DJ is implemented as a Java package requiring no compile time support, it can be trivially added (simply by importing it and programming against its API) to existing projects where necessary, with absolutely no effect to parts of the project not programmed against the DJ API. However, this convenience comes at a cost of capability; we sacrifice capability and general applicability to achieve the goal of low adoption cost.

Throughout this paper, we eschew the terminology we use in earlier work and instead adopt the AspectJ [33] terminology which is rapidly becoming accepted as standard in the field.

1.1 A Simple Example

It is easier to understand the capabilities and limitations of DJ with an example in mind. Figure 1 is a small method written in Java using the DJ library. The purpose of the code is to sum all the Salary-objects reachable by has-a relationships from a Company-object. The example illustrates the Demeter style programming that is used with DJ.

It works as follows: the ClassGraph argument to `sumSalaries` is an object which represents the program's class structure. A class graph (for examples, see Figure 5 and 8) is a simple form of UML [3] class diagram that describes has-a and is-a relationships between classes. The class structure is computed in ClassGraph's constructor using reflection. [1]

It is used by the traverse method as a context in which to interpret traversal specifications. The traverse method starts in a given object (`this` in this case) and traverses specified paths (`''from Company to Salary''`) executing any applicable visitor methods along the way (in this case upon `start`, reaching Salary, and to calculate the `getReturnValue`).

The separation of traversal specification (where to go), behavior (what to do), and class graph structure (context to evaluate in), allows the method to be reused unchanged in a set of programs. The traversal specification can be seen as a predicate expressing path existence constraints: in the class graph there must exist two classes: Company and Salary, and there must exist at least one path in the class graph from Company to Salary. We don't care how many classes are between Company and Salary: it could be a company with a big organizational structure (divisions, departments, work groups, etc.) with many classes between or a company with a small organizational structure (only work groups) with few classes between.

Method sumSalaries is an example of an aspectual method. An aspectual method is evaluated in the context of a class graph object (or another DJ object that was built from a class graph object) and it invokes method traverse (or one of its sisters, e.g., fetch) inside the method. Those methods perform a traversal action guided by a traversal specification.

Method sumSalaries is an aspect in the sense of [14]: it is a modular unit of crosscutting implementation. It is a modular unit because it is a Java method. It is crosscutting, because using an ad-hoc implementation of this method would result in scattering of the information in this method to several classes. This is the case if the method is written in good object-oriented style following the Law of Demeter [18]. By violating the Law of Demeter we could write the traversal entirely in class Company but this would lead to brittle [2]. code by putting much of the information in the class graph just into method sumSalaries(). However, by using the traversal specifications of DJ , we are able to avoid encoding such assumptions in our method. Because method sumSalaries qualifies as an aspect, we call it an aspectual method.

1.2 Controling Scattering and Tangling

All but the simplest programs contain multiple concerns. By a concern we mean an issue the programmer is concerned about, e.g., how do I express behavior A, where do I have to go in this object, how do I synchronize this program. In normal programming practices, these concerns are distributed (scattered) throughout the program and intermingled with each other (tangled). Aspect-Oriented programming in general aims to separate these various concerns into localized aspect definitions. This is held to be beneficial for program maintenance; the programmer works with neat parcels of code that are automatically transformed to the scattered and tangled representation required for program execution.

DJ allows the programmer to write code in plain Java that maintains localized definition but has distributed ad-hoc implementation. Because it is a runtime library, it is limited in its power. While both DJ

[1] ClassGraph-objects can be manipulated after instantiation to fine-tune the behavior of aspectual methods. It is not uncommon to store ClassGraph-objects in `static` variables to avoid passing them explicitly to all aspectual methods.

[2] By brittle, we mean code that has a propensity to require substantial hunting and fixing of broken assumptions after a small change to program structure. The Law of Demeter points out that by only making local assumptions about program structure, we are able to localise the effects of program modifications.

```
class Company {
  Double sumSalaries(edu.neu.ccs.demeter.dj.ClassGraph cg){
    String s = "from Company to Salary";
    edu.neu.ccs.demeter.dj.Visitor v = new edu.neu.ccs.demeter.dj.Visitor(){
      private double sum;
      public void start(){sum = 0.0};
      public void before(Salary host){
        sum += host.getValue(); }
      public Object getReturnValue() {return new Double(sum);}
    };
    return (Double) cg.traverse(this, s, v);
  }
}
```

Figure 1: *Simple Aspectual Method.*

and AspectJ are able to express behavior that needs to be spread across several classes, DJ is not able to modify existing methods.

1.3 Loose Coupling between Behavior and Structure

A key insight in the Demeter project is the importance of a loose coupling between structure and behavior. It is the loose coupling provided by DJ which allows us to localize the distributed implementation of our behavior without adversely affecting maintainance. DJ allows us to write distributed behavior by specifying the structural locations where we need to add behavior in a very flexible manner.

It would be possible to write a localized method that performed the same task as an aspectual method written in DJ , but it would be at the price of maintainability – the very goal of aspect oriented programming. This is because without DJ we would be forced to explicitly write code to visit the points of interest in the object graph. Any time the class structure changes, we need to revisit that code to make sure its assumptions about the structure have not been broken.

DJ allows us to encode the important assumptions that an aspectual method makes about the class graph in a format that can be checked at runtime. Effectively, we evaluate all constraints in the context of the running application's class graph to dynamically discover how the aspectual method's behavior should be executed. No code generation takes place with DJ ; a path set is interpreted and traversed by a single method that takes the reified class structure as input rather than being implemented by a bunch of hard-coded methods that live directly on the class structure.

1.4 Connection to Aspects and AspectJ

Because we do aspect-oriented programming for a limited set of aspects in Java and because AspectJ has already developed a terminology for general purpose aspect-oriented programming in a small extension to Java, we recall key definitions of the AspectJ language [14]: Join points are principled points in the execution of the program; pointcuts are a means of referring to collections of join points and certain values at those join points; advice is a method-like construct that can be attached to pointcuts; and aspects are modular units of crosscutting implementation, comprised of pointcuts, advice, and ordinary Java member declarations.

Method sumSalaries() contains a definition of a collection of join points and it contains a definition of advice for those join points. The join points are defined by a traversal specification and the advice by a visitor object.

The join points are defined in an unusual way: Instead of referring to points in the execution of a program given to us, we first define a traversal in terms of a class graph given to us and then we consider the join points defined by this traversal. In other words, we use a version of AOP where the pointcut definitions indirectly define a collection of join points based on information in the class graph.

The advice is also defined in an unusal way: in the Java method sumSalaries(...) different join points receive different advice while in AspectJ all join points in a pointcut get the same advice. With DJ , some points in a point cut get no advice at all if they are not mentioned in the visitor object. For further discussion, see section 4.1.

The rest of this paper is organized as follows: In section 2 we explain the concepts behind aspectual methods in more detail and introduce a simple explanation of the semantics of traversal specifications. In section 3 we present a more interesting example of an aspectual method. This aspectual method implements the operation: report all used entities that are not defined. We conclude with additional features of DJ and show their analog in AspectJ.

2 Concepts behind Aspectual Methods

An aspectual method provides one way to implement the Demeter style of programming in Java. We recall that the Demeter style is characterized by parameterizing a program by a class graph, by using class graph predicates to define traversals through objects of the class graph and to define traversal enhancements (additional code that is executed during the traversal).

2.1 ClassGraph, Strategy and Visitor

We use the following three key concepts to express aspectual methods: ClassGraph, Strategy and Visitor. To each concept corresponds one Java class in the DJ package. So programming with aspectual methods amounts to instantiating the classes of the DJ package in the context of a method (either inside the method or passing them as a parameter).

A ClassGraph-object is a graph whose nodes are classes and whose edges are is-a and has-a relationships between classes. Class ClassGraph provides methods to create and maintain a class graph. The simplest way to create a ClassGraph-object is to call the constructor ClassGraph() without arguments which will create the class graph using Java reflection by taking all classes in the default package.

Before we explain the interface of ClassGraph, we need class Strategy. A Strategy-object describes a traversal specification. DJ follows the rule that wherever a Strategy-object may be used, we can instead use a String-object. We have applied this rule in Fig. 1 and we will apply it again in the following: we describe traversal specifications as strings. A traversal specification may be applied to both a ClassGraph-object and a Java object. From the point of view of a ClassGraph-object, a traversal specification is a subgraph of the transitive closure of the ClassGraph-object if it is applied to a class graph it selects a subset of the paths in the class graph. If applied to a Java object, a traversal specification defines a subgraph of the object graph representing the Java object.

Class ClassGraph supports the following two methods called traverse and fetch. traverse we have already seen used in the introduction: it provides the basic building block for programming in Demeter style. Method fetch retrieves one object along one path in a Java object (Fig 4 shows several examples of fetch).

The following table summarizes class graphs and traversal specification. It might be surprising that traversal specifications are classified as graphs with nodes and edges. Indeed, from A to B is a two node graph with one edge. In this paper we only use traversal specifications whose graph is a straight line, although DJ supports traversal specifications that are series-parallel graphs [30] and even general graphs [20].

A class graph defines a set of objects (namely the objects that follow the rules of the class graph) and a traversal specification defines a set of paths in the objects belonging to a class graph that satisfies the traversal specification viewed as a predicate. For an example, see section 2.3.

Graph	Nodes	Edges
ClassGraph	Classes	is-a and has-a relationships
Strategy	Classes	has-a paths in object graph

2.2 Traversal Semantics

The traversal concern deals with how to traverse through objects. A traversal is a function that maps each object graph rooted at object o to a subgraph rooted at o. This subgraph may then be traversed using one of several techniques, e.g. depth-first traversal that we use with DJ . We use a language tailored to the traversal concern and sentences in this language we express as strings in Java programs. The simplest expression in our language is of the form "from A to B". This means: given an object of class A (an A-object), select a subgraph of the A-object that contains all B-objects. The subgraph of the A-object is not a minimum subgraph, but it is minimum relative to the information in the class graph without look-ahead in the object graph. This is explained further below. XPath [5], the traversal language of XML [4] has a similar construct (roughly A//B) that has the meaning of returning all B-objects reachable from the A-object. In both cases it does not matter how many objects are contained between the A-object and the B-object.

The traversal relies on the following primitive: Given an object graph of some class graph, and a root object o of class A from which the traversal from A to B starts, we need to decide which objects to visit from o, i.e., we need to compute first(o), the set of edges that we need to traverse from o. Our goal is to make the traversal efficient in the sense that we don't want to look ahead in the object graph whether going to an object in first(o) will eventually lead us to a B-object. We only look ahead in the class graph where we have complete information about the shape of objects. So first(o) will contain all those edges after which, according to the class graph information, there is still a possibility of reaching a B-object. We accept that for this particular object some of the traversals might result in a dead end.

2.3 Traversal Example

Consider the class graph in Figure 5 that we will also use later and consider the traversal: from EquationSystem to Numerical and the EquationSystem-object es shown in Fig. 2 using the UML object diagram notation. When we start the traversal at object es, we don't know yet that this particular object does not contain a Numerical-object. The class graph in Figure 5 tells us that an Equation_List-object might contain a Numerical-object and therefore we traverse to object els. The class graph tells us that an equation might contain a Numerical-object and therefore we traverse to object e1. The class graph in Figure 5 tells us that traversing through the lhs link would be futile because we can never find a Numerical-object in there. The class graph also tells us that traversing through the rhs link might lead us to a Numerical-object but for this particular object the link contains a Variable-object and a Variable-object never contains a Numerical-object. Therefore the traversal terminates prematurely at object e1.

2.4 Traversals with Edge Constraints

A traversal specification may be decorated with additional negative constraints attached to the edges. An example of a negative constraint is a bypassing {X,Y} constraint. It might be used in the context of "from A bypassing {X,Y} to B" which means that we don't want to visit any X- or Y-objects as we traverse from an A-object to a B-object along has-a relationships. The traversal language of XPath supports a similar capability.

The traversal specifications that we use in this paper are only simple straight line specifications:

```
from A to B                 (one edge, two nodes)
from A through B to C       (two edges, three nodes)
from A bypassing B to C     (one edge with constraint, two nodes)
```

They cover many interesting cases. Traversal specification languages and their semantics are described in detail in [30, 20].

3 Example: Aspectual Checking

We demonstrate the use of aspectual methods on a larger example and show their reusability. The example involves two collaborating aspectual methods.

Fig. Eq4

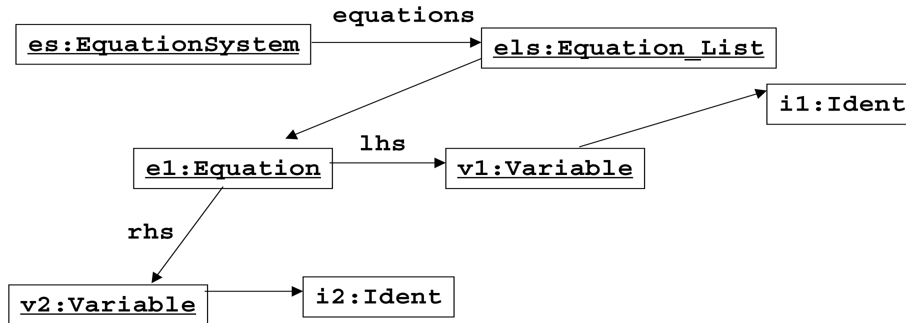

Figure 2: EquationSystem-*object to illustrate traversal. The class graph for class* EquationSystem *is in Fig. 5.*

3.1 Equation System Example

Consider an equation system that contains equations with a left-hand-side (LHS) and a right-hand-side (RHS). We use a mathematical notation whose structure is defined by the class graph in Fig. 5.

```
X1 = (X2 + X3)
X2 = (X5 + (X3 * X1))
X3 = 7

Fig. F1
```

The LHS is a variable and the RHS is an expression involving variables. We say a variable is defined, if it appears on the LHS of an equation. In Fig. F1, variables X1,X2 and X3 are defined. We say a variable is used, if it appears on the RHS of an equation. In Fig. F1, variables X2, X3 and X5 are used.

The purpose of our Java program is to determine all used variables that are not defined. In Fig. F1, variable X5 is used but not defined.

We want our program to be generic and applicable to different equation systems and even different grammar formalisms.

3.2 Grammar Example

Consider a grammar that contains productions with a left-hand-side and a right-hand-side. We use a simple form of EBNF (Extended Backus Naur Form) notation whose structure is defined by the class graph in Fig. 8.

```
X1 -> X2 X3.
```

```
X2 -> X5.
X3 -> Ident.
```

Fig. F2

The LHS is a nonterminal and the RHS is an expression involving nonterminals. We say a nonterminal is defined, if it appears on the LHS of a production. In Fig. F2, nonterminals X1,X2 and X3 are defined. We say a nonterminal is used, if it appears on the RHS of a production. In Fig. F2, nonterminals X2, X3, X5 and Ident are used.

The purpose of our Java program is to determine all used nonterminals that are not defined. In Fig. F2, nonterminals X5 and Ident are used but not defined.

3.3 Generic Solution

Notice that the problem formulation for the grammar problem is the formulation for the equation system problem subject to the following substitutions.

```
Equation system          equation        variable
Grammar                  production      nonterminal
```

We introduce the following generic terminology: System, Definition, Thing and Body, generalizing the equation system and grammar concepts as follows.

```
Equation system          equation                variable                expression
Grammar                  production              nonterminal             body

System                   Definition              Thing                   Body
```

The right-most column refers to the RHS of a definition. An alternative name for Body would be RHS. We are going to formulate our program in terms of the generic terminology. The purpose of the program is to report all used things that are not defined.

3.3.1 The Class Graph

Our first step is to identify a class graph that defines the structural relationships between the four concepts System, Definition, Thing and Body. We use the UML class diagram notation for the class graph, see Fig. 3. This class graph we will later stretch to fit into various application class graphs. To illustrate the stretching, we use the small class graph on the left of the Fig. 3 where S denotes System, D denotes Definition, B denotes Body and T denotes Thing.

3.3.2 The Traversals

We can already identify two traversals that we need for solving the problem. The first traversal, called usedThings, will traverse all things on the RHS and the second traversal, called definedThings, will traverse all things on the LHS.

Traversal usedThings we first describe as: usedThings1 = (System definitions Definition body Body things Thing) which means: go from a System-object through the has-a edge called definitions to all Definition-objects then through the has-a edge called body to Body-objects and finally through the has-a edge called things to all Thing-objects. Instead of this verbose traversal description, we prefer to use the following traversal specification:

```
usedThings = from System through Body to Thing
```

In AspectJ-terminology usedThings has the flavor of a pointcut: it defines all executions of a traverse method that is implicitly defined by the traversal specification.

For the class graph in Figure 3 the description usedThings1 and the specification usedThings have the same meaning. However, usedThings works for many more class graphs because it contains less information.

The traversal for finding the definedThings is expressed by the following traversal specification:

Fig. UML1

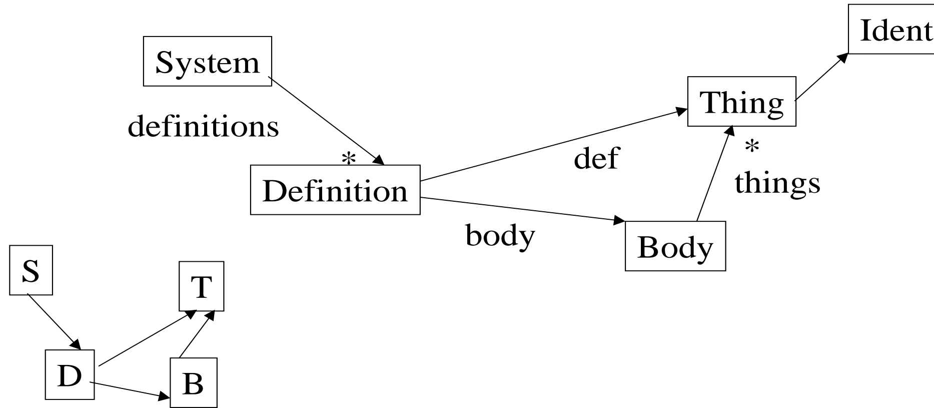

Figure 3: *Class graph for* System.

```
definedThings= from System bypassing Body to Thing
```

This is an example of a traversal specification that uses a bypassing clause.

3.3.3 The Java Program

Our Java program uses the following design: we have an aspectual method getDefThings(...) that uses the traversal specification definedThings and we write an aspectual method checkDefined(...) that uses traversal specification usedThings and the result of getDefThings(...) to check that all used things are defined and to report the undefined things. Finally, method reportUndef(...) calls both getDefThings(...) and checkDefined(...). The complete Java program for class System is in Fig. 4. It describes a collaboration between 5 kinds of objects: System, Definition, Body, Thing and Ident objects. The detailed relationships between those objects are left open.

3.4 Equation System Example revisited

We explain the Java program in the context of equation systems. We use the following name map that has to be applied manually by editing the code:

```
System : EquationSystem
Definition : Equation
Body : Expression
Thing : Variable
```

The class graph for class EquationSystem is in Fig. 5.

An ad-hoc implementation of aspectual method getDefThings(...) cuts across four classes (Figure 6). The crosscutting is shown in the figure by shading the classes that are croscut and by using five-pointed-stars on the has-a edges that are involved in the traversal.

```
class System{
  String id = "from Thing to edu.neu.ccs.demeter.Ident"; // pointcut
  void reportUndef(ClassGraph cg){
                 checkDefined(cg, getDefThings(cg));}
  HashSet getDefThings(ClassGraph cg){
                String definedThings = // pointcut specification
                "from System bypassing Body to Thing";
                Visitor v = new Visitor(){
                  HashSet return_val  = new HashSet();
                  void before(Thing v1){ // traversal advice
                          return_val.add(cg.fetch(v1, id) );}
                  public Object getReturnValue(){return return_val;}
                };
                cg.traverse(this, definedThings, v);
                return (HashSet)v.getReturnValue();
  }
  void checkDefined(ClassGraph cg, final HashSet classHash){
                String usedThings = // pointcut specification
                "from System through Body to Thing";
                cg.traverse(this, usedThings, new Visitor(){
                  void before(Thing v){ // traversal advice
                    Ident vn = (Ident) cg.fetch(v, id);
                    if (!classHash.contains(vn))
                        System.out.println("The object "+ vn
                        + " is undefined.");
                  }
                });
  }
}

class SystemClient {
  void use() {
    System s = new System(...);
    ClassGraph cg = new ClassGraph();
    s.reportUndef(cg);
  }
}
```

Figure 4: *Reusable System Checking. Class graph in Fig. 3*

Fig. Eq1

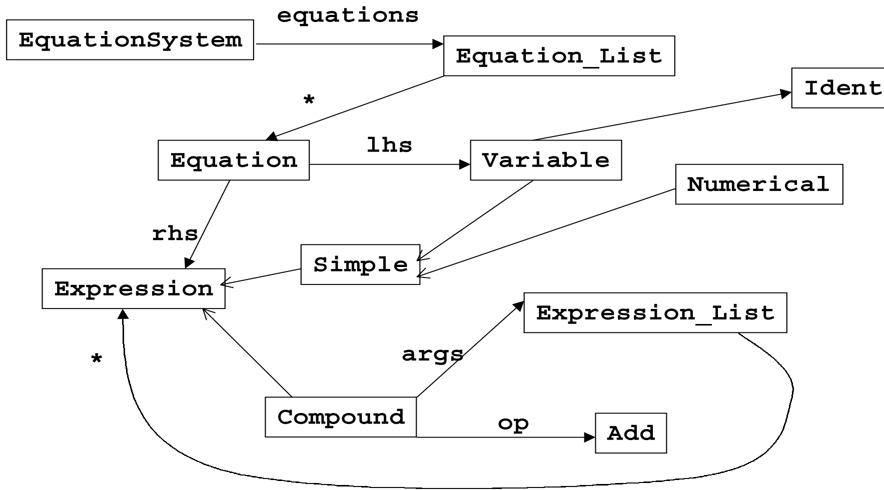

Figure 5: *Class graph for* EquationSystem.

Fig. Eq2

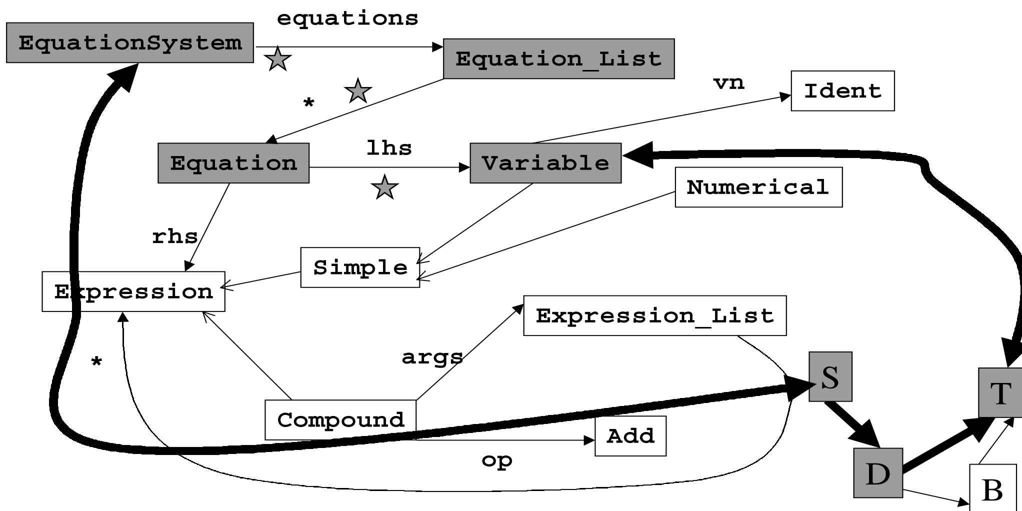

Figure 6: *Crosscutting of* getDefThings *caused by* from EquationSystem bypassing Expression to Variable.

Fig. Eq3

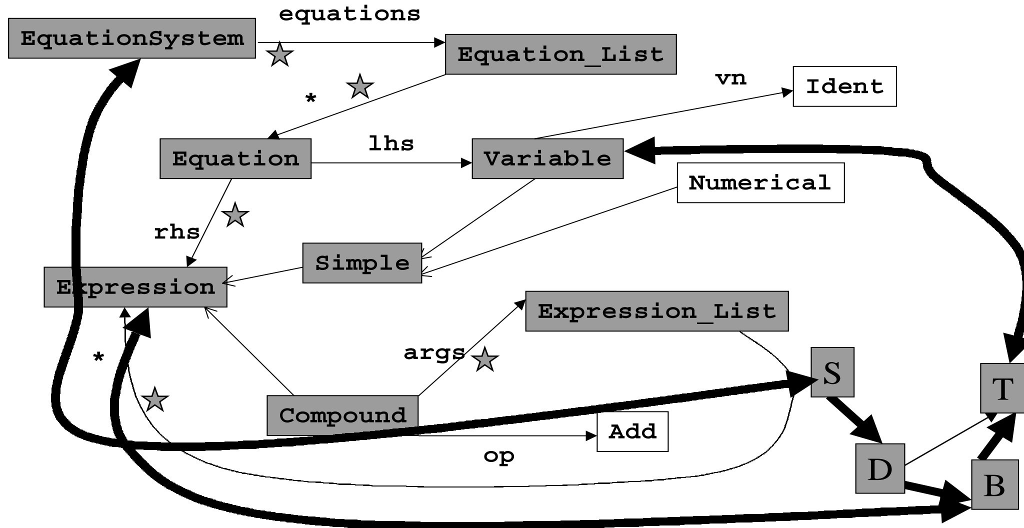

Figure 7: *Crosscutting of* checkDefined *caused by* from EquationSystem through Expression to Variable.

The method uses traversal specification definedThings together with an accumulation visitor that collects all defined things into a Java HashSet. An ad-hoc implementation of aspectual method checkDefined(...) cuts across eight classes (Fig. 7).

The method uses traversal specification usedThings to find all used things and to check whether each used thing was defined.

3.5 Grammar Example revisited

Next, we reuse the Java program in a different context: checking whether all nonterminals in a grammar are defined in the grammar. We use the following name map that has to be applied manually by editing the code:

```
System : Grammar
Definition : Production
Body : Body
Thing : NonTerm
```

Fig. 8 shows a class graph for simple grammars. An ad-hoc implementation of aspectual method getDefThings(...) cuts across four classes (Fig. 9). An ad-hoc implementation of aspectual method checkDefined(...) cuts across eight classes (Fig. 10).

Both the equation system and grammar example demonstrate the stretching of an aspectual method to fit into larger class graphs. In the equation and grammar examples the traversal specifications used in the aspectual methods are reused without change, except that the four names had to be substituted. In other examples we might have to refine the traversal specifications upon reuse in a new class graph.

A brief remark on the efficiency of Java programs written with the DJ package. Our current implementation makes heavy use of reflection which makes the programs slow. However, we have done initial work on a preprocessor of Java programs written with DJ that partially evaluates the programs when the class

Fig. G1

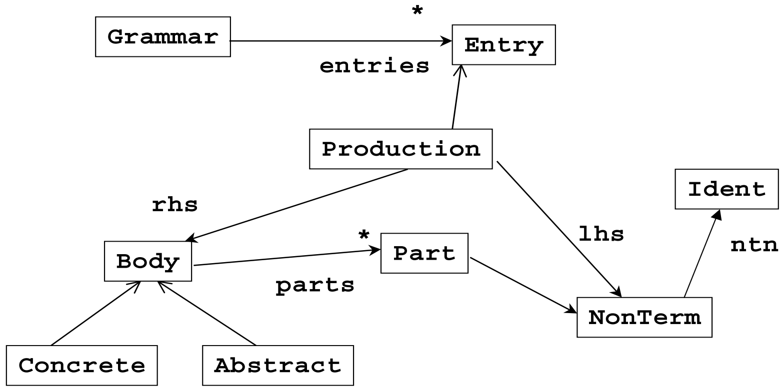

Figure 8: *Class graph for Grammars.*

Fig. G2

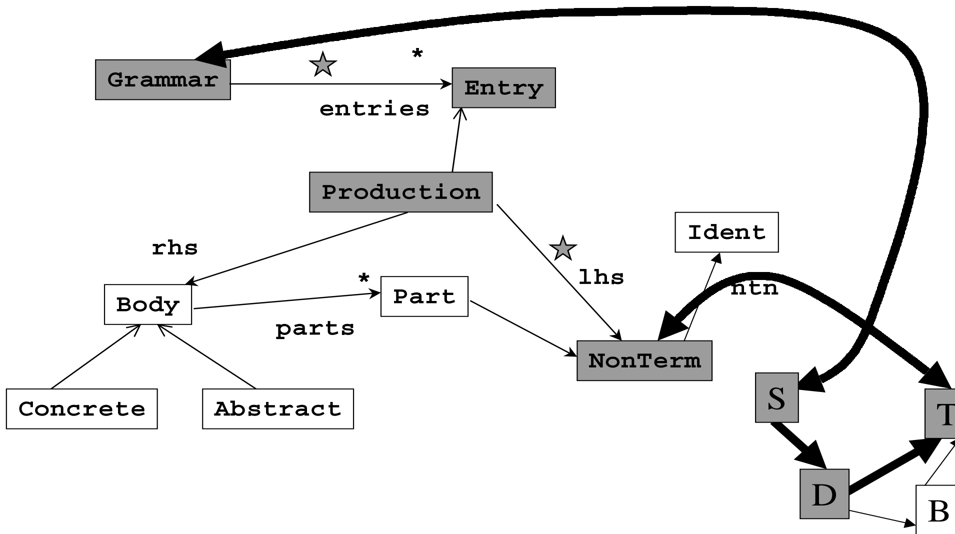

Figure 9: *Crosscutting of* getDefThings *caused by* from Grammar bypassing Body to NonTerm.

Fig. G3

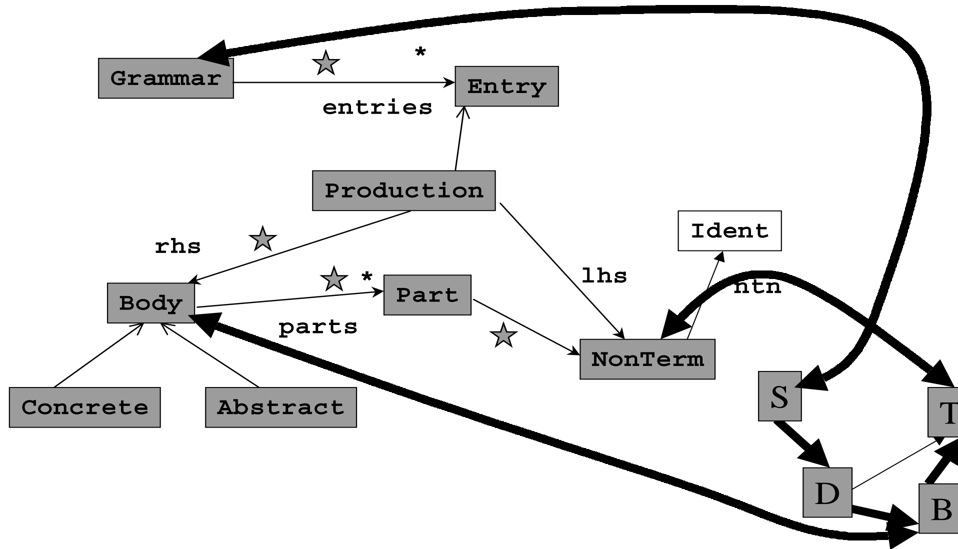

Figure 10: *Crosscutting of* checkDefined *caused by* from Grammar through Body to NonTerm.

graph and traversal specifications are known to be constant. This leads to much faster programs where the traversal code is generated and compiled. This work is an application of the DemeterJ techniques.

3.6 Caching

Of secondary conceptual importance are the DJ classes: ObjectGraph, ObjectGraphSlice and TraversalGraph. A traversal specification may be applied to both a ClassGraph-object and a Java object. If applied to a class graph, a traversal specification selects a subset of the paths in the ClassGraph-object. The subset of the paths is represented by a TraversalGraph-object where TraversalGraph is an additional DJ class that efficiently stores numerous reachability computations in the class graph. If applied to a Java object, a traversal specification defines a subgraph of the object graph representing the Java object. The subgraph is represented as an ObjectGraphSlice-object where ObjectGraphSlice is an additional DJ class.

Classes TraversalGraph and ObjectGraphSlice are available for convenience because in some programs the same TraversalGraph or ObjectGraphSlice may be reused many times. DJ also contains a class ObjectGraph that contains a Java object and a ClassGraph-object. Classes ClassGraph, TraversalGraph, ObjectGraph, and ObjectGraphSlice support the four methods called traverse, fetch, gather and asList.

3.7 Object graph slices and abstract point cuts

In the examples above, we parameterized aspectual methods by class graphs. In an alternative aspect-oriented programming style, we could parameterize the methods by an ObjectGraphSlice-object. Class ObjectGraphSlice is the largest class in the DJ package because many computations are reduced to computations on ObjectGraphSlice-objects. Recall that an ObjectGraphSlice-object consists of a Java object and a TraversalGraph-object that itself is constructed from a ClassGraph-object and a traversal specification. Conceptually, an ObjectGraphSlice-object is a subgraph of the object graph rooted at a Java object. The subgraph is determined by the part of the object that is traversed by the traversal specification. A formal

ObjectGraphSlice parameter has the flavor of an abstract pointcut in AspectJ. It names a set of execution points during the execution of a traversal method but does not give the details. An actual ObjectGraphSlice-object has the flavor of a concrete pointcut in AspectJ. It is determined by a traversal specification that can be thought of as a regular expression construct. In AspectJ, concrete pointcuts are defined by pointcut designators that might involve regular expressions.

So we have the following correspondence between AspectJ and DJ concepts, the DJ concepts being a special case of the AspectJ concepts:

```
AspectJ                      DJ

Abstract pointcut            ObjectGraphSlice formal parameter
Concrete pointcut            ObjectGraphSlice-object
```

4 Related Work

We first relate our work to AOP.

4.1 Aspects and AspectJ

AOP languages have three main elements: a join point model, a specification language for expressing sets of join points and a means of specifying behavior involving join points.

1. The DJ join point model is based on points in the execution of traversal functions defined by traversal specifications. The join point model has the following ingredients: The class graph model, the object graph model, the traversal specification model that defines the join points. In a first approximation the join points of DJ are the ingredients of an object graph with nodes (objects) and edges (links between objects). But the real join points are the execution points of a traversal function that traverses the object graph.

2. The DJ specification language for expressing join points is the traversal specification language in collaboration with the class graph language.

3. The DJ language for advice is the visitor language for specifying what should happen at each object and at each link.

The DJ concepts (and concepts of earlier versions of Demeter) are specializations of the AOP concepts.

While AspectJ is a general purpose language for AOP in Java, DJ only focuses on certain kinds of concerns. They include

- behavioral concerns that deal with groups of collaborating objects that offer new behavior (as opposed to modifying existing behavior),
- the traversal concern and
- the object structure concern.

Regarding the object structure concern, DJ actually frees the programmer from the details of this concern.

DJ complements AspectJ with point cut definition capabilities that are cumbersome to express in AspectJ alone without using the DJ package.

Filman and Friedman claim that aspect-oriented programming is quantification and obliviousness [8]. Aspectual methods are aspects also in this sense. When you write a Java program (the base program) you don't have to plan for the aspectual methods (the aspects) that you will write later. The base program is oblivious of the aspectual methods although writing the base program in a certain way may make it easier to express the traversal specifications involved in the aspectual methods. An aspectual method also involves quantification because you may use the same aspectual method with a large class of Java programs.

4.2 Other Related Work

The visitor design pattern [9] also proposes a visitor style of programming. But with DJ , the visitor pattern is so much easier to use because updates to the class structure become simpler. The scaffolding needed for the visitor design pattern is unnecessary because the visitor methods are discovered by reflection at runtime. DJ visitor objects are "clean" while visitor objects in the visitor design pattern are loaded with information from the class graph that makes changes to the class graph cumbersome.

[24] uses reflection to implement traversals. DJ improves on the earlier paper (1) by using the AP Library [29] that provides an efficient and general implementation of traversal specifications and (2) by implementing the idea in Java without extension and (3) by reifying the concepts directly as Java classes.

Hyper/J [32] plays in a similar area as DJ but is is more general. Hyperslices are expressing enhancements to abstract join point sets while hypermodules play the role of expressing sets of join points by mapping the abstract join point sets to concrete ones.

The Demeter research group has not only explored the idea of aspectual methods but also the idea of aspectual components [27, 15]. Aspectual components are a construct that extends Java.

Aspectual methods are sisters of the adaptive methods of DemeterJ (formerly called Demeter/Java; renamed because of trademark issues) [19, 7]. However, adaptive methods use syntax outside of Java and are more powerful while aspectual methods are Java methods importing the DJ package.

Aspectual methods are (completely specified) collaborations in the UML sense [3]. A collaboration consists of class diagram and a behavioral part. The class diagram defines the roles and their structural relationships. In case of an aspectual method, the roles are the classes mentioned in traversal specifications and the classes in visitors that have before/after/around methods. Aspectual methods cover a broad and useful class of collaborations.

5 Acknowledgements

Many thanks to Joshua Marshall for the initial development of DJ and to Pengcheng Wu for adding several features to DJ.

6 Summary

In this paper we have demonstrated how we can practice an interesting kind of aspect-oriented programming in Java by using the Java package edu.neu.ccs.demeter.DJ. The approach allows us to write Java methods whose ad-hoc implementation would cut across many classes. We call those methods aspectual methods. They cleanly separate behavioral concerns that involve multiple, collaborating objects from each other and from the details of the class graph. We have also demonstrated how we can use the DJ library to stretch a small class graph to fit into a larger class graph. This capability allows us to implement patterns of similar behavior in a reusable form.

We believe that the class graph-based pointcut definition mechanisms presented in this paper are a useful technique for both AspectJ and Java. It would be useful to consider a better integration with the pointcut designators of AspectJ. For example, a traversal specification can be used to specify a set of objects in a structure-shy way. AspectJ already has the instanceof pointcut designator where instanceof(T) means all join points at which the currently executing object is an instance of T or of one of T's subtypes. It would be useful to have a pointcut designator instanceof(S) where instanceof(S) means all join points at which the currently executing object is an instance of T or of one of T's subtypes where T is in the traversal of traversal specification S.

References

[1] M. Aksit and A. Tripathi. Data abstraction mechanisms in Sina/ST. In *Object-Oriented Programming Systems, Languages and Applications Conference,* in *Special Issue of SIGPLAN Notices*, number 11, pages 265–275. ACM, November 1989.

[2] Ken Anderson and Dean Allemang. Demeter/CLOS. In *http://www.ccs.neu.edu/home/lieber/outside-impl.html*. Northeastern University, 1997-1998.

[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison Wesley, 1999. ISBN 0-201-57168-4.

[4] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen (eds.). Extensible Markup Language. http://www.w3.org/TR/REC-XML, February 1998.

[5] James Clark and Steve DeRose (eds.). XML Path Language (XPath), version 1.0. http://www.w3.org/TR/XPath, November 1999.

[6] Demeter Research Group. Online Material on Adaptive Programming and Demeter. In *http://www.ccs.neu.edu/research/demeter/*. Northeastern University, 1989-2001.

[7] Joshua Marshall Doug Orleans, Johan Ovlinger, Kedar Patankar, Binoy Samuel, and Karl Lieberherr. DemeterJ. Technical report, Northeastern University, December 1998. http://www.ccs.neu.edu/research/demeter/DemeterJava/.

[8] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA*, Minneapolis, USA, 2000. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/aop-is.pdf.

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

[10] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 411–428, October 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.

[11] Outside Demeter implementors. Demeter Implementations. In *http://www.ccs.neu.edu/home/lieber/outside-impl.html*. 1997-1998.

[12] G. Kiczales, J. Des Rivière, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[13] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, (1), January 1996.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mike Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In Jorgen Knudsen, editor, *European Conference on Object-Oriented Programming*, Budapest, 2001. Springer Verlag.

[15] 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.

[16] Karl J. Lieberherr. Component enhancement: An adaptive reusability mechanism for groups of collaborating classes. In J. van Leeuwen, editor, *Information Processing '92, 12th World Computer Congress*, pages 179–185, Madrid, Spain, 1992. Elsevier.

[17] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, ISBN 0-534-94602-X, entire book at www.ccs.neu.edu/research/demeter.

[18] Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[19] Karl J. Lieberherr and Doug Orleans. Preventive program maintenance in Demeter/Java (research demonstration). In *International Conference on Software Engineering*, pages 604–605, Boston, MA, 1997. ACM Press.

[20] Karl J. Lieberherr and Boaz Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, Sep. 1997. http://www.ccs.neu.edu/research/demeter/AP-Library/.

[21] Cristina Isabel Videira Lopes. *D: A Language Framework for Distributed Programming.* PhD thesis, Northeastern University, 1997. 274 pages.

[22] Cristina Videira Lopes. Graph-based optimizations for parameter passing in remote invocations. In Luis-Felipe Cabrera and Marvin Theimer, editors, *4th International Workshop on Object Orientation in Operating Systems*, pages 179–182, Lund, Sweden, August 1995. IEEE, Computer Society Press.

[23] Cristina Videira Lopes. Adaptive parameter passing. In *2nd International Symposium on Object Technologies for Advanced Software*, pages 118–136, Kanazawa, Japan, March 1996. Springer-Verlag.

[24] Cristina Videira Lopes and Karl Lieberherr. AP/S++: case-study of a MOP for purposes of software evolution. In *Reflection '96*, S. Francisco, CA, April 1996.

[25] Cristina Videira Lopes and Karl J. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In Remo Pareschi and Mario Tokoro, editors, *European Conference on Object-Oriented Programming*, pages 81–99, Bologna, Italy, 1994. Springer Verlag, Lecture Notes in Computer Science.

[26] Joshua Marshall, Doug Orleans, and Karl Lieberherr. DJ: Dynamic Structure-Shy Traversal in Pure Java. Technical report, Northeastern University, May 1999. http://www.ccs.neu.edu/research/demeter/DJ/.

[27] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. Technical Report NU-CCS-98-3, Northeastern University, April 1998. To appear in OOPSLA '98.

[28] Doug Orleans and Karl Lieberherr. DemeterJ. Technical report, Northeastern University, 1996-2001. http://www.ccs.neu.edu/research/demeter/DemeterJava/.

[29] Doug Orleans and Karl Lieberherr. AP Library: The Core Algorithms of AP. Technical report, Northeastern University, May 1999. http://www.ccs.neu.edu/research/demeter/AP-Library/.

[30] Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, March 1995.

[31] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin-layers. In *European Conference on Object-Oriented Programming*. Springer Verlag, 1998.

[32] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, Los Angeles, 1999. ACM.

[33] Xerox PARC AspectJ team. AspectJ home page. http://aspectj.org. Continuously updated.