

Applying Traversals Over Derived Edges

Fabio Rojas and Karl Lieberherr

College of Computer and Information Sciences
Northeastern University
360 Huntington Avenue
Boston MA 02115
{frojas,lieber}@ccs.neu.edu

Abstract. DJ is a Java tool that allows developers to introduce the visitor pattern into existing Java programs without requiring them to modify their existing classes. DJ uses reflection and Demeter’s adaptive traversal specification language to create a traversal of an object graph. As DJ traverses a class graph it calls the appropriate methods of the visitors provided to it. This combination of the visitor pattern and the adaptive traversals is also known as the adaptive visitor pattern. This paper presents a series of extensions that increase DJ’s traversal capabilities. These extensions allow the adaptive traversal to call the methods of objects within the object graph being traversed. DJ can then traverse the return values of these method calls. This functionality can come into play when a traversal’s desired behavior would result in the addition of objects to the object graph that is being traversed. Three different strategies for providing the arguments to the derived edges are discussed.

1 Introduction

DJ[11] is a Java library that uses reflection[13] to implement the traversal capabilities available in the Demeter model[6, 10]. The classes `ClassGraph`, `Strategy` and `Visitor` provide the programmer with the Demeter API to write Java programs which are separated into three concerns: behavioral, structural, and traversal. DJ is implemented in pure-Java using reflection and can be used to implement the visitor pattern[5] over an already existing class graph without the need to modify the class graph.

DJ uses the notion of a traversal strategy to specify the paths that it will take during its traversal of an object graph. When a traversal is given a source object and a visitor, the traversal will visit each object reachable from the source object that is on the path to an object of the target class. As each object is visited the appropriate methods within the visitor will be called. The combination of traversal strategies and the use of visitors is known as the adaptive visitor pattern[10]. The pattern is adaptive because the traversal are able to adapt to changes with the class graph.

Until now DJ has been limited with respect to the edges that it can travel through during its traversals. DJ could only consider edges in the class graph that

correspond to fields. DJ also had the capability to consider methods that take no arguments and had a non-void return type as edges eligible for traversal. DJ would call the method and would then continue the traversal through the return value of the method call. We call these method edges, derived edges because the object that they point to is derived from the result of a method call. DJ was not able to treat methods of arity greater than zero as derived edges. This limitation meant that either all the objects that would be traversed must be present when the traversal was initiated, or they could only be instantiated as the result of the call to a method that takes no arguments.

The ability to traverse derived edges corresponding to methods of arity greater than zero can be useful when part of the object graph being traversed is not instantiated before the traversal begins. JDBC[4] and other Java API's[3] provide methods that transmit queries to remote databases and which return the results of the queries as a collection of objects that can be accessed by the program that made the method call. These objects provide a way to access the information retrieved from the database. The ability to traverse derived edges would allow DJ to call methods which retrieve data from remote servers and to traverse down the objects which encapsulate that data.

We present an extension to DJ that allows the traversals to proceed down all derived edges. In order to achieve this DJ's traversal engine was augmented to use three new types of adaptive visitors. These new visitors provide a means for delivering arguments to derived edges. Once the arguments are provided the return values of these edges can be traversed.

2 Using DJ

A UML diagram for a simple class graph is shown in Figure 1. Figure 2 shows an object graph belonging to this class graph. We will use these graphs to illustrate capabilities of DJ and to provide a motivating example for the features that we have added to DJ.

2.1 Specifying Traversals Strategies

Traversal strategies are specified using a subset of the Demeter traversal language. A strategy specifies a source class, a target class, and an optional set of qualifiers on the traversal[11]. These qualifiers serve to limit the scope of the traversal. The traversal "From Hotel to *" will indicate to DJ that its traversal should touch every object that is reachable from a source Hotel object. The wildcard * can be used in place of a class name in most circumstances.

A traversal can be restricted to travel only through paths that will eventually lead to Promotion objects by using the strategy "from Hotel to Promotion". For example, an object of class ReservationDB can never lead to an object of class Promotion, so DJ would not traverse down a ReservationDB object.

Traversals can also be restricted through any number of *via* and *bypassing* clauses. These clauses tell DJ to only consider paths that go through a particular

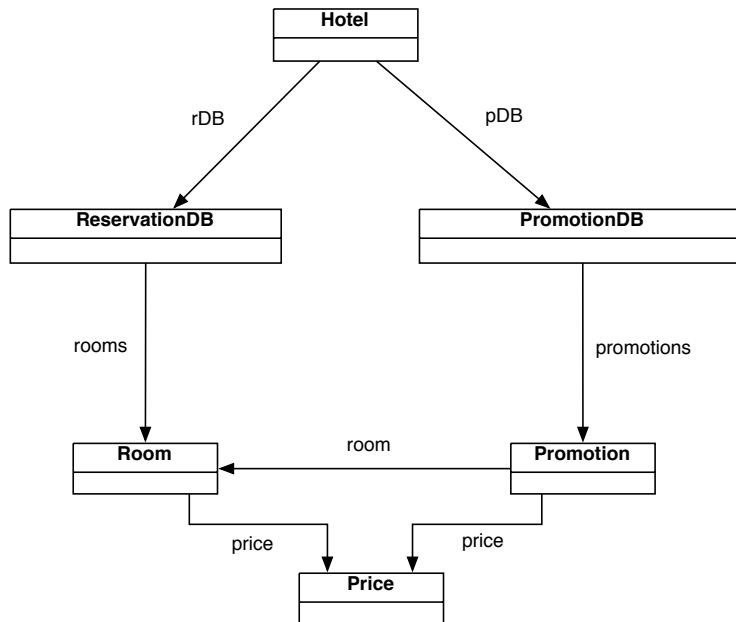


Fig. 1. A simple Class Graph

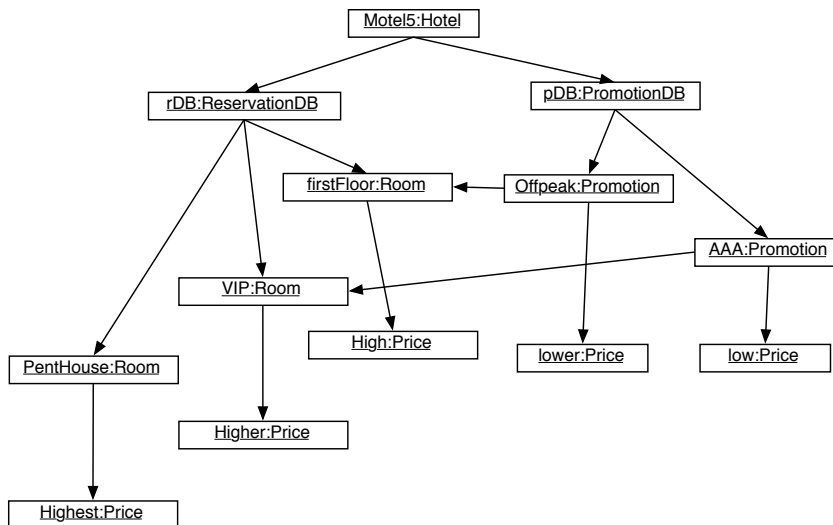


Fig. 2. A sample object graph

edge or class, or to avoid paths that go through a particular edge or class. The strategy "from Hotel via ReservationDB to Room" will force DJ to only consider paths that go through a ReservationDB object. The strategy

"from Hotel bypassing PromotionDB to Room"

produces the same results, since it restricts the traversal to only consider paths that do not travel through a PromotionDB object. Thus the only viable paths from a Hotel object to a Room object must travel through a ReservationDB object.

Edges are specified using a notation that indicates the class that edge belongs to, the edge name, and the type of the object the edge points to. The strategy:

"from Hotel bypassing ->Hotel,rDB,* to Price"

will tell DJ to perform a traversal from a Hotel object to all Price objects while bypassing all the edges that start at a node of type Hotel, whose label (field name) is rDB and which point to any target type.

2.2 Using Traversals

Traversal specifications are used by DJ, along with a ClassGraph object and any number of visitors to perform the actual traversal. Listing 1.1 provides an example of a visitor that can be used with DJ to display the path taken during a traversal. A sample use of the DisplayVisitor class can be seen in Listing 1.2. This example uses the traversal strategy: "from Hotel bypassing PromotionDB to Price". Listing 1.3 shows the result of performing the traversal with motel5 as its source (the output has been indented for easier reading).

Listing 1.1. The DisplayVisitor class

```
public class DisplayVisitor extends Visitor {
    public void before(Object o) {
        System.out.println("Before: " + o);
    }
    public void after (Object o) {
        System.out.println("After: " + o);
    }
}
```

Listing 1.2. Using a DisplayVisitor with a Traversal Strategy

```
public void display(Hotel Motel5) {
    String strategy = "from Hotel bypassing PromotionDB to Price";
    DisplayVisitor dv = new DisplayVisitor();
    ClassGraph cg = new ClassGraph();
    cg.traverse(Motel5, strategy, dv);
}
```

Listing 1.3. DisplayVisitor's Output

```

Before: Hotel: Motel5
  Before: ReservationDB: rDB
    Before: [Room: PentHouse, Room: VIP, Room: First Floor]
      Before: Room: PentHouse
        Before: Price: Highest
        After: Price: Highest
      After: Room: PentHouse
    Before: Room: VIP
      Before: Price: Higher
      After: Price: Higher
    After: Room: VIP
  Before: Room: First Floor
    Before: Price: High
    After: Price: High
  After: Room: First Floor
  After: [Room: PentHouse, Room: VIP, Room: First Floor]
After: ReservationDB: rDB
After: Hotel: Motel5
  
```

2.3 Set Operations

DJ not only allows us to traverse the current object graph, it also provides the ability to gather all the target objects into a collection that can then be examined after the traversal has been completed. For example, the following call:

```
cg.gather(Motel5, "From Hotel to Room")
```

will return a collection of all objects of class `Room` that are reachable from `Motel5`.

DJ also has the ability to take two traversal specifications and to return the target objects that were reachable by both traversals. Listing 1.4 shows a sample use of the `ClassGraph.targetIntersection()` method. After the code in the figure is executed the `List set1` will hold all objects that are reachable from `Motel5` using both the `s1` and `s2` traversal strategies. This is different from the concept of path intersection used in DAJ[14] where intersection means that only paths that would be traversed by both traversals are considered.

The other set operations provided are `ClassGraph.setDifference()` which performs set difference operations and `ClassGraph.union()` which performs the set union operation.

Listing 1.4. Using the `Classgraph.targetIntersection()` method.

```

String s1 = "From Hotel via ReservationDB to Room";
String s2 = "From Hotel via PromotionDB to Room";
List set1 = cg.targetIntersection(Motel5, s1, s2);
  
```

3 Traversals over Derived Edges

DJ provides the ability to traverse object graphs, but its traversals are limited to following branches only through derived edges that correspond to methods that take no arguments. By providing the ability to traverse down the return values of method edges we can instantiate new objects during the traversal. Thus the object graph can grow during the traversal in way that is parameterized by the arguments passed to the derived edges encountered during the traversal. This is useful feature when part of the object graph refers to objects that serve as an interface to some back end data storage system, such as a database or network connection.

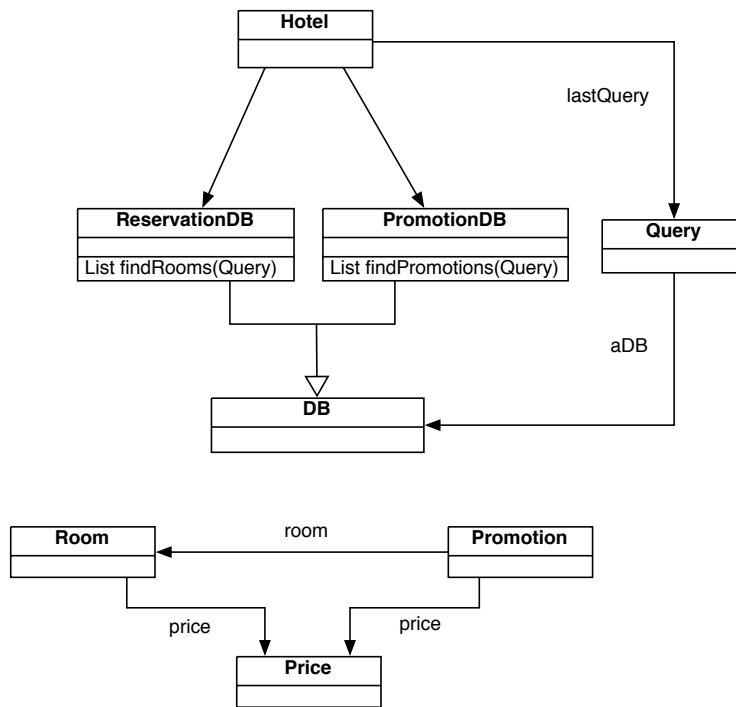


Fig. 3. A modified Class Graph

In this class graph, **DB** and its subclasses provide an interface to a database that stores the room, promotion, and price information. The objects of class **DB** provide methods for submitting queries to the database. The results of these queries are then returned in the form of collections containing newly instantiated objects that encapsulate the results. Being able to traverse derived edges will allow us to use DJ to specify traversals that are able to navigate these query results.

3.1 Argument Visitors

As originally implemented DJ already possessed the ability to traverse down a limited set of derived edges. DJ used reflection to call the zero argument methods if the return type of the method could be a node on the path to the target of a traversal.

We extended this functionality by introducing a new type of adaptive visitor. This visitor is embodied in a subclass of `Visitor` which we call `ArgumentVisitor`. Since the `ArgumentVisitor` class extends the `Visitor` it can use the standard `before()` and `after()` methods but each `ArgumentVisitor` instance also has an argument map that provides a map from a class and method name or an object and method name to an argument list. The argument map can be populated at the time that the `ArgumentVisitor` is instantiated. The maps can also be manipulated during the traversal through the visitor's `before()` and `after()` methods. If all the arguments needed to travel down a particular derived edge are not known then that edge will not be traversed and the next potential path to a target object will be examined.

Using an `ArgumentVisitor` increases the number of edges that DJ can traverse, but it can lead to code that is less structure shy. Normally traversals are robust when it comes with dealing with changes in the class graph. `ArgumentVisitors` increase the dependency on the current class graph since they rely on particular method and class names. Changing the class graph is now more likely to results in programs that still run but that use traversals that no longer compute the correct results.

Listing 1.5. Using an `ArgumentVisitor`

```
public List findRooms(Hotel aHotel, Query aQuery) {
    String strategy = "from Hotel to Rooms";
    ArgumentVisitor av = new ArgumentVisitor();

    // Add a map for the ReservationDB.findRooms() method
    av.addMap("ReservationDB", "findRooms", new Object[] {aQuery});

    ClassGraph cg = new ClassGraph();
    return cg.gather(aHotel, strategy, av);
}
```

An example demonstrating the usage of an `ArgumentVisitor` is given in Listing 1.5. This method takes an object of class `Hotel` and an object of class `Query` and gathers all the `Room` objects that match that query. The strategy that is used will take DJ from a `Hotel` object to all the `Room` objects that are reachable from it. In the class graph on Figure 3 there are no edges that lead from `Hotel` to `Room`. But the derived edges that correspond to `ReservationDB.findRooms()` and `ReservationDB.findPromotions()` can lead to a `List` object. In Java classes such as `List` and `Vector` contain a set objects of class `Object`. Since all classes inherit from

the `Object` class these collection objects might contain `Room` objects. DJ will always act conservatively and will traverse down any edge that can potentially lead to a `Room` object. Thus if it is given an argument map for one of the two methods mentioned earlier, it will call that method.

The method call:

```
av.addClassMap("ReservationDB", "findRooms", new Object[ ] aQuery);
```

creates a new argument map. Whenever an object of type `ReservationDB` is encountered its `findRooms()` method will be executed with `aQuery` as its argument. The `ArgumentMap.addClassMap()` method creates a map that can be used with all objects of the class specified in its first argument. The class can be specified with either its fully qualified name, with the appropriate `java.lang.Class` object, or with an object of the desired type. The method that the map belongs to can be specified either by providing its name as a string, or by using the appropriate `java.lang.reflect.Method`. The argument list can be either provided as an array or a `Vector`.

The method call:

```
av.addObjectMap(aHotel.rDB, "findRooms", new Object[ ] aQuery);
```

creates the same argument map except that it is only used when the object referenced by `aHotel.rDB` is encountered during the traversal. If an object is encountered that argument map specified by both `ArgumentVisitor.addClassMap()` and `ArgumentVisitor.addObjectMap()` then the map provided by latter is used.

The argument maps can also be populated by defining new subclasses of `ArgumentVisitor`. The new subclass can then populate its argument maps in its constructors or within its `before()` and `after()` methods.

3.2 Context Visitors

`ArgumentVisitors` provide a way of delivering arguments to derived edges, but they require a certain amount of overhead. The argument maps must be populated during the traversal. In some cases this is unnecessary because the traversal is already aware of the objects that will be passed in as arguments to the derived edges. If a traversal has already traveled through a node then it might be beneficial to have the `ArgumentVisitor` consider the history or context of the traversal when determining whether or not it has enough information to travel down a derived edge. Two types of adaptive visitors have been developed which build argument maps from the context of the traversal.

The first of these new visitors is the `ContextVisitor`. A `ContextVisitor` uses a set of stacks one for each class that the traversal has encountered to maintain the context of the traversal. Before an object of a particular class is traversed a reference to it pushed unto the stack for its class. After the object has been traversed it is popped from the appropriate stack. These stacks are the context of the traversal. A `ContextVisitor` extends the `ArgumentVisitor` class so it is aware of the context of the traversal but it also has the set of argument maps that all `ArgumentVisitors` have. If DJ decides that traversing down a derived edge can lead to a target object, it first checks to see if it has a fully populated argument map for that edge. If not it will try and create one using the traversal context. If the

`ContextVisitor`'s stacks have enough objects of the proper types it will provide DJ with an argument list. This list will be used to call the method. If both methods of building the argument list fail then the edge will not be traversed.

Listing 1.6. Using a `ContextVisitor`

```
public List redoLastQuery(Hotel aHotel, Query aQuery) {
    String strategy = "from Hotel via ->Hotel,lastQuery,Query to Rooms";
    ContextVisitor cv = new ContextVisitor();

    ClassGraph cg = new ClassGraph();
    return cg.gather(aQuery, strategy, cv);
}
```

Listing 1.6 demonstrates the use of a `ContextVisitor`. The traversal strategy will go from a `Hotel` object to all `Room` objects, that are reachable from a `Hotel`'s `lastQuery` field. When the traversal reaches the `DB` object reachable through `aQuery`'s `aDB` field the `Query` object will be in the traversal's context. This object will be used as an argument to call either the `findRooms()` or the `findPromotions()` method, depending on whether the object is of class `ReservationDB` or `PromotionDB`.

The `AfterContextVisitor` is the second type of visitor that can be used to build argument lists. While a `ContextVisitor` only keeps track of objects that lie within the current branch of the traversal, the `AfterContextVisitor` keeps track of all objects that have been traversed so far. An `AfterContextVisitor` tries to build an argument list using both information about the current branch of the object graph being traversed and its knowledge of history of the traversal along other branches.

Using an `AfterContextVisitor` can be problematic because its behavior can be subject to nondeterminism. DJ explores edges by using reflection. The order in which the Java reflection API lists the fields and methods within classes is undefined[13], meaning that relying on a specific order for the edge traversals is not possible from one Java implementation to another.

Using a `ContextVisitor` or an `AfterContextVisitor` will most likely introduce inefficiencies into the program because each step of traversal will involve a certain amount of popping and pushing onto the stacks that they contain. We felt that at this time it was more important to concentrate on developing models for creating the contexts for the traversals than in improving their efficiency. An implementation of the traversal engine generates its traversal logic statically as opposed to relying on reflection, such as DAJ[14, 12], could use static information about the class graph to remove unnecessary stack operations.

Case studies need to be conducted to determine if the `ContextVisitor` and the `AfterContextVisitor` approaches for automatically building the argument lists is useful in practice. We can also imagine that there are several other methods for building argument lists that might be useful to developers. These can be implementing by creating new subclasses of `ArgumentVisitor`.

4 Related Work

One of HydroJ's goals is to address the brittle parameter problem[8]. HydroJ address the problem by using semi-structured, self-describing messages. These messages allow systems built using HydroJ to be robust in terms of changing data definitions. If the structure of the messages that a component of the system receives changes in a way that is orthogonal to the component's needs, then that component can use the self-describing nature of the message to extract the information it needs. Both HydroJ and DJ try to insulate systems from changes to the class graph. HydroJ achieves this by specifying a pattern language that can be used to specify the structure of messages that a method expects, while DJ uses traversals to remain insulated from changes in the class graph.

The concepts of essential structure and implicit context[15] try to address similar issues as DJ. DJ uses traversals to make code reusable and resistant to changes in the class graphs within the underlying system. Essential structure and implicit context try to address these issues by allowing a module to view the rest of the system in a manner distinct from the way in which the other modules view the system. The differences in views are then reconciled by the implicit context. Since each module has its own view independent of the other modules it will not be affected by the changes in the these other modules. DJ allows different components to have different view of the system. These views are expressed in the form of the traversal strategies that the components have access to.

Different techniques have being introduced to aid in the development of middleware-shy[1, 2]. Software that is middleware-shy can be used with different middleware systems that implement the same interfaces. DJ's traversal based approach allows for the development of software that is easy to reuse with different middleware systems. By specifying traversals which do not refer to classes from a particular middleware package the traversals will be reusable with other packages. In order to effectively use middleware we felt that it was necessary to provide a mechanism to traverse derived edges.

5 Summary and Future Work

We have presented an extension to DJ that permits traversals to travel down all the derived edges within the object graphs being traversed. Traversals over derived edges can be useful when the system being developed uses API's that allow it receive data stored within remote servers. If a traversal takes a path that leads it down a derived edge then the traversal will be examining objects which which encapsulate the data that is stored remotely.

Our extensions to DJ are available for download at: <http://www.ccs.neu.edu/home/frojas/DJ.tgz>

A subject of previous study has been the semantics of traversals over non-derived edges[9]. We want to extend the framework that has been laid out to encompass traversals over derived edges.

Recent efforts by the Demeter Group[6] have resulted in a port of the core DJ functionality to the Microsoft Common Language Runtime(CLR). This means that any of the over dozen languages that can run on the CLR can take advantage of the adaptive visitor pattern. Once our extensions have been integrated into the release versions of DJ, we will proceed to port the extensions to CLR version of DJ.

DAJ[14, 12] is an extension of AspectJ[7] that allows programmers to use the Demeter traversal model while working with AspectJ. DAJ does not support the extensions presented in this paper. In the future we plan to explore the possibility of adding the derived edge traversal functionality to DAJ.

Acknowledgements

We would like to acknowledge work of Doug Orleans, the implementor of DJ. We also thank Therapon Skotiniotis and Johan Ovlinger for lending us their typesetting expertise. We would like to thank Adrial Colyer for proposing the middleware-shy project <http://www.ccs.neu.edu/home/lieber/ap-projects.html>. This paper was partially motivated by the concept of middleware-shy systems.

References

1. Gordon S. Blair and Geoff Coulson. The case for reflective middleware.
2. Ron Bodkin, Adrian Colyer, and Jim Hugunin. Applying AOP for Middleware Platform Independence. <http://aosd.net/archive/2003/program/bodkin.pdf>.
3. Linds G. DeMichiel. *Enterprise JavaBeans Specification Version 2.1*. Sun Microsystems, Inc., November 2003.
4. Jon Ellis and Linda Ho with Maydene Fisher. *JDBC 3.0 Specification*. Sun Microsystems, Inc., October 2001.
5. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
6. Demeter Research Group. <http://www.ccs.neu.edu/demeter>.
7. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In Jorgen Knudsen, editor, *European Conference on Object-Oriented Programming*, Budapest, 2001. Springer Verlag.
8. Keunwoo Lee, Anthony LaMarca, and Craig Chambers. Hydroj: object-oriented pattern matching for evolvable distributed systems. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 205–223. ACM Press, 2003.
9. Karl Lieberherr and Mitchell Wand. Navigating through object graphs using local meta-information. Technical Report NU-CCS-2001-05, Northeastern University, May 2001. <http://www.ccs.neu.edu/research/demeter/biblio/new-strategy-semantics.html>.
10. 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.

11. Doug Orleans and Karl Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag. 8 pages.
12. Doug Orleans and Karl Lieberherr. DAJ: Demeter in AspectJ. Technical report, Northeastern University, January 2003. <http://www.ccs.neu.edu/research/demeter/DAJ/>.
13. Inc. Sun Microsystems. Package `java.lang.reflect`. <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/reflect/package-summary.html>.
14. John Sung and Karl Lieberherr. DAJ: A Case Study of Extending AspectJ. Technical Report NU-CCS-02-16, Northeastern University, November 2002.
15. Robert J. Walker. *Essential Software Structure through Implicit Context*. PhD thesis, The University of British Columbia, 2003.